Lecture Notes in Computer Science 1965
Edited by G. Goos, J. Hartmanis and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Tokyo

Çetin K. Koç Christof Paar (Eds.)

Cryptographic Hardware and Embedded Systems – CHES 2000

Second International Workshop
Worcester, MA, USA, August 17-18, 2000
Proceedings

Springer

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Volume Editors

Çetin K. Koç
Oregon State University
Corvallis, Oregon 97331, USA
E-mail: koc@ece.orst.edu

Christof Paar
WPI, ECE Dept., Cryptography and Information Security (CRIS) Group
100 Institute Rd., Worcester, MA 01609, USA
E-mail: christof@ece.wpi.edu

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Cryptographic hardware and embedded systems : second international workshop ; proceedings / CHES 2000, Worcester, MA, USA, August 17 - 18, 2000. Çetin K. Koç ; Christof Paar (ed.). - Berlin ; Heidelberg ; New York ; Barcelona ; Hong Kong ; London ; Milan ; Paris ; Singapore ; Tokyo : Springer, 2000
(Lecture notes in computer science ; Vol. 1965)
ISBN 3-540-41455-X

CR Subject Classification (1998): E.3, C.2, C.3, B.7.2, G.2.1, D.4.6, K.6.5, F.2.1, J.2

ISSN 0302-9743
ISBN 3-540-41455-X Springer-Verlag Berlin Heidelberg New York

Springer-Verlag Berlin Heidelberg New York
a member of BertelsmannSpringer Science+Business Media GmbH

Printed in Germany

Typesetting: Camera-ready by author, data conversion by Christian Grosche, Hamburg
Printed on acid-free paper SPIN: 10781072 06/3142 5 4 3 2 1 0

Preface

These are the pre-proceedings of CHES 2000, the second workshop on Cryptographic Hardware and Embedded Systems. The first workshop, CHES'99, which was held at WPI in August 1999, was received quite enthusiastically by people in academia and industry who are interested in hardware and software implementations of cryptography. We believe there has been a long-standing need for a workshop series combining theory and practice for integrating strong data security into modern communications and e-commerce applications. We are very glad that we had the opportunity to serve this purpose and to create the CHES workshop series.

As is evident by the papers in these proceedings, there have been many excellent contributions. Selecting the papers for this year's CHES was not an easy task, and we regret that we had to reject several good papers due to the limited availability of time. There were 51 submitted contributions to CHES 2000, of which 25 were selected for presentation. This corresponds to a paper acceptance rate of 49% for this year, which is a decrease from the 64% acceptance rate for CHES'99. All papers were reviewed. In addition to the contributed presentations, we have invited two speakers.

As last year, the focus of the workshop is on all aspects of cryptographic hardware and embedded system design. Of special interest were contributions that describe new methods for efficient hardware implementations and high-speed software for embedded systems, e.g., smart cards, microprocessors, DSPs, etc. In addition, there were again several very interesting and innovative papers dealing with cryptanalysis in practice, ranging from side-channel attacks to FPGA-based attack hardware.

We hope to continue to make the CHES workshop series a forum of intellectual exchange in creating secure, reliable, and robust security solutions for tomorrow. CHES workshops will continue to deal with hardware and software implementations of security protocols and systems, including security for embedded, wireless Internet access applications.

We thank everyone whose involvement made the CHES workshop such a successful event. In particular we would like to thank Dan Bailey, Adam Elbirt, Jorge Guajardo, Linda Looft, Jennifer Parissi, Francisco Rodriguez, Andre Weimerskirch, and Adam Woodbury.

August 2000

Çetin K. Koç
Christof Paar

Acknowledgements

The program chairs express their thanks to the program committee, the referees for their help in getting the best quality papers selected, and the companies that supported the workshop.

The program committee members:

- Gordon Agnew <g.agnew@coulomb.uwaterloo.ca>
  University of Waterloo, Canada
- Wayne Burleson <burleson@galois.ecs.umass.edu>
  University of Massachusetts at Amherst, USA
- Kris Gaj <kgaj@gmu.edu>
  George Mason University, USA
- Peter Kornerup <kornerup@imada.sdu.dk>
  Odense University, Denmark
- Arjen Lenstra <arjen.lenstra@citicorp.com>
  Citibank, USA
- Jean-Jacques Quisquater <jjq@dice.ucl.ac.be>
  Université Catholique de Louvain, Belgium
- Patrice Roussel <proussel@ichips.intel.com>
  Intel Corporation, USA
- Christoph Ruland <RULAND@nue.et-inf.uni-siegen.de>
  Universität Siegen, Germany
- Joseph Silverman <jhs@ntru.com>
  Brown University and NTRU Cryptosystems, Inc., USA
- Colin Walter <C.Walter@sna.co.umist.ac.uk>
  Computation Departmen – UMIST, U.K.
- Michael Wiener <michael.wiener@entrust.com>
  Entrust Technologies, Canada

The referees:

- Gordon Agnew <g.agnew@coulomb.uwaterloo.ca>
- Dan Bailey <bailey@wpi.wpi.edu>
- Thomas Blum <tblum@crcg.edu>
- Joe Buhler <jpb@msri.org>
- Wayne Burleson <burleson@ecs.umass.edu>
- Brendon Chetwynd <spunge@ece.wpi.edu>
- Craig Clapp <craigc@pictel.com>
- Jean-Sébastien Coron <coron@clipper.ens.fr>
- Adam Elbirt <aelbirt@nac.net>
- Kris Gaj <kgaj@gmu.edu>
- Jorge Guajardo <guajardo@ece.wpi.edu>
- Jim Goodman <jimg@mtl.mit.edu>

– Guang Gong <ggong@cacr.math.uwaterloo.ca>
– Dan Gordon <gordon@ccrwest.org>
– Frank Gürkaynak <kgf@WPI.EDU>
– Anwar Hasan <ahasan@claude.uwaterloo.ca>
– Julio Hernandez <julioher@dcc.unicamp.br>
– Arjen Lenstra <arjen.lenstra@citicorp.com>
– Jens-Peter Kaps <kaps@ece.WPI.EDU>
– David King <dave.king@gsc.gte.com>
– Çetin Koç <koc@ece.orst.edu>
– Peter Kornerup <kornerup@imada.sdu.dk>
– Uwe Krieger <uwe.krieger@cryptovision.com>
– Tom Messerges <Tom_Messerges-ADTL01@email.mot.com>
– Gerardo Orlando <Gerardo.Orlando@GSC.GTE.Com>
– Christof Paar <christof@ece.wpi.edu>
– Jean-Jacques Quisquater <jjq@dice.ucl.ac.be>
– Bob Rance <rrance@lucent.com>
– Perry J. Robertson <pjrober@sandia.gov>
– Francisco Rodriguez <rodrigfr@ece.orst.edu>
– Patrice L. Roussel <proussel@ichips.intel.com>
– Christoph Ruland <RULAND@nue.et-inf.uni-siegen.de>
– Eva Saar <Eva.Saar@telekom.de>
– Erkay Savaş <savas@ece.orst.edu>
– Frank Schaefer-Lorinser <F.Schaefer-Lorinser@telekom.de>
– Adi Shamir <shamir@wisdom.weizmann.ac.il>
– Joseph Silverman <jhs@ntru.com>
– Jerry Sobelman <sobelman@mail.ece.umn.edu>
– Alex Tenca <tenca@ece.orst.edu>
– Colin Walter <C.Walter@sna.co.umist.ac.uk>
– Michael Wiener <michael.wiener@entrust.com>
– D. Craig Wilcox <dcwilco@sandia.gov>
– Ed Witzke <elwitzk@sandia.gov>
– Huapeng Wu <h3wu@cacr.math.uwaterloo.ca>
– Reto Zimmermann <reto.zimmermann@gte.net>

The companies that supported the CHES workshop:

– cv cryptovision GmbH - http://www.cryptovision.com
– Intel Corporation - http://www.intel.com
– NTRU Cryptosystems, Inc. - http://www.ntru.com
– Secunet Security Networks AG - http://www.secunet.de
– SITI – Secured Information Technology, Inc.

Table of Contents

Software Implementation of Elliptic Curve Cryptography over Binary Fields

Darrel Hankerson[1]*, Julio López Hernandez[2], and Alfred Menezes[3]

[1] Dept. of Discrete and Statistical Sciences, Auburn University, USA
hankedr@mail.auburn.edu
[2] Dept. of Computer Science, University of Valle, Colombia
jlopez@borabora.univalle.edu.co
[3] Dept. of Combinatorics and Optimization, University of Waterloo, Canada
ajmeneze@uwaterloo.ca

Abstract. This paper presents an extensive and careful study of the software implementation on workstations of the NIST-recommended elliptic curves over binary fields. We also present the results of our implementation in C on a Pentium II 400 MHz workstation.

1 Introduction

Elliptic curve cryptography (ECC) was proposed independently in 1985 by Neal Koblitz [19] and Victor Miller [29]. Since then a vast amount of research has been done on its secure and efficient implementation. In recent years, ECC has received increased commercial acceptance as evidenced by its inclusion in standards by accredited standards organizations such as ANSI (American National Standards Institute) [1,2], IEEE (Institute of Electrical and Electronics Engineers) [13], ISO (International Standards Organization) [14,15], and NIST (National Institute of Standards and Technology) [33].

Before implementing an ECC system, several choices have to be made. These include selection of elliptic curve domain parameters (underlying finite field, field representation, elliptic curve), and algorithms for field arithmetic, elliptic curve arithmetic, and protocol arithmetic. The selections can be influenced by security considerations, application platform (software, firmware, or hardware), constraints of the particular computing environment (e.g., processing speed, code size (ROM), memory size (RAM), gate count, power consumption), and constraints of the particular communications environment (e.g., bandwidth, response time). Not surprisingly, it is difficult, if not impossible, to decide on a single "best" set of choices—for example, the optimal choices for a PC application can be quite different from the optimal choice for a smart card application.

Over the past 15 years, numerous papers have been written on various aspects of ECC implementation. Most of these papers do not consider all the factors involved in an efficient implementation. For example, many papers focus only on finite field arithmetic, or only on elliptic curve arithmetic.

* Supported by a grant from Auburn University COSAM.

Ç.K. Koç and C. Paar (Eds.): CHES 2000, LNCS 1965, pp. 1–24, 2000.

The contribution of this paper is an extensive and careful study of the software implementation on workstations of the NIST-recommended elliptic curves over binary fields. While the only significant constraint in workstation environments may be processing power, some of our work may also be applicable to other more constrained environments (e.g., see [4] for implementations on a pager and the Palm Pilot). We also present the results of our implementation in C (no hand-coded assembler was used) on a Pentium II 400 MHz workstation. These results serve to validate our conclusions based primarily on theoretical considerations. While some effort was made to optimize the code (e.g., loop unrolling), it is likely that significant performance improvements can be obtained especially if the code is tuned for a specific platform. Nonetheless, we hope that our work will serve as a benchmark for future efforts in this area.

The remainder of this paper is organized as follows. §2 describes the NIST curves over binary fields and presents some rationale for their selection. In §3, we describe methods for arithmetic in binary fields. §4 and §5 consider efficient techniques for elliptic curve arithmetic. In §6, we select the best methods for performing elliptic curve operations in ECC protocols such as the ECDSA. Finally, we draw our conclusions in §7 and discuss avenues for future work in §8.

2 NIST Curves over Binary Fields

In February 2000, FIPS 186-1 was revised by NIST to include the elliptic curve digital signature algorithm (ECDSA) as specified in ANSI X9.62 [1] with further recommendations for the selection of underlying finite fields and elliptic curves; the revised standard is called FIPS 186-2 [33].

FIPS 186-2 has 10 recommended finite fields: 5 prime fields, and the binary fields $\mathbb{F}_{2^{163}}$, $\mathbb{F}_{2^{233}}$, $\mathbb{F}_{2^{283}}$, $\mathbb{F}_{2^{409}}$, and $\mathbb{F}_{2^{571}}$. For each of the prime fields, one randomly selected elliptic curve was recommended, while for each of the binary fields one randomly selected elliptic curve and one Koblitz curve was selected.

The fields were selected so that the bitlengths of their orders are at least twice the key lengths of common symmetric-key block ciphers—this is because exhaustive key search of a k-bit block cipher is expected to take roughly the same time as the solution of an instance of the elliptic curve discrete logarithm problem using Pollard's rho algorithm for an appropriately-selected elliptic curve over a finite field whose order has bitlength $2k$. The correspondence between symmetric cipher key lengths and field sizes is given in Table 1. For binary fields $\mathbb{F}_{2^m}$, m was chosen so that there exists a Koblitz curve of almost prime order over $\mathbb{F}_{2^m}$. Since the order $\#E(\mathbb{F}_{2^l})$ divides $\#E(\mathbb{F}_{2^m})$ whenever l divides m, this requirement imposes the condition that m be prime.

Since the NIST binary curves are all defined over fields $\mathbb{F}_{2^m}$ where m is prime, our paper excludes from consideration fields such as $\mathbb{F}_{2^{176}}$ for which efficient techniques are known for field arithmetic [6,12]. This exclusion is not a concern in light of recent advances in algorithms for the discrete logarithm problem for elliptic curves over $\mathbb{F}_{2^m}$ when m has a small non-trivial factor [9,10].

Table 1. NIST-recommended field sizes for U.S. Federal Government use.

Symmetric cipher key length	Example algorithm	Bitlength of p in prime field $\mathbb{F}_p$	Dimension m of binary field $\mathbb{F}_{2^m}$
80	SKIPJACK	192	163
112	Triple-DES	224	233
128	AES Small [34]	256	283
192	AES Medium [34]	384	409
256	AES Large [34]	521	571

The remainder of this paper considers the efficient implementation of the NIST-recommended random and Koblitz curves over the fields $\mathbb{F}_{2^{163}}$, $\mathbb{F}_{2^{233}}$, and $\mathbb{F}_{2^{283}}$. The results can be extrapolated to curves over $\mathbb{F}_{2^{409}}$ and $\mathbb{F}_{2^{571}}$.

Description of the NIST Curves over Binary Fields. The NIST elliptic curves over $\mathbb{F}_{2^{163}}$, $\mathbb{F}_{2^{233}}$ and $\mathbb{F}_{2^{283}}$ are listed in Table 2. The following notation is used. The elements of $\mathbb{F}_{2^m}$ are represented using a polynomial basis representation with reduction polynomial $f(x)$ (see §3.1). The reduction polynomials for the fields $\mathbb{F}_{2^{163}}$, $\mathbb{F}_{2^{233}}$ and $\mathbb{F}_{2^{283}}$ are $f(x) = x^{163} + x^7 + x^6 + x^3 + 1$, $f(x) = x^{233} + x^{74} + 1$, and $f(x) = x^{283} + x^{12} + x^7 + x^5 + 1$, respectively. An elliptic curve E over $\mathbb{F}_{2^m}$ is specified by the coefficients $a, b \in \mathbb{F}_{2^m}$ of its defining equation $y^2 + xy = x^3 + ax^2 + b$. The number of points on E defined over $\mathbb{F}_{2^m}$ is nh, where n is prime, and h is called the co-factor. A random curve over $\mathbb{F}_{2^m}$ is denoted by B-m, while a Koblitz curve over $\mathbb{F}_{2^m}$ is denoted by K-m.

3 Binary Field Arithmetic

This section presents algorithms that are suitable for performing binary field arithmetic in software. For concreteness, we assume that the implementation platform has a 32-bit architecture. The bits of a word W are numbered from 0 to 31, with the rightmost bit of W designated as bit 0.

3.1 Field Representation

Of the many representations of $\mathbb{F}_{2^m}$, m prime, that have been studied, it appears that a polynomial basis representation with a trinomial or pentanomial as the reduction polynomial yields the simplest and fastest implementation in software. We will henceforth use a polynomial basis representation.

Let $f(x) = x^m + r(x)$ be an irreducible binary polynomial of degree m. The elements of $\mathbb{F}_{2^m}$ are the binary polynomials of degree at most $m-1$ with addition and multiplication performed modulo $f(x)$. A field element $a(x) = a_{m-1}x^{m-1} + \cdots + a_2x^2 + a_1x + a_0$ is associated with the binary vector $a = (a_{m-1}, \ldots, a_2, a_1, a_0)$ of length m. Let $t = \lceil m/32 \rceil$, and let $s = 32t - m$. In software, we store a in an array of t 32-bit words: $A = (A[t-1], \ldots, A[2], A[1], A[0])$, where the rightmost bit of $A[0]$ is a_0, and the leftmost s bits of $A[t-1]$ are unused (always set to 0).

Addition of field elements is performed bitwise, thus requiring only t word operations.

Table 2. NIST-recommended elliptic curves over $\mathbb{F}_{2^{163}}$, $\mathbb{F}_{2^{233}}$ and $\mathbb{F}_{2^{283}}$.

B-163: $a = 1$, $h = 2$,
$b =$ 0x 00000002 0A601907 B8C953CA 1481EB10 512F7874 4A3205FD
$n =$ 0x 00000004 00000000 00000000 000292FE 77E70C12 A4234C33
B-233: $a = 1$, $h = 2$,
$b =$ 0x 00000066 647EDE6C 332C7F8C 0923BB58 213B333B 20E9CE42 81FE115F 7D8F90AD
$n =$ 0x 00000100 00000000 00000000 00000000 0013E974 E72F8A69 22031D26 03CFE0D7
B-283: $a = 1$, $h = 2$,
$b =$ 0x 027B680A C8B8596D A5A4AF8A 19A0303F CA97FD76 45309FA2 A581485A F6263E31 3B79A2F5
$n =$ 0x 03FFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFEF90 399660FC 938A9016 5B042A7C EFADB307
K-163: $a = 1$, $b = 1$, $h = 2$,
$n =$ 0x 00000004 00000000 00000000 00020108 A2E0CC0D 99F8A5EF
K-233: $a = 0$, $b = 1$, $h = 4$,
$n =$ 0x 00000080 00000000 00000000 00000000 00069D5B B915BCD4 6EFB1AD5 F173ABDF
K-283: $a = 0$, $b = 1$, $h = 4$,
$n =$ 0x 01FFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFE9AE 2ED07577 265DFF7F 94451E06 1E163C61

3.2 Multiplication

The shift-and-add method (Algorithm 1) for field multiplication is based on the observation that $a \cdot b = a_{m-1}x^{m-1}b + \cdots + a_2x^2b + a_1xb + a_0b$. Iteration i of the algorithm computes $x^ib \bmod f(x)$ and adds the result to the accumulator c if $a_i = 1$. Note that $b \cdot x \bmod f(x)$ can be easily computed by a left-shift of the vector representation of b, followed by the addition of $r(x)$ to b if $b_m = 1$.

Algorithm 1. Right-to-left shift-and-add field multiplication

INPUT: Binary polynomials $a(x)$ and $b(x)$ of degree at most $m - 1$.
OUTPUT: $c(x) = a(x) \cdot b(x) \bmod f(x)$.

1. If $a_0 = 1$ then $c \leftarrow b$; else $c \leftarrow 0$.
2. For i from 1 to $m - 1$ do
   2.1 $b \leftarrow b \cdot x \bmod f(x)$.
   2.2 If $a_i = 1$ then $c \leftarrow c + b$.
3. Return(c).

While Algorithm 1 is well-suited for hardware where a vector shift can be performed in one clock cycle, the large number of word shifts make it less desirable for software implementation. We next consider faster methods for field multiplication which first multiply the field elements as polynomials, and then reduce the result modulo $f(x)$.

Polynomial Multiplication. The comb method for polynomial multiplication is based on the observation that if $b(x) \cdot x^k$ has been computed for some $k \in [0, 31]$, then $b(x) \cdot x^{32j+k}$ can be easily obtained by appending j zero words to the right of the vector representation of $b(x) \cdot x^k$. Algorithm 2 considers the bits of the words of A from right to left, while Algorithm 3 considers the bits from left to right. The following notation is used: if $C = (C[n], \ldots, C[2], C[1], C[0])$ is a vector, then $C\{j\}$ denotes the truncated vector $(C[n], \ldots, C[j+1], C[j])$.

Algorithm 2. Right-to-left comb method for polynomial multiplication

INPUT: Binary polynomials $a(x)$ and $b(x)$ of degree at most $m-1$.
OUTPUT: $c(x) = a(x) \cdot b(x)$.

1. $C \leftarrow 0$.
2. For k from 0 to 31 do
   2.1 For j from 0 to $t-1$ do
   If the kth bit of $A[j]$ is 1 then add B to $C\{j\}$.
   2.2 If $k \neq 31$ then $B \leftarrow B \cdot x$.
3. Return(C).

Algorithm 3. Left-to-right comb method for polynomial multiplication

INPUT: Binary polynomials $a(x)$ and $b(x)$ of degree at most $m-1$.
OUTPUT: $c(x) = a(x) \cdot b(x)$.

1. $C \leftarrow 0$.
2. For k from 31 downto 0 do
   2.1 For j from 0 to $t-1$ do
   If the kth bit of $A[j]$ is 1 then add B to $C\{j\}$.
   2.2 If $k \neq 0$ then $C \leftarrow C \cdot x$.
3. Return(C).

Algorithms 2 and 3 are both faster than Algorithm 1 since there are fewer vector shifts (multiplications by x). Algorithm 2 is faster than Algorithm 3 since the vector shifts in the former involve the t-word vector B, while the vector shifts in the latter involve the $2t$-word vector C. In [27] it was observed that Algorithm 3 can be sped up considerably at the expense of some storage overhead by precomputing $u(x) \cdot b(x)$ for all polynomials $u(x)$ of degree less than w, where w divides the word length, and considering the bits of the $A[j]$'s w at a time. The modified method with $w = 4$ is presented as Algorithm 4.

Algorithm 4. Left-to-right comb method with windows of width $w = 4$

INPUT: Binary polynomials $a(x)$ and $b(x)$ of degree at most $m-1$.
OUTPUT: $c(x) = a(x) \cdot b(x)$.

1. Compute $B_u = u(x) \cdot b(x)$ for all polynomials $u(x)$ of degree at most 3.
2. $C \leftarrow 0$.
3. For k from 7 downto 0 do
   3.1 For j from 0 to $t-1$ do
   Let $u = (u_3, u_2, u_1, u_0)$, where u_i is bit $(4k+i)$ of $A[j]$. Add B_u to $C\{j\}$.
   3.2 If $k \neq 0$ then $C \leftarrow C \cdot x^4$.
4. Return(C).

The last method we consider for polynomial multiplication was first described by Karatsuba for multiplying integers (see [18]). Suppose that m is even. To multiply two binary polynomials $a(x)$ and $b(x)$ of degree at most $m-1$, we first split up $a(x)$ and $b(x)$ each into two polynomials of degree at most $(m/2)-1$: $a(x) = A_1(x)X + A_0(x)$, $b(x) = B_1(x)X + B_0(x)$, where $X = x^{m/2}$. Then

$$a(x)b(x) = A_1B_1X^2 + [(A_1 + A_0)(B_1 + B_0) + A_1B_1 + A_0B_0]X + A_0B_0,$$

which can be derived from three products of polynomials of degree $(m/2)-1$. These products in turn can be computed recursively. For the case $m = 163$, we first prepended a 0 bit to the field elements a and b so that their bitlength is 164, and then used Karatsuba's method to subdivide the multiplication of a and b into multiplications of polynomials of degree at most 40. The latter multiplications were performed using a variant of Algorithm 4. For the case $m = 233$ (resp. $m = 283$), we first prepended twenty-three (five) 0 bits to a and b, and then used Karatsuba's method to subdivide the multiplication of a and b into multiplications of polynomials of degree at most 63 (71).

Reduction. Let $c(x)$ be a binary polynomial of degree at most $2m-2$. Algorithm 5 reduces $c(x)$ modulo $f(x)$ one bit at a time, starting with the leftmost bit. It is based on the observation that $x^i \equiv x^{i-m}r(x) \pmod{f(x)}$ for $i \geq m$. The polynomials $x^k r(x)$, $0 \leq k \leq 31$, can be precomputed. If $r(x)$ is a low-degree polynomial, or if $f(x)$ is a trinomial, then the space requirements are smaller, and also the additions involving $x^k r(x)$ are faster.

Algorithm 5. Modular reduction (one bit at a time)

INPUT: A binary polynomial $c(x)$ of degree at most $2m-2$.
OUTPUT: $c(x) \bmod f(x)$.

1. *Precomputation.* Compute $u_k(x) = x^k r(x)$, $0 \leq k \leq 31$.
2. For i from $2m-2$ downto m do
   2.1 If $c_i = 1$ then
   Let $j = \lfloor (i-m)/32 \rfloor$ and $k = (i-m) - 32j$.
   Add $u_k(x)$ to $C\{j\}$.
3. Return$((C[t-1], \ldots, C[1], C[0]))$.

If $f(x)$ is a trinomial, or a pentanomial with middle terms close to each other, then reduction of $c(x)$ modulo $f(x)$ can be efficiently performed one word at a time. For example, consider reducing the ninth word $C[9]$ of $c(x)$ modulo $f(x) = x^{163} + x^7 + x^6 + x^3 + 1$. Here, $m = 163$ and $t = 6$. We have

$$\begin{aligned}
x^{288} &\equiv x^{132} + x^{131} + x^{128} + x^{125} \pmod{f(x)} \\
x^{289} &\equiv x^{133} + x^{132} + x^{129} + x^{126} \pmod{f(x)} \\
&\vdots \\
x^{319} &\equiv x^{163} + x^{162} + x^{159} + x^{156} \pmod{f(x)}.
\end{aligned}$$

By considering columns on the right side of the above congruences, it follows that reduction of $C[9]$ can be performed by adding $C[9]$ four times to C, with

the rightmost bit of $C[9]$ added to bits 132, 131, 128 and 125 of C. This leads to Algorithm 6 for modular reduction which can be easily extended to other reduction polynomials. For the reduction polynomials considered in this paper, Algorithm 6 is faster than Algorithm 5 and furthermore has no storage overhead.

Algorithm 6. Modular reduction (one word at a time)

INPUT: A binary polynomial $c(x)$ of degree at most 324.
OUTPUT: $c(x)$ mod $f(x)$, where $f(x) = x^{163} + x^7 + x^6 + x^3 + 1$.
1. For i from 10 downto 6 do {Reduce $C[i]$ modulo $f(x)$}
   1.1 $T \leftarrow C[i]$.
   1.2 $C[i-6] \leftarrow C[i-6] \oplus (T \ll 29)$.
   1.3 $C[i-5] \leftarrow C[i-5] \oplus (T \ll 4) \oplus (T \ll 3) \oplus T \oplus (T \gg 3)$.
   1.4 $C[i-4] \leftarrow C[i-4] \oplus (T \gg 28) \oplus (T \gg 29)$.
2. $T \leftarrow C[5]$ AND 0xFFFFFFF8. {Clear bits 0, 1 and 2 of $C[5]$}
3. $C[0] \leftarrow C[0] \oplus (T \ll 4) \oplus (T \ll 3) \oplus T \oplus (T \gg 3)$.
4. $C[1] \leftarrow C[1] \oplus (T \gg 28) \oplus (T \gg 29)$.
5. $C[5] \leftarrow C[5]$ AND 0x00000007. {Clear the unused bits of $C[5]$}
6. Return$((C[5], C[4], C[3], C[2], C[1], C[0]))$.

3.3 Squaring

Squaring a polynomial is much faster than multiplying two arbitrary polynomials since squaring is a linear operation in $\mathbb{F}_{2^m}$; that is, if $a(x) = \sum_{i=0}^{m-1} a_i x^i$, then $a(x)^2 = \sum_{i=0}^{m-1} a_i x^{2i}$. The binary representation of $a(x)^2$ is obtained by inserting a 0 bit between consecutive bits of the binary representation of $a(x)$. To facilitate this process, a table of size 512 bytes can be precomputed for converting 8-bit polynomials into their expanded 16-bit counterparts [37].

Algorithm 7. Squaring

INPUT: $a \in \mathbb{F}_{2^m}$.
OUTPUT: a^2 mod $f(x)$.
1. *Precomputation.* For each byte $v = (v_7, \ldots, v_1, v_0)$, compute the 16-bit quantity $T(v) = (0, v_7, \ldots, 0, v_1, 0, v_0)$.
2. For i from 0 to $t-1$ do
   2.1 Let $A[i] = (u_3, u_2, u_1, u_0)$ where each u_j is a byte.
   2.2 $C[2i] \leftarrow (T(u_1), T(u_0))$, $C[2i+1] \leftarrow (T(u_3), T(u_2))$.
3. Compute $b(x) = c(x)$ mod $f(x)$.
4. Return(b).

3.4 Inversion

Algorithm 8 computes the inverse of a non-zero field element $a \in \mathbb{F}_{2^m}$ using a variant of the Extended Euclidean Algorithm (EEA) for polynomials. The algorithm maintains the invariants $ba + df = u$ and $ca + ef = v$ for some d and e which are not explicitly computed. At each iteration, if $\deg(u) \geq \deg(v)$, then a partial division of u by v is performed by subtracting $x^j v$ from u, where $j = \deg(u) - \deg(v)$. In this way the degree of u is decreased by at least 1, and on average by 2. Subtracting $x^j c$ from b preserves the invariants. The algorithm terminates when $\deg(u) = 0$, in which case $u = 1$ and $ba + df = 1$; hence $b = a^{-1}$ mod $f(x)$.

Algorithm 8. Extended Euclidean Algorithm for inversion in $\mathbb{F}_{2^m}$

INPUT: $a \in \mathbb{F}_{2^m}$, $a \neq 0$.
OUTPUT: $a^{-1} \bmod f(x)$.

1. $b \leftarrow 1$, $c \leftarrow 0$, $u \leftarrow a$, $v \leftarrow f$.
2. While $\deg(u) \neq 0$ do
   2.1 $j \leftarrow \deg(u) - \deg(v)$.
   2.2 If $j < 0$ then: $u \leftrightarrow v$, $b \leftrightarrow c$, $j \leftarrow -j$.
   2.3 $u \leftarrow u + x^j v$, $b \leftarrow b + x^j c$.
3. Return(b).

The Almost Inverse Algorithm (AIA, Algorithm 9) is from [37]. For $a \in \mathbb{F}_{2^m}$, $a \neq 0$, a pair (b, k) is returned where $ba \equiv x^k \pmod{f(x)}$. A reduction is then applied to obtain $a^{-1} = bx^{-k} \bmod f(x)$. The invariants are $ba + df = ux^k$ and $ca + ef = vx^k$ for some d and e which are not explicitly calculated. After step 2, both u and v have a constant term of 1; after step 5, u is divisible by x and hence the degree of u is always reduced at each iteration. The value of k is incremented in step 2.1 to preserve the invariants. The algorithm terminates when $u = 1$, giving $ba + df = x^k$. While EEA eliminates bits of u and v from left to right (high degree to low degree), AIA eliminates bits from right to left. In addition, in AIA some bits are also lost on the left in the case $\deg(u) = \deg(v)$ before step 5. Consequently, AIA is expected to take fewer iterations than EEA.

The reduction step can be performed as follows. Let $s = \min\{i \geq 1 \mid f_i = 1\}$, where $f(x) = f_m x^m + \cdots + f_1 x + f_0$. Let b' be the polynomial formed by the s rightmost bits of b. Then $b'f + b$ is divisible by x^s and $b'' = (b'f + b)/x^s$ has degree less than m; thus $b'' = bx^{-s} \bmod f(x)$. This process can be repeated to finally obtain $bx^{-k} \bmod f(x)$. The reduction polynomial is said to be *suitable* if $s \geq 32$, since then fewer iterations are required in the reduction step.

Algorithm 9. Almost Inverse Algorithm for inversion in $\mathbb{F}_{2^m}$

INPUT: $a \in \mathbb{F}_{2^m}$, $a \neq 0$.
OUTPUT: $b \in \mathbb{F}_{2^m}$ and $k \in [0, 2m-1]$ such that $ba \equiv x^k \pmod{f(x)}$.

1. $b \leftarrow 1$, $c \leftarrow 0$, $u \leftarrow a$, $v \leftarrow f$, $k \leftarrow 0$.
2. While x divides u do:
   2.1 $u \leftarrow u/x$, $c \leftarrow cx$, $k \leftarrow k + 1$.
3. If $u = 1$ then return(b,k).
4. If $\deg(u) < \deg(v)$ then: $u \leftrightarrow v$, $b \leftrightarrow c$.
5. $u \leftarrow u + v$, $b \leftarrow b + c$.
6. Goto step 2.

Algorithm 10 is a modification of Algorithm 9, producing the inverse directly. Rather than maintaining the integer k, the algorithm performs a division of b whenever u is divided by x. Note that if b is not divisible by x, then b is replaced by $b + f$ (and d by $d - a$) in step 2.2 before the division. On termination, $ba + df = 1$, whence $b = a^{-1} \bmod f(x)$.

Algorithm 10. Modified Almost Inverse Algorithm for inversion in $\mathbb{F}_{2^m}$

INPUT: $a \in \mathbb{F}_{2^m}$, $a \neq 0$.
OUTPUT: $a^{-1} \bmod f(x)$.

1. $b \leftarrow 1$, $c \leftarrow 0$, $u \leftarrow a$, $v \leftarrow f$.
2. While x divides u do:
   2.1 $u \leftarrow u/x$.
   2.2 If x divides b then $b \leftarrow b/x$; else $b \leftarrow (b+f)/x$.
3. If $u = 1$ then return(b).
4. If $\deg(u) < \deg(v)$ then: $u \leftrightarrow v$, $b \leftrightarrow c$.
5. $u \leftarrow u + v$, $b \leftarrow b + c$.
6. Goto step 2.

Step 2 of AIA is simpler than that in MAIA. In addition, the b and c appearing in these algorithms grow more slowly in AIA. Thus one can expect AIA to outperform MAIA if the reduction polynomial is suitable, and conversely.

3.5 Timings

Table 3 presents timing results for operations in the fields $\mathbb{F}_{2^{163}}$, $\mathbb{F}_{2^{233}}$ and $\mathbb{F}_{2^{283}}$. The field arithmetic was implemented in C and the timings obtained on a Pentium II 400 MHz workstation.

Table 3. Timings (in μs) for operations in $\mathbb{F}_{2^{163}}$, $\mathbb{F}_{2^{233}}$ and $\mathbb{F}_{2^{283}}$. The reduction polynomials are, respectively, $f(x) = x^{163}+x^7+x^6+x^3+1$, $f(x) = x^{233}+x^{74}+1$, and $f(x) = x^{283} + x^{12} + x^7 + x^5 + 1$.

	$m = 163$	$m = 233$	$m = 283$
Addition	0.10	0.12	0.13
Modular reduction (Algorithm 6)	0.18	0.22	0.35
Multiplication (including reduction)			
Shift-and-add (Algorithm 1)	16.36	27.14	37.95
Right-to-left comb (Algorithm 2)	6.87	12.01	14.74
Left-to-right comb (Algorithm 3)	8.40	12.93	15.81
LR comb with windows of size 4 (Algorithm 4)	3.00	5.07	6.23
Karatsuba	3.92	7.04	8.01
Squaring (Algorithm 7)	0.40	0.55	0.75
Inversion			
Extended Euclidean Algorithm (Algorithm 8)	30.99	53.22	70.32
Almost Inverse Algorithm (Algorithm 9)	42.49	68.63	104.28
Modified Almost Inverse Algorithm (Algorithm 10)	40.26	73.05	96.49

As expected, addition, modular reduction, and squaring are relatively inexpensive compared to multiplication and inversion. The left-to-right comb method with windows of size 4 is the fastest multiplication algorithm, however it requires a modest amount of extra storage (e.g., 336 bytes for 14 polynomials in the case

$m = 163$). Our implementation of Karatsuba's algorithm is competitive and requires a similar amount of storage since the base multiplications were performed using the left-to-right comb method with windows of size 4.

We found the Extended Euclidean Algorithm to be faster than the Almost Inverse Algorithm and the Modified Almost Inverse Algorithm, contrary to the findings of [37] and [7]. This discrepancy is partially explained by the unsuitable form of the reduction polynomial for $m = 163$ and $m = 283$ (see [7]). Also, we found that AIA and MAIA were more difficult to optimize than EEA without resorting to hand-coded assembler. In any case, the ratio of the fastest inversion method to the fastest multiplication method was found to be roughly 10 to 1, again contrary to the roughly 3 to 1 ratio reported in [37], [6] and [7]. This discrepancy could be attributed to a considerably faster implementation of multiplication in our work. As a result, we chose to represent elliptic curve points in projective coordinates instead of affine coordinates as was done in [37] and [7] (see §4).

4 Elliptic Curve Point Representation

Affine Coordinates. Let E be an elliptic curve over $\mathbb{F}_{2^m}$ given by the (affine) equation $y^2 + xy = x^3 + ax^2 + b$, where $a \in \{0, 1\}$. Let $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ be two points on E with $P_1 \neq -P_2$. Then the coordinates of $P_3 = P_1 + P_2 = (x_3, y_3)$ can be computed as follows:

$$x_3 = \lambda^2 + \lambda + x_1 + x_2 + a, \quad y_3 = (x_1 + x_3)\lambda + x_3 + y_1, \text{ where}$$
$$\lambda = \frac{y_1 + y_2}{x_1 + x_2} \text{ if } P_1 \neq P_2, \quad \text{and } \lambda = \frac{y_1}{x_1} + x_1 \text{ if } P_1 = P_2. \tag{1}$$

In either case, when $P_1 \neq P_2$ (general addition) and $P_1 = P_2$ (doubling), the formulas for computing P_3 require 1 field inversion and 2 field multiplications—as justified in §3.5, we can ignore the cost of field additions and squarings.

Projective Coordinates. In situations where inversion in $\mathbb{F}_{2^m}$ is expensive relative to multiplication, it may be advantageous to represent points using projective coordinates of which several types have been proposed. In *standard* projective coordinates, the projective point $(X : Y : Z)$, $Z \neq 0$, corresponds to the affine point $(X/Z, Y/Z)$. The projective equation of the elliptic curve is $Y^2Z + XYZ = X^3 + aX^2Z + bZ^3$. In *Jacobian* projective coordinates [5], the projective point $(X : Y : Z)$, $Z \neq 0$, corresponds to the affine point $(X/Z^2, Y/Z^3)$ and the projective equation of the curve is $Y^2 + XYZ = X^3 + aX^2Z^2 + bZ^6$. In [25], a new set of projective coordinates was introduced. Here, a projective point $(X : Y : Z)$, $Z \neq 0$, corresponds to the affine point $(X/Z, Y/Z^2)$, and the projective equation of the curve is

$$Y^2 + XYZ = X^3Z + aX^2Z^2 + bZ^4. \tag{2}$$

Formulas which do not require inversions for adding and doubling points in projective coordinates can be derived by first converting the points to affine

coordinates, then using the formulas (1) to add the affine points, and finally clearing denominators. Also of use in left-to-right point multiplication methods (see §5.1) is the addition of two points using mixed coordinates—one point given in affine coordinates and the other in projective coordinates. Doubling formulas for the projective equation (2) are: $2(X_1 : Y_1 : Z_1) = (X_3 : Y_3 : Z_3)$, where

$$Z_3 = X_1^2 \cdot Z_1^2, \;\; X_3 = X_1^4 + b \cdot Z_1^4, \;\; Y_3 = bZ_1^4 \cdot Z_3 + X_3 \cdot (aZ_3 + Y_1^2 + bZ_1^4). \quad (3)$$

Formulas for addition in mixed coordinates are: $(X_1 : Y_1 : Z_1) + (X_2 : Y_2 : 1) = (X_3 : Y_3 : Z_3)$, where

$$\begin{gathered} A = Y_2 \cdot Z_1^2 + Y_1, \;\; B = X_2 \cdot Z_1 + X_1, \;\; C = Z_1 \cdot B, \;\; D = B^2 \cdot (C + aZ_1^2), \\ Z_3 = C^2, \;\; E = A \cdot C, \;\; X_3 = A^2 + D + E, \;\; F = X_3 + X_2 \cdot Z_3, \\ G = X_3 + Y_2 \cdot Z_3, \;\; Y_3 = E \cdot F + Z_3 \cdot G. \end{gathered} \quad (4)$$

The field operation counts for point addition and doubling in the various coordinate systems are listed in Table 4. Since our implementation of inversion is at least 10 times as expensive as multiplication (see §3.5), unless otherwise stated, all our elliptic curve operations will use projective coordinates.

Table 4. Operation counts for point addition and doubling.

Coordinate system	General addition	General addition (mixed coordinates)	Doubling
Affine	$1I$, $2M$	—	$1I$, $2M$
Standard projective $(X/Z, Y/Z)$	$13M$	$12M$	$7M$
Jacobian projective $(X/Z^2, Y/Z^3)$	$14M$	$10M$	$5M$
Projective $(X/Z, Y/Z^2)$	$14M$	$9M$	$4M$

5 Point Multiplication

This section considers methods for computing kP, where k is an integer and P is an elliptic curve point. This operation is called *point multiplication* or *scalar multiplication*, and dominates the execution time of elliptic curve cryptographic schemes. We will assume that $\#E(\mathbb{F}_{2^m}) = nh$ where n is prime and h is small (so $n \approx 2^m$), P has order n, and $k \in_R [1, n-1]$. In §5.1 we consider techniques which do not exploit any special structure of the curve. In §5.2 we study techniques for Koblitz curves which use the Frobenius endomorphism. In both cases, one can take advantage of the situation where P is a fixed point (e.g., the base point in elliptic curve domain parameters) by precomputing some data which depends only on P. For surveys of exponentiation methods, see [11] and [28].

5.1 Random Curves

Algorithm 11 is the additive version of the basic repeated-square-and-multiply method for exponentiation.

Algorithm 11. (Left-to-right) binary method for point multiplication

INPUT: $k = (k_{t-1}, \ldots, k_1, k_0)_2$, $P \in E(\mathbb{F}_{2^m})$.
OUTPUT: kP.

1. $Q \leftarrow \mathcal{O}$.
2. For i from $t-1$ downto 0 do
   2.1 $Q \leftarrow 2Q$.
   2.2 If $k_i = 1$ then $Q \leftarrow Q + P$.
3. Return(Q).

The expected number of ones in the binary representation of k is $t/2 \approx m/2$, whence the expected running time of Algorithm 11 is approximately $m/2$ point additions and m point doublings, denoted $0.5mA + mD$. If affine coordinates (see §4) are used, then the running time expressed in terms of field operations is $3mM + 1.5mI$, where I denotes an inversion and M a field multiplication. If projective coordinates (see §4) are used, then Q is stored in projective coordinates, while P can be stored in affine coordinates. Thus the doubling in step 2.1 can be performed using (3), and the addition in step 2.2 can be performed using (4). The field operation count of Algorithm 11 is then $8.5mM + (2M + 1I)$ (1 inversion and 2 multiplications are required to convert back to affine coordinates).

If $P = (x, y) \in E(\mathbb{F}_{2^m})$ then $-P = (x, x + y)$. Thus subtraction of points on an elliptic curve over a binary field is just as efficient as addition. This motivates using a *signed digit representation* $k = \sum_{i=0}^{l-1} k_i 2^i$, where $k_i \in \{0, \pm 1\}$. A particularly useful signed digit representation is the *non-adjacent form* (NAF) which has the property that no two consecutive coefficients k_i are nonzero. Every positive integer k has a unique NAF, denoted NAF(k). Moreover, NAF(k) has the fewest non-zero coefficients of any signed digit representation of k. NAF(k) can be efficiently computed using Algorithm 12 [38].

Algorithm 12. Computing the NAF of a positive integer

INPUT: A positive integer k.
OUTPUT: NAF(k).

1. $i \leftarrow 0$.
2. While $k \geq 1$ do
   2.1 If k is odd then: $k_i \leftarrow 2 - (k \bmod 4)$, $k \leftarrow k - k_i$;
   2.2 Else: $k_i \leftarrow 0$.
   2.3 $k \leftarrow k/2$, $i \leftarrow i + 1$.
3. Return($(k_{i-1}, k_{i-2}, \ldots, k_1, k_0)$).

Algorithm 13 modifies Algorithm 11 by using NAF(k) instead of the binary representation of k. It is known that the length of NAF(k) is at most one longer than the binary representation of k. Also, the average density of non-zero coefficients among all NAFs of length l is approximately 1/3 [32]. It follows that the expected running time of Algorithm 13 is approximately $(m/3)A + mD$.

Algorithm 13. Binary NAF method for point multiplication

INPUT: $\text{NAF}(k) = \sum_{i=0}^{l-1} k_i 2^i$, $P \in E(\mathbb{F}_{2^m})$.
OUTPUT: kP.

1. $Q \leftarrow \mathcal{O}$.
2. For i from $l-1$ downto 0 do
   2.1 $Q \leftarrow 2Q$.
   2.2 If $k_i = 1$ then $Q \leftarrow Q + P$.
   2.3 If $k_i = -1$ then $Q \leftarrow Q - P$.
3. Return(Q).

If some extra memory is available, the running time of Algorithm 13 can be decreased by using a window method which processes w digits of k at a time. One approach we did not implement is to first compute $\text{NAF}(k)$ or some other signed digit representation of k (e.g., [23] or [30]), and then process the digits using a sliding window of width w. Algorithm 14 from [38], described next, is another window method.

A *width-w NAF* of an integer k is an expression $k = \sum_{i=0}^{l-1} k_i 2^i$, where each non-zero coefficient k_i is odd, $|k_i| < 2^{w-1}$, and at most one of any w consecutive coefficients is nonzero. Every positive integer has a unique width-w NAF, denoted $\text{NAF}_w(k)$. Note that $\text{NAF}_2(k) = \text{NAF}(k)$. $\text{NAF}_w(k)$ can be efficiently computed using Algorithm 12 modified as follows: in step 2.1 replace "$k_i \leftarrow 2-(k \bmod 4)$" by "$k_i \leftarrow k \bmod 2^w$", where $k \bmod 2^w$ denotes the integer u satisfying $u \equiv k \pmod{2^w}$ and $-2^{w-1} \leq u < 2^{w-1}$. It is known that the length of $\text{NAF}_w(k)$ is at most one longer than the binary representation of k. Also, the average density of non-zero coefficients among all width-w NAFs of length l is approximately $1/(w+1)$ [38]. It follows that the expected running time of Algorithm 14 is approximately $(1D + (2^{w-2}-1)A) + (m/(w+1)A + mD)$. When using projective coordinates, the running time in the case $m = 163$ is minimized when $w = 4$. For the cases $m = 233$ and $m = 283$, the minimum is attained when $w = 5$; however, since the running times are only slightly greater when $w = 4$, we selected $w = 4$ for our implementation.

Algorithm 14. Window NAF method for point multiplication

INPUT: Window width w, $\text{NAF}_w(k) = \sum_{i=0}^{l-1} k_i 2^i$, $P \in E(\mathbb{F}_{2^m})$.
OUTPUT: kP.

1. Compute $P_i = iP$, for $i \in \{1, 3, 5, \ldots, 2^{w-1} - 1\}$.
2. $Q \leftarrow \mathcal{O}$.
3. For i from $l-1$ downto 0 do
   3.1 $Q \leftarrow 2Q$.
   3.2 If $k_i \neq 0$ then:
   If $k_i > 0$ then $Q \leftarrow Q + P_{k_i}$;
   Else $Q \leftarrow Q - P_{k_i}$.
4. Return(Q).

Algorithm 15 is from [26] and is based on an idea of Montgomery [31]. Let $Q_1 = (x_1, y_1)$, $Q_2 = (x_2, y_2)$ with $Q_1 \neq \pm Q_2$. Let $Q_1 + Q_2 = (x_3, y_3)$ and $Q_1 - Q_2 = (x_4, y_4)$. Then using the addition formulas (1), it can be verified that

$$x_3 = x_4 + \frac{x_1}{x_1 + x_2} + \left(\frac{x_1}{x_1 + x_2}\right)^2. \tag{5}$$

Thus, the x-coordinate of $Q_1 + Q_2$ can be computed from the x-coordinates of Q_1, Q_2 and $Q_1 - Q_2$. Iteration j of Algorithm 15 for determining kP computes $T_j = (lP, (l+1)P)$, where l is the integer given by the j leftmost bits of k. Then $T_{j+1} = (2lP, (2l+1)P)$ or $((2l+1)P, (2l+2)P)$ if the $(j+1)$st leftmost bit of k is 0 or 1, respectively. Each iteration requires one doubling and one addition using (5). After the last iteration, having computed the x-coordinates of $kP = (x_1, y_1)$ and $(k+1)P = (x_2, y_2)$, the y-coordinate of kP can be recovered as:

$$y_1 = x^{-1}(x_1 + x)[(x_1 + x)(x_2 + x) + x^2 + y] + y. \tag{6}$$

Equation (6) is derived using the addition formula (1) for computing the x-coordinate x_2 of $(k+1)P$ from $kP = (x_1, y_1)$ and $P = (x, y)$. Algorithm 15 is presented using standard projective coordinates (see §4). The approximate running time is $6mM + (1I + 10M)$. One advantage of Algorithm 15 is that it does not have any extra storage requirements.

Algorithm 15. Montgomery point multiplication

INPUT: $k = (k_{t-1}, \ldots, k_1, k_0)_2$ with $k_{t-1} = 1$, $P = (x, y) \in E(\mathbb{F}_{2^m})$.
OUTPUT: kP.

1. $X_1 \leftarrow x$, $Z_1 \leftarrow 1$, $X_2 \leftarrow x^4 + b$, $Z_2 \leftarrow x^2$. {Compute $(P, 2P)$}
2. For i from $t-2$ downto 0 do
   2.1 If $k_i = 1$ then
   $T \leftarrow Z_1$, $Z_1 \leftarrow (X_1 Z_2 + X_2 Z_1)^2$, $X_1 \leftarrow x Z_1 + X_1 X_2 T Z_2$.
   $T \leftarrow X_2$, $X_2 \leftarrow X_2^4 + b Z_2^4$, $Z_2 \leftarrow T^2 Z_2^2$.
   2.2 Else
   $T \leftarrow Z_2$, $Z_2 \leftarrow (X_1 Z_2 + X_2 Z_1)^2$, $X_2 \leftarrow x Z_2 + X_1 X_2 Z_1 T$.
   $T \leftarrow X_1$, $X_1 \leftarrow X_1^4 + b Z_1^4$, $Z_1 \leftarrow T^2 Z_1^2$.
3. $x_3 \leftarrow X_1 / Z_1$.
4. $y_3 \leftarrow (x + X_1/Z_1)[(X_1 + xZ_1)(X_2 + xZ_2) + (x^2 + y)(Z_1 Z_2)](xZ_1 Z_2)^{-1} + y$.
5. Return$((x_3, y_3))$.

If the point P is fixed and some storage is available, then point multiplication can be sped up by precomputing some data which depends only on P. For example, if the points $2P, 2^2P, \ldots, 2^{t-1}P$ are precomputed, then the right-to-left binary method has expected running time $(m/2)A$ (all doublings are eliminated). In [3], a refinement of this idea was proposed. Let $(k_{d-1}, \ldots, k_1, k_0)_{2^w}$ be the 2^w-ary representation of k, where $d = \lceil t/w \rceil$, and let $Q_j = \sum_{i:k_i=j} 2^{wi} P$. Then

$$\begin{aligned} kP &= \sum_{i=0}^{d-1} k_i (2^{wi} P) = \sum_{j=1}^{2^w-1} \left(j \sum_{i:k_i=j} 2^{wi} P \right) = \sum_{j=1}^{2^w-1} j Q_j \\ &= Q_{2^w-1} + (Q_{2^w-1} + Q_{2^w-2}) + \cdots + (Q_{2^w-1} + Q_{2^w-2} + \cdots + Q_1). \end{aligned} \tag{7}$$

Algorithm 16 is based on this observation. Its expected running time is approximately $((d(2^w-1)/2^w-1)+(2^w-2))A$. Note that if projective coordinates are used, then only the additions in step 3.1 are in mixed coordinates.

Algorithm 16. Fixed-base windowing method

INPUT: Window width w, $d=\lceil t/w \rceil$, $k=(k_{d-1},\ldots,k_1,k_0)_{2^w}$, $P \in E(\mathbb{F}_{2^m})$.
OUTPUT: kP.

1. *Precomputation.* Compute $P_i = 2^{wi}P$, $0 \le i \le d-1$.
2. $A \leftarrow \mathcal{O}$, $B \leftarrow \mathcal{O}$.
3. For j from 2^w-1 downto 1 do
   3.1 For each i for which $k_i = j$ do: $B \leftarrow B + P_i$. {Add Q_j to B}
   3.2 $A \leftarrow A + B$.
4. Return(A).

In the comb method, proposed in [24], the binary representation of k is written in w rows, and the columns of the resulting rectangle are processed one column at a time. We define $[a_{w-1},\ldots,a_2,a_1,a_0]P = a_{w-1}2^{(w-1)d}P + \cdots + a_2 2^{2d}P + a_1 2^d P + a_0 P$, where $d=\lceil t/w \rceil$ and $a_i \in \mathbb{Z}_2$. The expected running time of Algorithm 17 is $((d-1)(2^w-1)/2^w)A + (d-1)D$.

Algorithm 17. Fixed-base comb method

INPUT: Window width w, $d=\lceil t/w \rceil$, $k=(k_{t-1},\ldots,k_1,k_0)_2$, $P \in E(\mathbb{F}_{2^m})$.
OUTPUT: kP.

1. *Precomputation.* Compute $[a_{w-1},\ldots,a_1,a_0]P \ \forall (a_{w-1},\ldots,a_1,a_0) \in \mathbb{Z}_2^w$.
2. By padding k on the left with 0's if necessary, write $k = K^{w-1}\|\cdots\|K^1\|K^0$, where each K^j is a bit string of length d. Let K_i^j denote the ith bit of K^j.
3. $Q \leftarrow \mathcal{O}$.
4. For i from $d-1$ downto 0 do
   4.1 $Q \leftarrow 2Q$.
   4.2 $Q \leftarrow Q + [K_i^{w-1},\ldots,K_i^1,K_i^0]P$.
5. Return(Q).

From Table 5 we see that the fixed-base comb method is expected to outperform the fixed-base window method for similar amounts of storage. For our implementation, we chose $w=4$ for the fixed-base comb method.

Table 5. Comparison of fixed-base window and fixed-base comb methods. w is the window width, S denotes the number of points stored in the precomputation phase, and T denotes the number of field operations. Affine coordinates were used for fixed-base window, and projective coordinates were used for fixed-base comb.

	$w=2$		$w=3$		$w=4$		$w=5$		$w=6$		$w=7$		$w=8$	
Method	S	T	S	T	S	T	S	T	S	T	S	T	S	T
Fixed-base window	81	756	54	648	40	624	32	732	27	1068	23	1788	20	3288
Fixed-base comb	2	885	6	660	14	514	30	419	62	363	126	311	254	272

5.2 Koblitz Curves

Koblitz curves are elliptic curves defined over $\mathbb{F}_2$, and were first proposed for cryptographic use in [20]. The primary advantage of Koblitz curves is that point multiplication algorithms can be devised that do not use any point doublings. All the algorithms and facts stated in this section are due to Solinas [38].

There are two Koblitz curves: $E_0 : y^2 + xy = x^3 + 1$ and $E_1 : y^2 + xy = x^3 + x^2 + 1$. Let $\mu = (-1)^{1-a}$. We have $\#E_a(\mathbb{F}_2) = 3 - \mu$. We assume that $\#E_a(\mathbb{F}_{2^m})$ is almost prime, i.e., $\#E_a(\mathbb{F}_{2^m}) = hn$, where n is prime and $h = 3-\mu$. The number of points is given by $\#E_a(\mathbb{F}_{2^m}) = 2^m + 1 - V_m$, where $\{V_k\}$ is the Lucas sequence defined by $V_0 = 2$, $V_1 = \mu$, $V_{k+1} = \mu V_k - 2V_{k-1}$ for $k \geq 1$.

Since E_a is defined over $\mathbb{F}_{2^m}$, the *Frobenius map* $\tau : E_a(\mathbb{F}_{2^m}) \rightarrow E_a(\mathbb{F}_{2^m})$ defined by $\tau(\mathcal{O}) = \mathcal{O}$, $\tau((x, y)) = (x^2, y^2)$ is well-defined. Moreover, it can be efficiently computed since squaring in $\mathbb{F}_{2^m}$ is relatively inexpensive (see §3.5). It is known that $(\tau^2 + 2)P = \mu\tau P$ for all $P \in E_a(\mathbb{F}_{2^m})$. Hence the Frobenius map can be regarded as the complex number τ satisfying $\tau^2 + 2 = \mu\tau$, i.e., $\tau = (\mu + \sqrt{-7})/2$. It now makes sense to multiply points in $E_a(\mathbb{F}_{2^m})$ by elements of the ring $\mathbb{Z}[\tau]$: if $u_{l-1}\tau^{l-1} + \cdots + u_1\tau + u_0 \in \mathbb{Z}[\tau]$ and $P \in E_a(\mathbb{F}_{2^m})$, then

$$(u_{l-1}\tau^{l-1} + \cdots + u_1\tau + u_0)P = u_{l-1}\tau^{l-1}(P) + \cdots + u_1\tau(P) + u_0P. \quad (8)$$

The strategy for developing an efficient point multiplication algorithm is find a "nice" expression for k of the form $k = \sum_{i=0}^{l-1} u_i\tau^i$, and then use (8) to compute kP. Here, "nice" means that l is relatively small and the non-zero coefficients u_i are small (e.g., ± 1) and sparse.

Since $\tau^2 + 2 = \mu\tau$, every element in $\mathbb{Z}[\tau]$ can be expressed in canonical form $r_0 + r_1\tau$, where $r_0, r_1 \in \mathbb{Z}$. $\mathbb{Z}[\tau]$ is a Euclidean domain, and hence also a unique factorization domain, with respect to the norm function $N(r_0 + r_1\tau) = r_0^2 + \mu r_0 r_1 + 2r_1^2$. The norm function is multiplicative. We have $N(\tau) = 2$, $N(\tau - 1) = h$, $N(\tau^m - 1) = \#E_a(\mathbb{F}_{2^m})$, and $N(\delta) = n$ where $\delta = (\tau^m - 1)/(\tau - 1)$.

A *τ-adic NAF* or *TNAF* of an element $\kappa \in \mathbb{Z}[\tau]$ is an expression $\kappa = \sum_{i=0}^{l-1} u_i\tau^i$ where $u_i \in \{0, \pm 1\}$, and no two consecutive coefficients u_i are nonzero. Every $\kappa \in \mathbb{Z}[\tau]$ has a unique TNAF, denoted TNAF(κ), which can be efficiently computed using Algorithm 18.

Algorithm 18. Computing the TNAF of an element in $\mathbb{Z}[\tau]$

INPUT: $\kappa = r_0 + r_1\tau \in \mathbb{Z}[\tau]$.
OUTPUT: TNAF(κ).

1. $i \leftarrow 0$.
2. While $r_0 \neq 0$ or $r_1 \neq 0$ do
   2.1 If r_0 is odd then: $u_i \leftarrow 2 - (r_0 - 2r_1 \bmod 4)$, $r_0 \leftarrow r_0 - u_i$;
   2.2 Else: $u_i \leftarrow 0$.
   2.3 $t \leftarrow r_0$, $r_0 \leftarrow r_1 + \mu r_0/2$, $r_1 \leftarrow -t/2$, $i \leftarrow i + 1$.
3. Return$((u_{i-1}, u_{i-2}, \ldots, u_1, u_0))$.

To compute kP, one can find TNAF(k) using Algorithm 18, and then use (8). Now, the length $l(\alpha)$ of TNAF(α) satisfies $\log_2(N(\alpha)) - 0.55 < l(\alpha) <$

$\log_2(N(\alpha)) + 3.52$ when $l \geq 30$. It follows that $l(k) \approx 2\log_2 k$, which is twice as long as the length of NAF(k). To circumvent the problem of a long TNAF, notice that if $\rho = k \bmod \delta$ then $kP = \rho P$ for all points P of order n (because $\delta P = \mathcal{O}$). Since $N(\rho) < N(\delta) = n$, it follows that $l(\rho) \approx m$, which suggests that TNAF(ρ) should be used instead of TNAF(k) for computing kP. Algorithm 19 is an efficient method for computing an element $\rho' \in \mathbb{Z}[\tau]$ such that $\rho' \equiv k \pmod{\delta}$; we write $\rho' = k$ partmod δ. The parameter C ensures that TNAF(ρ') is not much longer than TNAF(ρ). In fact, $l(\rho) \leq m + a$, and if $C \geq 2$ then $l(\rho') \leq m + a + 3$. Also, the probability that $\rho' \neq \rho$ is less than $2^{-(C-5)}$.

Algorithm 19. Partial reduction modulo δ

INPUT: $k \in [1, n-1]$, $C \geq 2$, $s_0 = d_0 + \mu d_1$, $s_1 = -d_1$, where $\delta = d_0 + d_1\tau$.
OUTPUT: $\rho' = k$ partmod δ.

1. $k' \leftarrow \lfloor k/2^{a-C+(m-9)/2} \rceil$.
2. For i from 0 to 1 do
   2.1 $g' \leftarrow s_i \cdot k'$, $j' \leftarrow V_m \cdot \lfloor g'/2^m \rceil$, $\lambda_i \leftarrow \lfloor (g' + j')/2^{(m+5)/2} + \frac{1}{2} \rfloor / 2^C$.
   2.2 $f_i \leftarrow \lfloor \lambda_i + \frac{1}{2} \rfloor$, $\eta_i \leftarrow \lambda_i - f_i$, $h_i \leftarrow 0$.
3. $\eta \leftarrow 2\eta_0 + \mu\eta_1$.
4. If $\eta \geq 1$ then
   4.1 If $\eta_0 - 3\mu\eta_1 < -1$ then $h_1 \leftarrow \mu$; else $h_0 \leftarrow 1$.
   Else
   4.2 If $\eta_0 + 4\mu\eta_1 \geq 2$ then $h_1 \leftarrow \mu$.
5. If $\eta < -1$ then
   5.1 If $\eta_0 - 3\mu\eta_1 \geq 1$ then $h_1 \leftarrow -\mu$; else $h_0 \leftarrow -1$.
   Else
   5.2 If $\eta_0 + 4\mu\eta_1 < -2$ then $h_1 \leftarrow -\mu$.
6. $q_0 \leftarrow f_0 + h_0$, $q_1 \leftarrow f_1 + h_1$, $r_0 \leftarrow k - (s_0 + \mu s_1)q_0 - 2s_1 q_1$, $r_1 \leftarrow s_1 q_0 - s_0 q_1$.
7. Return$(r_0 + r_1\tau)$.

The average density of non-zero coefficients among all TNAFs of length l is approximately $1/3$. Hence Algorithm 20 which uses TNAF(ρ') for computing kP has an expected running time of approximately $(m/3)A$.

Algorithm 20. TNAF method for point multiplication

INPUT: TNAF$(\rho') = \sum_{i=0}^{l-1} u_i \tau^i$ where $\rho' = k$ partmod δ, $P \in E_a(\mathbb{F}_{2^m})$.
OUTPUT: kP.

1. $Q \leftarrow \mathcal{O}$.
2. For i from $l-1$ downto 0 do
   2.1 $Q \leftarrow \tau Q$.
   2.2 If $u_i = 1$ then $Q \leftarrow Q + P$.
   2.3 If $u_i = -1$ then $Q \leftarrow Q - P$.
3. Return(Q).

We now extend Algorithm 20 to a window method analogous to Algorithm 14. Let $t_w = 2U_{w-1}U_w^{-1} \bmod 2^w$, where $\{U_k\}$ is the Lucas sequence defined by $U_0 = 0$, $U_1 = 1$, $U_{k+1} = \mu U_k - 2U_{k-1}$ for $k \geq 1$. Then the map $\phi_w : \mathbb{Z}[\tau] \to \mathbb{Z}_{2^w}$ induced by $\tau \mapsto t_w$ is a surjective ring homomorphism with kernel $\{\alpha \in \mathbb{Z}[\tau] : \tau^w | \alpha\}$. It follows that a set of distinct representatives of the congruence classes

modulo τ^w whose elements are not divisible by τ is $\{\pm 1, \pm 3, \ldots, \pm(2^{w-1} - 1)\}$. Define $\alpha_i = i \bmod \tau^w$ for $i \in \{1, 3, \ldots, 2^{w-1} - 1\}$. A *width-w TNAF* of $\kappa \in \mathbb{Z}[\tau]$, denoted $\mathrm{TNAF}_w(\kappa)$, is an expression $\kappa = \sum_{i=0}^{l-1} u_i \tau^i$, where $u_i \in \{0, \pm\alpha_1, \pm\alpha_3, \ldots, \pm\alpha_{2^{w-1}-1}\}$, and at most one of any w consecutive coefficients is nonzero. Algorithm 21 is an efficient method for computing $\mathrm{TNAF}_w(\kappa)$.

Algorithm 21. Computing a width-w TNAF of an element in $\mathbb{Z}[\tau]$

INPUT: w, t_w, $\alpha_u = \beta_u + \gamma_u\tau$ for $u \in \{1, 3, \ldots, 2^{w-1} - 1\}$, $\rho = r_0 + r_1\tau \in \mathbb{Z}[\tau]$.
OUTPUT: $\mathrm{TNAF}_w(\rho)$.

1. $i \leftarrow 0$.
2. While $r_0 \neq 0$ or $r_1 \neq 0$ do
   2.1 If r_0 is odd then
   $u \leftarrow r_0 + r_1 t_w \bmod 2^w$.
   If $u > 0$ then $s \leftarrow 1$; else $s \leftarrow -1$, $u \leftarrow -u$.
   $r_0 \leftarrow r_0 - s\beta_u$, $r_1 \leftarrow r_1 - s\gamma_u$, $u_i \leftarrow s\alpha_u$.
   2.2 Else: $u_i \leftarrow 0$.
   2.3 $t \leftarrow r_0$, $r_0 \leftarrow r_1 + \mu r_0/2$, $r_1 \leftarrow -t/2$, $i \leftarrow i + 1$.
3. Return$((u_{i-1}, u_{i-2}, \ldots, u_1, u_0))$.

The average density of non-zero coefficients among all TNAF_ws of length l is approximately $1/(w + 1)$. Since the length of $\mathrm{TNAF}_w(\rho')$ is approximately $l(\rho')$, it follows that Algorithm 22 which uses $\mathrm{TNAF}(\rho')$ for computing kP has an expected running time of approximately $(2^{w-2} - 1 + m/(w + 1))A$.

Algorithm 22. Window TNAF method for point multiplication

INPUT: $\mathrm{TNAF}_w(\rho') = \sum_{i=0}^{l-1} u_i \tau^i$, where $\rho' = k$ partmod δ, $P \in E_a(\mathbb{F}_{2^m})$.
OUTPUT: kP.

1. Compute $P_u = \alpha_u P$, for $u \in \{1, 3, 5, \ldots, 2^{w-1} - 1\}$.
2. $Q \leftarrow \mathcal{O}$.
3. For i from $l - 1$ downto 0 do
   3.1 $Q \leftarrow \tau Q$.
   3.2 If $u_i \neq 0$ then:
   Let u be such that $\alpha_u = u_i$ or $\alpha_{-u} = -u_i$.
   If $u > 0$ then $Q \leftarrow Q + P_u$;
   Else $Q \leftarrow Q - P_{-u}$.
4. Return(Q).

If the point P is fixed, then the points P_u in step 1 of Algorithm 22 can be precomputed. The resulting method, which we call fixed-base window TNAF (or Algorithm 23), has an expected running time of $(m/(w + 1))A$.

Table 6 lists the expected number of elliptic curve additions for point multiplication using the window TNAF and fixed-base window TNAF methods for the fields $\mathbb{F}_{2^{163}}$, $\mathbb{F}_{2^{233}}$ and $\mathbb{F}_{2^{283}}$. In our implementations, we chose window width $w = 5$ for the window TNAF method and $w = 6$ for the fixed-base window TNAF method.

Table 6. Estimates for window TNAF and fixed-base window TNAF costs at various window widths.

Window width w	Number of precomputed points	Number of elliptic curve additions					
		Fixed-base window TNAF			Window TNAF		
		$m = 163$	$m = 233$	$m = 283$	$m = 163$	$m = 233$	$m = 283$
2	0	54	78	94	54	78	94
3	1	41	58	71	42	59	72
4	3	33	47	57	36	50	60
5	7	27	39	47	34	46	54
6	15	23	33	40	38	48	55
7	31	20	29	35	51	64	66

5.3 Timings

In Table 7 we present rough estimates of costs in terms of both elliptic curve operations and field operations for the various point multiplication methods in the case $m = 163$. These estimates serve as a guideline for comparing point multiplication algorithms without concern for platform or implementation specifics.

Table 8 presents timing results for the NIST curves B-163, B-233, B-283, K-163, K-233 and K-283. The implementation was done in C and the timings were obtained on a Pentium II 400 MHz workstation. The big number library in OpenSSL [35] was used to perform multiprecision integer arithmetic.

The timings in Table 8 are consistent with the estimates in Table 7. In general, point multiplication on Koblitz curves is significantly faster than on random curves. The difference is especially pronounced in the case where P is not known a priori (Montgomery vs. window TNAF). For the window TNAF method with $w = 5$ and $m = 163$, the timings for the three components were 50 μs for partial reduction (Algorithm 19), 126μs for width-w TNAF computation (Algorithm 21), and 1266μs for elliptic curve operations (Algorithm 22).

6 ECDSA Elliptic Curve Operations

The execution times of elliptic curve cryptographic schemes such as the ECDSA [16,21] are typically dominated by point multiplications. In ECDSA, there are two types of point multiplications, kP where P is fixed (signature generation), and $kP+lQ$ where P is fixed and Q is not known a priori (signature verification). One method to speed the computation of $kP+lQ$ is simultaneous multiple point multiplication (Algorithm 24), also known as Shamir's trick [8]. Algorithm 24 has an expected running time of $(2^{2w}-3)A+((d-1)(2^{2w}-1)/2^{2w}A+(d-1)wD)$, and requires storage for 2^{2w} points.

Table 7. Rough estimates of point multiplication costs for $m = 163$.

Method	Coordinates	w	Points stored	EC operations A	EC operations D	Field operations M	Field operations I	Field operations Total[a]
Binary	affine	—	0	82	163	490	245	2940
(Algorithm 11)	projective	—	0	82	163	1390	1	1400
Binary NAF	affine	—	0	54	163	434	217	2604
(Algorithm 13)	projective	—	0	54	163	1140	1	1150
Window NAF	affine	4	3	36	164	400	200	2400
(Algorithm 14)	projective	4	3	3^b+33	164	955	5	1005
Montgomery	affine	—	0	163^c	163	329	327	3600
(Algorithm 15)	projective	—	0	163^c	163	988	1	998
Fixed-base window	affine	6	27	89	0	178	89	1068
(Algorithm 16)	projective	6	27	27+62^d	0	1113	1	1123
Fixed-base comb	affine	4	14	38	40	156	78	936
(Algorithm 17)	projective	4	14	38	40	504	1	514
TNAF	affine	—	0	54	0	108	54	648
(Algorithm 20)	projective	—	0	54	0	488	1	498
Window TNAF	affine	5	7	34	0	68	34	408
(Algorithm 22)	projective	5	7	7^b+27	0	261	8	341
Fixed-base window TNAF	affine	6	15	23	0	46	23	276
(Algorithm 23)	projective	6	15	23	0	209	1	219

[a] Total cost in field multiplications assuming $1I = 10M$.
[b] Additions are in affine coordinates
[c] Additions using formula (5).
[d] Additions are not in mixed coordinates.

Table 8. Timings (in μs) for point multiplication on random and Koblitz curves over $\mathbb{F}_{2^{163}}$, $\mathbb{F}_{2^{233}}$ and $\mathbb{F}_{2^{283}}$. Unless otherwise stated, projective coordinates were used.

	$m = 163$	$m = 233$	$m = 283$
Random curves			
Binary (Alg 11, affine coordinates)	9178	21891	34845
Binary (Alg 11)	4716	10775	16123
Binary NAF (Alg 13)	4002	9303	13896
Window NAF with $w = 4$ (Alg 14)	3440	7971	11997
Montgomery (Alg 15)	3240	7697	11602
Fixed-base comb with $w = 4$ (Alg 17)	1683	3966	5919
Koblitz curves			
TNAF (Alg 20)	1946	4349	6612
Window TNAF with $w = 5$ (Alg 22)	1442	2965	4351
Fixed-base window TNAF with $w = 6$ (Alg 23)	1176	2243	3330

Algorithm 24. Simultaneous multiple point multiplication

INPUT: Window width w, $k = (k_{t-1}, \ldots, k_1, k_0)_2$, $l = (l_{t-1}, \ldots, l_1, l_0)_2$, P, Q.
OUTPUT: $kP + lQ$.

1. Compute $iP + jQ$ for all $i, j \in [0, 2^w - 1]$.
2. Write $k = (k^{d-1}, \ldots, k^1, k^0)$ and $l = (l^{d-1}, \ldots, l^1, l^0)$ where each k^i and l^i is a bitstring of length w, and $d = \lceil t/w \rceil$.
3. $R \leftarrow \mathcal{O}$.
4. For i from $d-1$ downto 0 do
   4.1 $R \leftarrow 2^w R$.
   4.2 $R \leftarrow R + (k^i P + l^i Q)$.
5. Return(R).

Table 9 lists the most efficient methods for computing kP, P fixed, for random curves and Koblitz curves. For each type of curve, two cases are distinguished—when there is no extra memory available and when memory is not heavily constrained. Table 10 does the same for computing $kP + lQ$ where P is fixed and Q is not known a priori.

Table 9. Timings (in μs) of the fastest methods for point multiplication kP, P fixed, in ECDSA signature generation.

Curve type	Memory constrained?	Fastest method	m=163	m=233	m=283
Random	No	Fixed-base comb ($w = 4$)	1683	3966	5919
	Yes	Montgomery	3240	7697	11602
Koblitz	No	Fixed-base window TNAF (w=6)	1176	2243	3330
	Yes	TNAF	1946	4349	6612

Table 10. Timings (in μs) of the fastest methods for point multiplications $kP + lQ$, P fixed and Q not known a priori, in ECDSA signature verification.

Curve type	Memory constrained?	Fastest method	m=163	m=233	m=283
Random	No	Montgomery + Fixed-base comb ($w = 4$)	5005	11798	17659
	No	Simultaneous ($w = 2$)	4969	11332	16868
	Yes	Montgomery	6564	15531	23346
Koblitz	No	Window TNAF ($w = 5$) + Fixed-base window TNAF (w=6)	2702	5348	7826
	Yes	TNAF	3971	8832	13374

7 Conclusions

We found that significant performance improvements can be achieved by the use of projective coordinates over affine coordinates due to the high inversion to multiplication ratio observed in our implementation.

Implementing the specialized algorithms for Koblitz curves is straightforward. Point multiplication for Koblitz curves is considerably faster than on random curves, yielding faster implementations of elliptic curve cryptographic schemes. For both random and Koblitz curves, substantial performance improvements can be obtained with only a modest commitment of memory for storage of tables and precomputed data.

While some effort was made to optimize the code, it is likely that considerable performance enhancements can be obtained especially if the code is tuned for a specific platform. For example, the times for the AIA and MAIA methods (see §3.5) compared with inversion using EEA require some explanation. Even with optimization efforts (but in C only) and a suitable reduction trinomial in the $m = 233$ case, we found that the EEA implementation was significantly faster on the Pentium II. Non-optimal register allocation may have contributed to the relatively poor showing of AIA and MAIA, suggesting that a few hand-coded assembly sections may be desirable. Even with the same source code, compiler and hardware differences are apparent. On a Sun Ultra, for example, we found that EEA required roughly 9 times as long as multiplication using the same code as on the Pentium II, and AIA and MAIA required approximately the same time as inversion using the EEA.

Despite the limitations of our analysis and implementation, we nonetheless hope that our work will serve as a benchmark for future efforts in this area.

8 Future Work

We did not implement the variant of Montgomery integer multiplication for $\mathbb{F}_{2^m}$ presented in [22]. We also did not implement the point multiplication method of [17,36] which uses point halvings instead of doublings since this method appears to be advantageous only when affine coordinates are employed.

We are currently investigating the software implementation of ECC over the NIST-recommended prime fields, and a comparison with the NIST-recommended binary fields. A careful and extensive study of ECC implementation in software for constrained devices such as smart cards, and in hardware, would be beneficial to practitioners. Also needed is a thorough comparison of the implementation of ECC, RSA, and discrete logarithm systems on various platforms, continuing the work reported in [7].

Acknowledgements

The authors would like to thank Mike Brown, Donny Cheung, Eric Fung, and Mike Kirkup for numerous fruitful discussions and for help with the implementation and timings.

References

1. ANSI X9.62, *Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)*, 1999.
2. ANSI X9.63, *Public Key Cryptography for the Financial Services Industry: Elliptic Curve Key Agreement and Key Transport Protocols*, working draft, August 1999.
3. E. Brickell, D. Gordon, K. McCurley and D. Wilson, "Fast exponentiation with precomputation", *Advances in Cryptology – Eurocrypt '92*, LNCS **658**, 1993, 200-207.
4. M. Brown, D. Cheung, D. Hankerson, J. Hernandez, M. Kirkup and A. Menezes, "PGP in constrained wireless devices", *Proceedings of the Ninth USENIX Security Symposium*, 2000.
5. D. Chudnovsky and G. Chudnovsky, "Sequences of numbers generated by addition in formal groups and new primality and factoring tests", *Advances in Applied Mathematics*, **7** (1987), 385-434.
6. E. De Win, A. Bosselaers, S. Vandenberghe, P. De Gersem and J. Vandewalle, "A fast software implementation for arithmetic operations in $GF(2^n)$", *Advances in Cryptology – Asiacrypt '96*, LNCS **1163**, 1996, 65-76.
7. E. De Win, S. Mister, B. Preneel and M. Wiener, "On the performance of signature schemes based on elliptic curves", *Algorithmic Number Theory, Proceedings Third Intern. Symp., ANTS-III*, LNCS **1423**, 1998, 252-266.
8. T. ElGamal, "A public key cryptosystem and a signature scheme based on discrete logarithms", *IEEE Transactions on Information Theory*, **31** (1985), 469-472.
9. S. Galbraith and N. Smart, "A cryptographic application of Weil descent", *Codes and Cryptography*, LNCS **1746**, 1999, 191-200.
10. P. Gaudry, F. Hess and N. Smart, "Constructive and destructive facets of Weil descent on elliptic curves", preprint, January 2000.
11. D. Gordon, "A survey of fast exponentiation methods", *Journal of Algorithms*, **27** (1998), 129-146.
12. J. Guajardo and C. Paar, "Efficient algorithms for elliptic curve cryptosystems", *Advances in Cryptology – Crypto'97*, LNCS **1294**, 1997, 342-356.
13. IEEE P1363, *Standard Specifications for Public-Key Cryptography*, 2000.
14. ISO/IEC 14888-3, *Information Technology – Security Techniques – Digital Signatures with Appendix – Part 3: Certificate Based-Mechanisms*, 1998.
15. ISO/IEC 15946, *Information Technology – Security Techniques – Cryptographic Techniques Based on Elliptic Curves*, Committee Draft (CD), 1999.
16. D. Johnson and A. Menezes, "The elliptic curve digital signature algorithm (ECDSA)", Technical report CORR 99-34, Dept. of C&O, University of Waterloo, 1999.
17. E. Knudsen, "Elliptic scalar multiplication using point halving", *Advances in Cryptology – Asiacrypt '99*, LNCS **1716**, 1999, 135-149.
18. D. Knuth, *The Art of Computer Programming – Seminumerical Algorithms*, Addison-Wesley, 3rd edition, 1998.
19. N. Koblitz, "Elliptic curve cryptosystems", *Mathematics of Computation*, **48** (1987), 203-209.
20. N. Koblitz, "CM-curves with good cryptographic properties", *Advances in Cryptology – Crypto'91*, LNCS **576**, 1992, 279-287.
21. N. Koblitz, A. Menezes and S. Vanstone, "The state of elliptic curve cryptography", *Designs, Codes and Cryptography*, **19** (2000), 173-193.
22. Ç. K. Koç and T. Acar, "Montgomery multiplication in $GF(2^k)$", *Designs, Codes and Cryptography*, **14** (1998), 57-69.

23. K. Koyama and Y. Tsuruoka, "Speeding up elliptic cryptosystems by using a signed binary window method", *Advances in Cryptology – Crypto'92*, LNCS **740**, 1993, 345-357.
24. C. Lim and P. Lee, "More flexible exponentiation with precomputation", *Advances in Cryptology – Crypto'94*, LNCS **839**, 1994, 95-107.
25. J. López and R. Dahab, "Improved algorithms for elliptic curve arithmetic in $GF(2^n)$", *Selected Areas in Cryptography – SAC '98*, LNCS **1556**, 1999, 201-212.
26. J. López and R. Dahab, "Fast multiplication on elliptic curves over $GF(2^n)$ without precomputation", *Cryptographic Hardware and Embedded Systems – CHES '99*, LNCS **1717**, 1999, 316-327.
27. J. López and R. Dahab, "High-speed software multiplication in $\mathbb{F}_{2^m}$", preprint, 2000.
28. A. Menezes, P. van Oorschot and S. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 1997.
29. V. Miller, "Uses of elliptic curves in cryptography", *Advances in Cryptology – Crypto'85*, LNCS **218**, 1986, 417-426.
30. A. Miyaji, T. Ono and H. Cohen, "Efficient elliptic curve exponentiation", *Proceedings of ICICS '97*, LNCS **1334**, 1997, 282-290.
31. P. Montgomery, "Speeding up the Pollard and elliptic curve methods of factorization", *Mathematics of Computation*, **48** (1987), 243-264.
32. F. Morain and J. Olivos, "Speeding up the computations on an elliptic curve using addition-subtraction chains", *Informatique théorique et Applications*, **24** (1990), 531-544.
33. National Institute of Standards and Technology, *Digital Signature Standard*, FIPS Publication 186-2, February 2000.
34. National Institute of Standards and Technology, *Advanced Encryption Standard*, work in progress.
35. OpenSSL, `http://www.openssl.org`
36. R. Schroeppel, "Elliptic curve point halving wins big", preprint, 2000.
37. R. Schroeppel, H. Orman, S. O'Malley and O. Spatscheck, "Fast key exchange with elliptic curve systems", *Advances in Cryptology – Crypto'95*, LNCS **963**, 1995, 43-56.
38. J. Solinas, "Efficient arithmetic on Koblitz curves", *Designs, Codes and Cryptography*, **19** (2000), 195-249.

Implementation of Elliptic Curve Cryptographic Coprocessor over $GF(2^m)$ on an FPGA

Souichi Okada, Naoya Torii, Kouichi Itoh, and Masahiko Takenaka

Fujitsu Laboratories Ltd.
64 Nishiwaki, Ohkubo-cho, Akashi 674-8555, Japan
{sokada,torii,kito,takenaka}@flab.fujitsu.co.jp

Abstract. We describe the implementation of an elliptic curve cryptographic (ECC) coprocessor over $GF(2^m)$ on an FPGA and also the result of simulations evaluating its LSI implementation. This coprocessor is suitable for server systems that require efficient ECC operations for various parameters. For speeding-up an elliptic scalar multiplication, we developed a novel configuration of a multiplier over $GF(2^m)$, which enables the multiplication of any bit length by using our data conversion method. The FPGA implementation of the coprocessor with our multiplier, operating at 3 MHz, takes 80 ms for 163-bit elliptic scalar multiplication on a pesudo-random curve and takes 45 ms on a Koblitz curve. The 0.25 μm ASIC implementation of the coprocessor, operating at 66 MHz and having a hardware size of 165 Kgates, would take 1.1 ms for 163-bit elliptic scalar multiplication on a pesudo-random curve and would take 0.65 ms on a Koblitz curve.
Keywords: Elliptic curve cryptography (ECC), coprocessor, elliptic scalar multiplication over $GF(2^m)$, IEEE P1363, Koblitz curve, multiplier.

1 Introduction

We describe the implementation of an elliptic curve cryptographic (ECC) [8] [13] [12] coprocessor over $GF(2^m)$ that is suitable for server systems. A cryptographic coprocessor for server systems must be flexible and provide a high-performance to process a large number of requests from various types of clients. A flexible coprocessor should be able to operate for various elliptic curve parameters. For example, it should be able to operate for arbitrary irreducible polynomials at any bit length. We therefore chose a polynomial basis (PB), because with reasonable hardware size it provides more flexibility than a normal basis (NB). And a high-performance coprocessor should perform fast elliptic scalar multiplication. The elliptic scalar multiplication is based on the multiplication over $GF(2^m)$. We therefore developed and implemented an efficient algorithm for bit parallel multiplication over $GF(2^m)$.

There have been many proposals regarding fast multipliers over $GF(2^m)$ [10] [7] [11]. A classical bit parallel multiplier made by piling up bit serial multipliers (each of which is known as a linear feedback shift register (LFSR) [10]) was

Ç.K. Koç and C. Paar (Eds.): CHES 2000, LNCS 1965, pp. 25–40, 2000.

proposed by Laws [10] and improved by Im [7]. One of the fastest multipliers was proposed by Mastrovito [11] but it is extremely difficult to implement with reasonable hardware size if the irreducible polynomials and the bit length are not fixed.

There have also been studies concerned with the hardware implementation of an ECC over $GF(2^m)$ [1] [2] [16] [15] [5]. The hardware described in [1] and [2] is based on NB. And that described in [16] is based on composite field arithmetic on PB. To reduce the hardware size needed for implementation of the ECC on PB, some new multiplier have been proposed. The basic idea behind them is that an m-bit$\times m$-bit multiplication is calculated by a w_1-bit$\times w_2$-bit multiplier (where $w_1, w_2 \leq m$). Hasan proposed a look-up table based algorithm for $GF(2^m)$ multiplication [5]. This method uses an m-bit$\times w_2$-bit multiplier. And Orlando and Paar developed a new sliced PB multiplier, called a super serial multiplier (SSM) [15], which is based on the LFSR. The SSM uses a w_1-bit$\times$1-bit multiplier.

In this paper we describe a fast multiplier over $GF(2^m)$ that can operate for arbitrary irreducible polynomials at any bit length, and we also describe the implementation of an ECC coprocessor with this multiplier on an FPGA. Our multiplier has two special characteristics. Our multiplier architecture extends the concept of the SSM. That is, our multiplier folds the bit parallel multiplier, whereas the SSM folds the LFSR. Our multiplier is a w_1-bit$\times w_2$-bit multiplier (where $w_1 > w_2$, $w_1, w_2 \leq m$), which offers better performance when w_1 is larger or w_2 is smaller in case of fixed hardware size. Our multiplier also does fast multiplication at any bit length by using a new data conversion method, in which the data is converted, a sequence of multiplications is done, and the result is obtained by inverse conversion. This method enables fast operation when a sequence of multiplications is required, as in ECC calculation.

We implemented an ECC coprocessor with our multiplier on a field programmable gate array (FPGA), EPF10K250AGC599-2 by ALTERA[3]. Our coprocessor performs a fast elliptic scalar multiplication on a pseudo-random curve [14] and a Koblitz curve [14] [9] for various parameters. It operates at 3 MHz and includes an 82-bit$\times$4-bit multiplier. For 163-bit elliptic scalar multiplication, it takes 80 ms on a pseudo-random curve and 45 ms on a Koblitz curve. We also evaluated the performance and the hardware size of our coprocessor with 0.25 μm ASIC by FUJITSU [4]. Our coprocessor can operate at up to 66 MHz using a 288-bit$\times$8-bit multiplier and its hardware size is about 165 Kgates. For 163-bit elliptic scalar multiplication, it takes 1.1 ms on a pseudo-random curve and 0.65 ms on a Koblitz curve. And for 571-bit elliptic scalar multiplication, it takes 22 ms on a pseudo-random curve and 13 ms on a Koblitz curve.

We describe our multiplication algorithm in section 2, the configuration of our multiplier in section 3, and the ECC coprocessor implementation in section 4.

2 Multiplication Algorithm

2.1 Polynomial Representation

In this paper we represent elements over $GF(2^m)$ in three different types: **bit-string**, **word-string**, and **block-string**. An element a over $GF(2^m)$ is expressed as a polynomial of degree less than m. That is,

$$a(x) = \sum_{i=0}^{m-1} a_i x^i, (a_i \in GF(2)).$$

In the bit-string the element a is represented as $a = (a_{m-1}, a_{m-2}, \ldots, a_0)$.

In the word-string the element a is represented with words which have a w_2-bit length. We denote the i-th word as A_i, and it can be represented with a bit-string as $A_i = (a_{w_2 \cdot i + w_2 - 1}, a_{w_2 \cdot i + w_2 - 2}, \ldots, a_{w_2 \cdot i})$. When $m = n_2 \cdot w_2$, the element a can be represented as $a = (A_{n_2-1}, A_{n_2-2}, \ldots, A_0)$ and we can express the element a by using the following equations:

$$a(x) = \sum_{i=0}^{n_2-1} A_i(x) \cdot x^{w_2 \cdot i},$$

$$A_j(x) = \sum_{k=0}^{w_2-1} a_{w_2 \cdot j + k} \cdot x^k.$$

In the block-string the element a is represented with blocks, which are sequences of words. We denote the block $\boldsymbol{A}_{[i,j]} = (A_i, A_{i-1}, \ldots, A_j)$ (where $i \geq j$). When $m = n_1 \cdot w_1$ and $w_1 = s \cdot w_2$, we can express the element a by using the following equations:

$$a(x) = \sum_{i=0}^{n_1-1} \boldsymbol{A}_{[s \cdot i + s - 1, s \cdot i]}(x) \cdot x^{w_1 \cdot i},$$

$$\boldsymbol{A}_{[s \cdot i + s - 1, s \cdot i]} = \sum_{j=0}^{s-1} A_{s \cdot i + j}(x) \cdot x^{w_2 \cdot j}.$$

2.2 Irreducible Polynomial Representation

The irreducible polynomial $f(x)$ over $GF(2^m)$ can be represented as

$$f(x) = x^m + \sum_{i=0}^{m-1} f_i \cdot x^i, \ (f_i \in GF(2)).$$

And the lowest m-bit sequence of f_i is denoted as $f^* = (f_{m-1}, f_{m-2}, \ldots, f_0)$. In this paper we also call $f^*(x)$ an irreducible polynomial.

2.3 Partial Multiplication Algorithm

In this section, we describe the multiplication algorithm over $GF(2^m)$ with w_1-bit$\times w_2$-bit partial multiplications. The multiplication with m-bit$\times w_2$-bit partial multiplication algorithm over $GF(2^m)$ has already reported by Hasan [5]. We extend this algorithm to w_1-bit$\times w_2$-bit partial multiplication.

Multiplication over $GF(2^m)$.
The multiplication algorithm over $GF(2^m)$ is the following Algorithm 1.

Algorithm 1.
Input : $a(x), b(x), f(x)$
Output : $r(x) = a(x) \cdot b(x) \pmod{f(x)}$

Step 1. $t(x) = a(x) \cdot b(x)$
Step 2. $e(x) = \lfloor t(x)/f(x) \rfloor$
Step 3. $r(x) = t(x) + e(x) \cdot f(x)$

Here $t(x)$ is a temporary variable, and $\lfloor t(x)/f(x) \rfloor$ is a quotient in which $t(x)$ is divided by $f(x)$.

Multiplication with m-Bit$\times w_2$-Bit Partial Multiplications.
We show Algorithm 2 so that $b(x)$ is handled word-by-word. This algorithm is based on the multiplication reported by Hasan [5].

Algorithm 2.
Input : $a(x), b(x), f(x)$
Output : $r(x) = a(x) \cdot b(x) \pmod{f(x)}$

Step 1. $r(x) = 0$
Step 2. *for* $(j = n_2 - 1; j \geq 0; j = j - 1)$ {
Step 3. $t(x) = r(x) \cdot x^{w_2} + a(x) \cdot B_j(x)$
Step 4. $e(x) = \lfloor t(x)/f(x) \rfloor$
Step 5. $r(x) = t(x) + e(x) \cdot f(x)$
Step 6. }

Multiplication with w_1-bit$\times w_2$-bit Partial Multiplications.
In Algorithm 2, $r(x)$ is calculated by m-bit$\times w_2$-bit partial multiplications. We have extended Algorithm 2 the following Algorithm 3 in which $a(x)$ is handled block-by-block.

Algorithm 3. (Proposed Algorithm)
Input : $a(x), b(x), f(x)$
Output : $r(x) = a(x) \cdot b(x) \pmod{f(x)}$

Step 1. $r(x) = 0$
Step 2. *for* $(j = n_2 - 1; j \geq 0; j = j - 1)$ {
Step 3. $Uc(x) = 0$

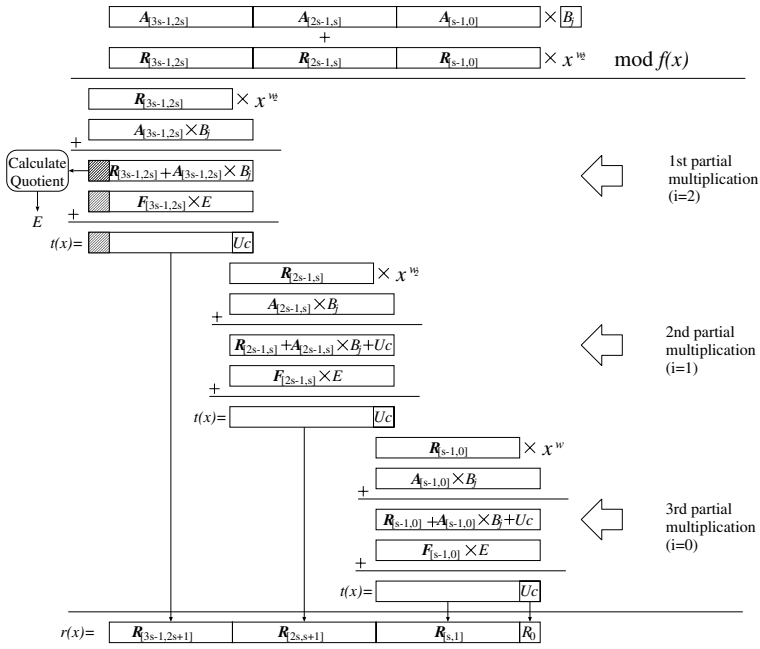


Fig. 1. m-bit×m-bit Multiplication with w_1-bit×w_2-bit Partial Multiplications (n_1=3).

Step 4. *for* $(i = n_1 - 1; i \geq 0; i = i - 1)$ {
Step 5. $\boldsymbol{T}_{[s,0]}(x) = \boldsymbol{R}_{[s\cdot i+s-1,s\cdot i]}(x)\cdot x^{w_2} + Uc(x)\cdot x^{w_1} + \boldsymbol{A}_{[s\cdot i+s-1,s\cdot i]}(x)\cdot B_j(x)$
Step 6. *if* $(i == n_1 - 1)$ $E(x) = \lfloor \boldsymbol{T}_{[s,0]}(x) \cdot x^{(n_1-1)\cdot w_1} / f(x) \rfloor$
Step 7. $\boldsymbol{T}_{[s,0]}(x) = \boldsymbol{T}_{[s,0]}(x) + E(x)\cdot \boldsymbol{F}_{[s\cdot i+s-1,s\cdot i]}(x)$
Step 8. *if* $(i == n_1 - 1)$ $\boldsymbol{R}_{[s\cdot i+s-1,s\cdot i+1]}(x) = \boldsymbol{T}_{[s-1,1]}(x)$
Step 9. *else* $\boldsymbol{R}_{[s\cdot i+s,s\cdot i+1]}(x) = \boldsymbol{T}_{[s,1]}(x)$
Step 10. $Uc(x) = T_0(x)$
Step 11. }
Step 12. $R_0(x) = Uc(x)$
Step 13. }

Figure 1 shows the calculations of Algorithm 3 from step 4 to step 12 when $n_1 = 3$. In Figure 1 there are three partial multiplications corresponding to step 5. Uc is the least significant word of intermediate value $t(x)$ and is added to the next partial product as the most significant word. The part of $r(x)$ is substituted with the highest s-word of $t(x)$. In the third partial multiplication, the lowest word of $r(x)$ is substituted with $Uc(x)$. Note that, $E(x)$ calculated in step 6 when $i = n_1 - 1$ is used when $i = n_1 - 1, n_1 - 2, \ldots, 0$ in step 7. This operation enables the partial multiplication over $GF(2^m)$.

2.4 An Example of Algorithm 3

In this example, $f(x) = x^{12} + x^{11} + x^8 + x^6 + 1$ over $GF(2^{12})$, $w_1 = 4$, and $w_2 = 2$. Thus the 12-bit×12-bit multiplication is executed by a 4-bit×2-bit partial multiplication When, $n_1 = 3$, $n_2 = 6$, $s = 2$,

$$a = (\boldsymbol{A}_{[5,4]}, \boldsymbol{A}_{[3,2]}, \boldsymbol{A}_{[1,0]}) = (A_5, A_4, A_3, A_2, A_1, A_0) = (11, 00, 10, 10, 10, 01),$$
$$b = (\boldsymbol{B}_{[5,4]}, \boldsymbol{B}_{[3,2]}, \boldsymbol{B}_{[1,0]}) = (B_5, B_4, B_3, B_2, B_1, B_0) = (01, 11, 00, 01, 11, 10),$$
$$f = (1100101000001), \text{ and } (F_5, F_4, F_3, F_2, F_1, F_0) = (10, 01, 01, 00, 00, 01).$$

The following is an example of the 12-bit×2-bit partial multiplication using Algorithm 3 when $j = 0$. For simplicity, we consider only steps 4 to 12, when r, t, E, and Uc are initialized by 0.

Step 4. $i = 2$
Step 5. $\boldsymbol{T}_{[2,0]}(x) = (00, 00) \cdot x^2 + (00) \cdot x^4 + (11, 00) \cdot (10) = (01, 10, 00)$
Step 6. $\boldsymbol{E}(x) = \lfloor (01, 10, 00) \cdot x^8 / (10, 01, 01, 00, 00, 01) \rfloor = (01)$
Step 7. $\boldsymbol{T}_{[2,0]}(x) = (01, 10, 00) + (01) \cdot (10, 01) = (01, 00, 01)$
Step 8. $\boldsymbol{R}_{[5,5]}(x) = (00)$
Step 10. $Uc(x) = (01)$

Step 4. $i = 1$
Step 5. $\boldsymbol{T}_{[2,0]}(x) = (00, 00) \cdot x^2 + (01) \cdot x^4 + (10, 10) \cdot (10) = (00, 01, 00)$
Step 7. $\boldsymbol{T}_{[2,0]}(x) = (00, 01, 00) + (01) \cdot (01, 00) = (00, 00, 00)$
Step 9. $\boldsymbol{R}_{[4,3]}(x) = (00, 00)$
Step 10. $Uc(x) = (00)$

Step 4. $i = 0$
Step 5. $\boldsymbol{T}_{[2,0]}(x) = (00, 00) \cdot x^2 + (00) \cdot x^4 + (10, 01) \cdot (10) = (01, 00, 10)$
Step 7. $\boldsymbol{T}_{[2,0]}(x) = (01, 00, 10) + (01) \cdot (00, 01) = (01, 00, 11)$
Step 9. $\boldsymbol{R}_{[2,1]}(x) = (01, 00)$
Step 10. $Uc(x) = (11)$

Step 12. $R_0(x) = (11)$

From the above calculation, $r = \boldsymbol{R}_{[5,0]} = (00, 00, 00, 01, 00, 11)$ is obtained.

It is clear that when $j = 0$ we can get the same partial product as that in Algorithm 2. That is,

$$\begin{aligned} t(x) &= r(x) \cdot x^{w_2} + a(x) \cdot B_0(x) \\ &= 0 + (1100, 1010, 1001)(10) \\ &= (1, 1001, 0101, 0010) \\ e(x) &= \lfloor t/f \rfloor = 1 \\ r(x) &= t + e \cdot f = (0000, 0001, 0011). \end{aligned}$$

2.5 Calculation of Quotient $E(x)$

In step 6 of Algorithm 3, division by $f(x)$ is used to calculate wh$E(x)$. But because $f(x) = x^m + f^*(x)$, it can also be calculated with the highest w_2-bit of $T_s(x)$ in $\boldsymbol{T}_{[s,0]}(x)$ and $(f_{m-1}, f_{m-2}, \ldots, f_{m-w_2+1})$, which is the highest $(w_2 - 1)$-bit of $f^*(x)$. Algorithm 4 shows this calculation of $E(x)$.

Algorithm 4.
Input : $T_s(x), (f_{m-1}, f_{m-2}, \ldots, f_{m-w_2+1})$
Output : $E(x) = \lfloor \boldsymbol{T}_{[s,0]}(x) \cdot x^{(n_1-1)\cdot w_1} / f(x) \rfloor$

Step 1. $E(x) = 0$
Step 2. $U(x) = T_s(x)$
Step 3. *for* $(i = w_2 - 1; i \geq 0; i = i - 1)$ {
Step 4. *if* $(u_i == 1)$ {
Step 5. $e_i = 1$
Step 6. *for* $(j = i - 1; j \geq 0; j = j - 1)$ $u_j = u_j + f_{m-i+j}$
Step 7. }
Step 8. }

Here $U(x)$ is a temporary word variable.

2.6 Data Conversion Method

In the previous sections we have assumed that m is a multiple of w_1. In this section we discuss the case in which it is not. That is, m has an arbitrary bit length. Let α be the minimum positive integer that satisfies $m + \alpha = n_1 \times w_1$.

In Algorithms 3 and 4, the multiplication is processed from higher block/word to lower block/word. In Algorithm 4 the most significant bit (a coefficient of the highest degree) is used to calculate a quotient. To calculate these algorithms efficiently, the elements over $GF(2^m)$ should be converted to fill the most significant block. That is, elements should be multiplied by x^α. In addition this conversion is homomorphic, but in multiplication it is not.

Addition.

$$r(x) = a(x) + b(x) \pmod{f(x)}$$
$$\Rightarrow (a(x)x^\alpha + b(x)x^\alpha) = (a(x) + b(x))x^\alpha = r(x)x^\alpha \pmod{f(x)x^\alpha}$$

Multiplication.

$$r(x) = a(x) \cdot b(x) \pmod{f(x)}$$
$$\Rightarrow (a(x)x^\alpha)(b(x)x^\alpha) = (a(x) \cdot b(x))x^\alpha x^\alpha \neq r(x)x^\alpha \pmod{f(x)x^\alpha}$$

To solve this problem, we need to multiply either multiplier $a(x)x^\alpha$ or multiplicand $b(x)x^\alpha$ by $x^{-\alpha}$. That is,

$$(a(x)x^\alpha) \cdot (b(x)x^\alpha \cdot x^{-\alpha}) = a(x) \cdot b(x)x^\alpha = r(x)x^\alpha \pmod{f(x)x^\alpha}$$

$r(x)$ can be retrieved by multiplying the above result by $x^{-\alpha}$. These processes are inefficient and cause a large overhead when a sequence of multiplications is required, as in ECC calculation.

So we propose a method in which all the input data is converted, before the sequence of multiplications is performed. In the final step, the data is inverse converted.

The element $a(x)$ is first converted to $a(x)x^{-\alpha} \pmod{f(x)}$ and then multiplied by x^{α}, as follows:

$$\overline{a}(x) = (a(x)x^{-\alpha} \pmod{f(x)})x^{\alpha}.$$

The addition algorithm is clearly unchanged by this conversion. To see that the multiplication algorithm is unchanged, consider

$$\begin{aligned}
&\overline{a}(x) \cdot \overline{b}(x) \pmod{f(x)x^{\alpha}} \\
&= (a(x)x^{-\alpha} \pmod{f(x)})x^{\alpha} \cdot (b(x)x^{-\alpha} \pmod{f(x)})x^{\alpha} \pmod{f(x)x^{\alpha}} \\
&= (a(x) \cdot b(x)x^{-\alpha} \pmod{f(x)})x^{\alpha} \pmod{f(x)x^{\alpha}} \\
&= \overline{a(x) \cdot b(x)} \pmod{f(x)x^{\alpha}}.
\end{aligned}$$

The inverse conversion for $\overline{a}(x)$ is processed by $\overline{a}(x) \pmod{f(x)}$. That is,

$$\begin{aligned}
\overline{a}(x) \pmod{f(x)} &= ((a(x)x^{-\alpha} \pmod{f(x)})x^{\alpha}) \pmod{f(x)} \\
&= (a(x)x^{-\alpha} \cdot x^{\alpha}) \pmod{f(x)} \\
&= a(x) \pmod{f(x)}.
\end{aligned}$$

By doing this conversion, the multiplication of Algorithm 3 can be expanded for any bit length. The conversion and the inverse conversion can be summarized as follows:

$$\begin{aligned}
\text{Conversion} &: \overline{a}(x) = (a(x)x^{-\alpha} \pmod{f(x)})x^{\alpha} \\
\text{Inverse conversion} &: a(x) = \overline{a}(x) \pmod{f(x)}
\end{aligned}$$

2.7 An Example of the Data Conversion Method

In this example, $f(x) = x^5 + x^2 + 1$ over $GF(2^5)$, $w_1 = 8$, $n_1 = 1$, and $a = 3$. We convert the elements over $GF(2^5)$ for calculating with an 8-bit×8-bit multiplication.

$$\begin{aligned}
a(x) &= x^4 + 1 \\
b(x) &= x^4 + x + 1 \\
c(x) &= a(x) \cdot b(x) = x^3 + x + 1 \pmod{f(x)}
\end{aligned}$$

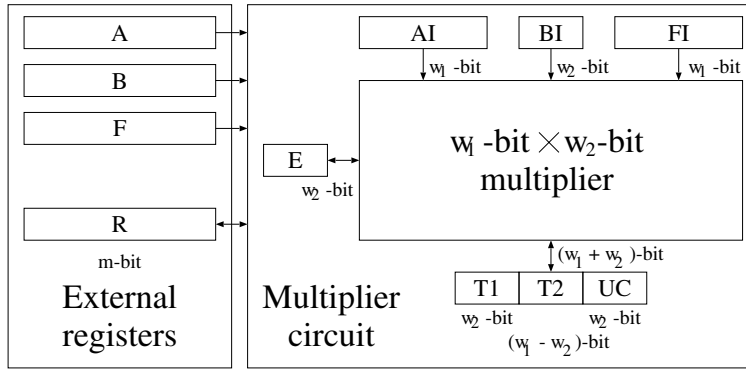


Fig. 2. Block Diagram of Our Multiplier

Conversion.
We convert $a(x)$ and $b(x)$ into $\overline{a}(x)$ and $\overline{b}(x)$ and calculate $\overline{c}(x)$, where $x^{\alpha} = x^3$ and $x^{-\alpha} = x^{-3} = x^4 + x^2 + x$.

$$\begin{aligned}\overline{a}(x) &= (a(x) \cdot x^{-\alpha} \pmod{f(x)}) \cdot x^{\alpha} = x^7 + x^5 \\ \overline{b}(x) &= (b(x) \cdot x^{-\alpha} \pmod{f(x)}) \cdot x^{\alpha} = x^7 + x^6 + x^5 + x^3\end{aligned}$$

Multiplication.

$$\begin{aligned}\overline{c}(x) &= \overline{a}(x) \cdot \overline{b}(x) \pmod{f(x) \cdot x^{\alpha}} \\ &= (x^7 + x^5) \cdot (x^7 + x^6 + x^5 + x^3) \pmod{x^8 + x^5 + x^3} \\ &= x^7 + x^6 + x^5 + x^4\end{aligned}$$

Inverse Conversion.

$$\begin{aligned}c(x) &= \overline{c}(x) \pmod{f(x) = x^5 + x^2 + 1} \\ &= x^3 + x + 1\end{aligned}$$

3 Our Multiplier

3.1 Block Diagram

Figure 2 is a block diagram of our multiplier. A, B, and F in Figure 2 are m-bit registers that store the multiplicand a, the multiplier b, and the irreducible polynomial f^*. R is an m-bit output register that stores the intermediate value of multiplication and the result. In this paper we call these registers "external registers." Each register A, F, and R is handled block by block, that is, A_i, F_i and R_i. And register B is handled word by word, that is, B_j. Moreover, R_i is divided into two sections: the highest $(w_1 - w_2)$-bit and the lowest w_2-bit. We denote

the highest $(w_1 - w_2)$-bit as RH_i and the lowest w_2-bit as RL_i . In addition, we call the registers $AI, BI, FI, E, T1, T2$, and UC in the multiplier "internal registers." AI, BI, and FI are respectively w_1-bit, w_2-bit and w_1-bit registers that store the input data, that is A_i, B_i and F_i, from the "external registers." These registers are used by the w_1-bit$\times w_2$-bit multiplier as input registers. E is a w_2-bit register that stores the intermediate value of the multiplier. $T1, T2$, and UC are respectively w_2-bit, $(w_1 - w_2)$-bit, and w_2-bit registers. They exchange the intermediate value and the result of the multiplier with R and are used by the multiplier as input and output registers.

3.2 Configuration

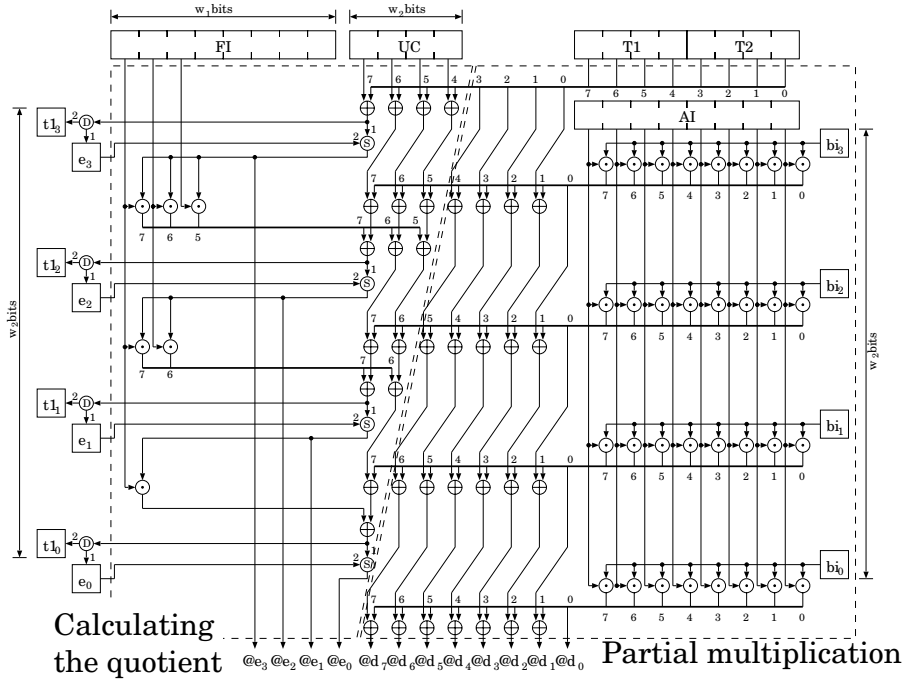
The following is the process flow for Algorithms 3 and 4 in our multiplier. Figure 3 shows our w_1-bit$\times w_2$-bit multiplier. We use $(w_1, w_2) = (8, 4)$ as an example.

The register value is denoted by the register name, and is expressed such as $AI = (ai_{w_1-1}, ai_{w_1-2}, \ldots, ai_0)$. In addition, the concatenation of a w_1-bit register AI and a w_2-bit register BI is denoted as $AI||BI$. That is, $AI||BI = (ai_{w_1-1}, ai_{w_1-2}, \ldots, ai_0, bi_{w_2-1}, bi_{w_2-2}, \ldots, bi_0)$.

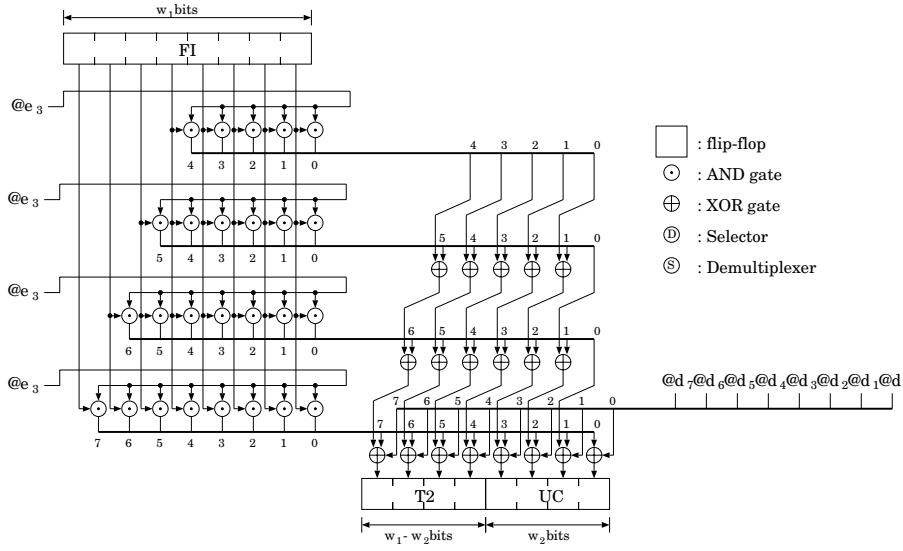
Input : a, b, f^*
Output : $r = a \cdot b \pmod f$

Proc. 1. $R \leftarrow 0$
Proc. 2. for $j = n_2 - 1$ to 0
Proc. 3. $T1 \leftarrow 0; T2 \leftarrow 0; E \leftarrow 0; UC \leftarrow 0$
Proc. 4. for $i = n_1 - 1$ to 0
Proc. 5. $(T1||T2) \leftarrow R_i$
Proc. 6. $AI \leftarrow A_i$
Proc. 7. $BI \leftarrow B_j$
Proc. 8. $FI \leftarrow F_i$
Proc. 9. $(T1||T2||UC) \leftarrow (T1||T2) \cdot x^{w_2} + UC \cdot x^{w_1} + AI \cdot BI$
Proc. 10. if $(i == n_1 - 1)$ $E \leftarrow \lfloor ((T1||T2) \cdot x^{w_2} + AI \cdot BI)/F_{n_1-1}) \rfloor$
Proc. 11. $(T1||T2||UC) \leftarrow (T1||T2||UC) + FI \cdot E$
Proc. 12. if $(i \neq n_1 - 1) RL_{i+1} \leftarrow T1$
Proc. 13. $RH_i \leftarrow T2$
Proc. 14. if $(i == 0)$ $RL_0 \leftarrow UC$
Proc. 15. next i
Proc. 16. next j

The modular multiplication in process 9-11 corresponds to that in steps 5-7 of Algorithm 3. The data storage in process 12-14 corresponds to that in steps 8-10 of Algorithm 3. Process 9 and 10 represent the w_1-bit$\times w_2$-bit multiplication which is executed by the multiplier shown in Figure 3(a), and process 11 represents the reduction by $f(x)$ which is executed by the multiplier shown in Figure 3(b). Process 10 represents the calculation of the quotient E, done by the left side of the circuit in Figure 3(a) which corresponds to Algorithm 4. If $i = (n_1 - 1)$ in process 10, selector S and demultiplexer D in Figure 3(1) are



(a) Partial Multiplication and Calculating Quotient Part



(b) Reduction Part

Fig. 3. An Example of Our w_1-bit×w_2-bit Multiplier ($w_1 = 8$, $w_2 = 4$)

Table 1. Hardware Size and Critical Path.

Configuration	Hardware size					critical path delay		
	#XOR	#AND	#SEL	#DMUL	#FF	#XOR	#AND	#SEL
Our multiplier	$2w_1w_2$	$2w_1w_2$	w_2	w_2	$3(w_1+w_2)$	w_2+2	w_2	w_2
Laws' multiplier [10]	$2w_1w_2$	$2w_1w_2$	w_2	w_2	$3(w_1+w_2)$	$2w_2+1$	w_2	w_2
Im's multiplier [7]	$2w_1w_2$	$2w_1w_2$	w_2	w_2	$3(w_1+w_2)$	w_2+1	w_2	w_2
SSM [15]	$2w_1$	$2w_1$	–	–	$3w_1+3$	2	1	–

switched to side 1, so that the quotient is stored in register E. If $i \neq n_1 - 1$ in process 10, selector S and demultiplexer D in Figure 3(a) are switched to side 2.

3.3 Performance Analysis

Hardware Size and Performance.
The hardware size and critical path delay of our multiplier are listed in Table 1. As the values of w_1 and w_2 increase, the number of XOR gates and AND gates also increase to be a dominant factor determining hardware size.

The performance of our multiplier is evaluated by the critical path delay and $C(m)$. The critical path delay is proportional to w_2, and $C(m)$ is the number of partial multiplications required for multiplication over $GF(2^m)$ with the w_1-bit$\times w_2$-bit multiplier:

$$C(m) = \lceil m/w_1 \rceil \times \lceil m/w_2 \rceil + \beta,$$

where $\lceil x \rceil$ is the least integer greater than x. We assume that the w_1-bit$\times w_2$-bit multiplier is performed in one cycle, and we ignore the data transfer cycle for pipeline processing in which data is transferred during the multiplication. The variable β is a constant, and it is the sum of the number of cycles for the input and the output.

The number of cycles for multiplication over $GF(2^m)$ is shown in Figure 4 for $(w_1, w_2) = \{(288, 8), (144, 16), (96, 24), (72, 32), (48, 48)\}$. Note that the hardware size of the multipliers is the same for each of these pairs. In our evaluation, the number of processing cycles is the product of the critical path delay and the $C(m)$ based on (288,8). For example, when $m = 288$ and $(w_1, w_2) = (144, 16)$, the $C(m)$ remains unchanged, but the critical path delay is twice as long. Thus the number of processing cycles for (144, 16) is evaluated twice as that for (288, 8).

From Figure 4, it can be seen that processing is faster when w_1 is larger and w_2 is smaller. Theoretically, then the processing is the fastest when $w_2 = 1$. But, the processing is not always the fastest when $w_2 = 1$. This is due to an upper boundary of the processing clock that depends on the hardware characteristics. So we selected $w_2 = 4$ for the FPGA implementation and $w_2 = 8$ for the ASIC implementation to get the best performance.

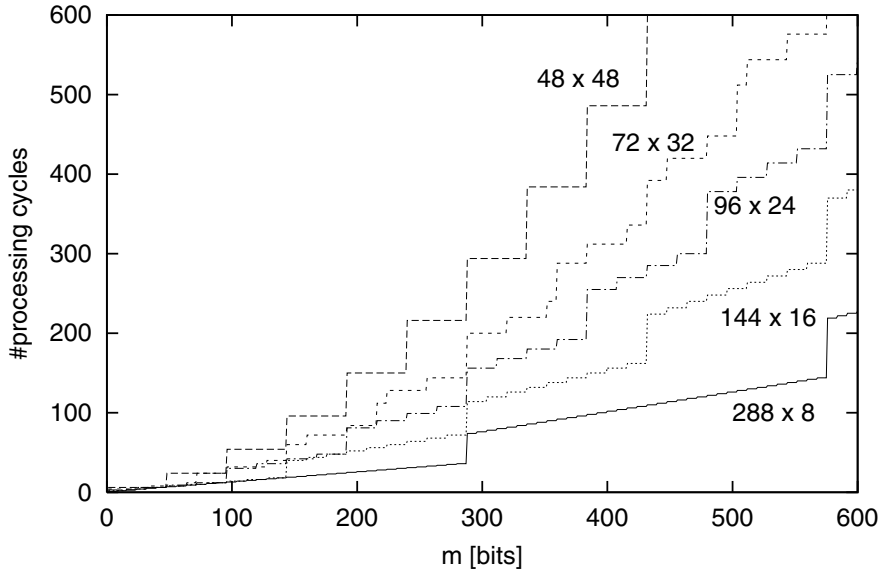


Fig. 4. Processing Cycles for Multiplication

Comparison.
In this section, we compare our multiplier with the three others based on the LFSR [10] [7] [15]. In [10] and [7], the multipliers are bit parallel m-bit$\times m$-bit multipliers based on the LFSR. In [10] polynomial is programmable, while in [7] it is fixed. The SSM is a bit serial multiplier based on the LFSR.

First we compare our multiplier with Im's multiplier and Laws' multiplier. For this comparison we modify Laws' multiplier and Im's multiplier so they can perform w_1-bit$\times w_2$-bit multiplications for arbitrary irreducible polynomials at any bit length. The hardware sizes and critical path delays are listed in Table 1.

The hardware size of our multiplier is the same as that of Laws' multiplier and Im's multiplier, and the difference of critical path delay is negligible in comparison with Im's multiplier the shortest one. If our methods were applied to Im's multiplier, it would improve flexibility and the performance.

Next, we compare our multiplier with the SSM. We evaluate the hardware sizes and critical path delays of the SSM, which are shown in Table 1. We consider the SSM is the special case of our multiplier when $w_2 = 1$. That is, the hardware size of the SSM is the same as that of our multiplier when $w_2 = 1$ and the difference of critical path delay is negligible in comparison with our multiplier and is the same as the Im's multiplier when $w_2 = 1$. We consider our multiplier architecture to be an extension of the concept of the SSM.

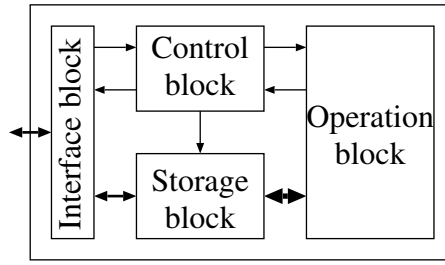


Fig. 5. Basic Diagram of the ECC Coprocessor

4 ECC Coprocessor Implementation

4.1 Basic Design

Figure 5 shows the basic design of the ECC coprocessor with our multiplier. It consists of four function blocks: an interface block, an operation block, a storage block, and a control block. The interface block controls the communications between host and coprocessor. The operation block contains the w_1-bit$\times w_2$-bit multiplier described in section 3. The storage block stores the input, output, and intermediate values. The control block controls the operation block in order to perform elliptic scalar multiplication.

We referred to IEEE P1363 draft [6] for elliptic curve algorithms on a pseudo-random curve, and to FIPS186-2 draft [14] for elliptic curve algorithms on a Koblitz curve [9].

4.2 Implementation

We designed the above coprocessor by using Verilog-HDL, and implemented the ECC coprocessor on an FPGA. The FPGA we used was the EPF10K250AGC599-2 by ALTERA [3]. It operates at 3 MHz. We implemented the coprocessor using an 82-bit×4-bit multiplier. It can do up to 163-bit elliptic scalar multiplication, and the processing times for 163-bit elliptic scalar performance for 163-bit elliptic scalar multiplication with this coprocessor are listed in Table 2.

We also designed and simulated an ECC coprocessor including a 288-bit×8-bit multiplier for up to 572-bit elliptic scalar multiplication. In this simulation we used CE71 series of 0.25 μm ASIC, which is macro-embedded cell arrays by

Table 2. Performance of FPGA Implementation

Length[*bit*]	processing time [msec]	
	Pseudo-random curve	Koblitz curve
163	80.3	45.6

Table 3. Performance of ASIC Implementation (Simulation)

$Length[bit]$	processing time [msec]	
	Pseudo-random curve	Koblitz curve
163	1.1	0.65
233	1.9	1.1
283	3	1.7
409	11	6.6
571	22	13

FUJITSU [4]. Our coprocessor can operate at up to 66 MHz and its hardware size is about 165 Kgates. The processing time for elliptic scalar multiplication on a pseudo-random curves and a Koblitz curves when $m = 163, 233, 283, 409$, and 571 are listed in Table 3. These bit lengths are recommended in FIPS 186-2 draft [14].

5 Conclusion

We described the implementation of an elliptic curve cryptographic coprocessor over $GF(2^m)$ on an FPGA (EPF10K250AGC599-2, ALTERA). This coprocessor is suitable for server systems and enables efficient ECC operations for various parameters.

We proposed a novel multiplier configuration over $GF(2^m)$ that makes ECC calculation faster and more flexible. Its two special characters are that its bit parallel multiplier architecture is an expansion of the concept of the SSM, and that its data conversion method makes possible fast multiplication at any bit length.

The ECC coprocessor implemented with our multiplier on an FPGA performs a fast elliptic scalar multiplication on a pseudo-random curve and on a Koblitz curve. For 163-bit elliptic scalar multiplication, operating at 3 MHz, it takes 80 ms on a pseudo-random curve and 45 ms on a Koblitz curve. We also simulated the operation of this coprocessor implemented as a 0.25 μm ASIC that can operate at 66 MHz and has a hardware size of 165 Kgates. For 163-bit elliptic scalar multiplication, it would take 1.1 ms on a pseudo-random curve and 0.65 ms on a Koblitz curve. And for 571-bit, it would take 22 ms on a pseudo-random curve and 13 ms on a Koblitz curve.

References

1. G.B. Agnew, R.C. Mullin, and S.A. Vanstone, "An implementation of elliptic curve cryptosystems over $GF(2^{155})$," IEEE Journal on Selected Areas in Communications, 11(5), pp. 804-813, 1993.
2. G.B. Agnew, R.C. Mullin, I.M. Onyschuk, and S.A. Vanstone, "An implementation for a fast public-key cryptosystem," Journal of Cryptography, vol.3, pp. 63-79, 1991.

3. Altera, "FLEX 10K Embedded programmable logic Family Data Sheet ver.4.01," 1999. http://www.altera.com/document/ds/dsf10k.pdf
4. Fujitsu, "CMOS Macro embedded type cell array CE71 Series," 2000 http://www.fujitsu.co.jp/hypertext/Products/Device/ CATALOG/AD0000-00001/10e-5b-2.html
5. M.A. Hasan, "Look-up Table Based Large Finite Field Multiplication in Memory Constrained Cryptosystems," IEEE Trans. Comput., vol.49, No.7, July 2000 (to be appear).
6. IEEE, "IEEE P1363/D13(Draft Version 13). Standard Specifications for Public Key Cryptography Annex A (Informative). Number-Theoretic Background," 1999.
7. J.H. Im, "Galois field multiplier," U.S Patent #5502665, 1996.
8. N. Koblitz, "Elliptic curve cryptosystems," Mathematics of Computation 48, pp.203-209, 1987.
9. N. Koblitz, "CM-curve with good cryptographic properties," Advances in Cryptology, Proc. Crypto'91, Springer-Verlag, pp.279-287 1992.
10. B.A. Laws and C.K. Rushforth, "A cellular-array multiplier for $GF(2^m)$," IEEE Trans. Comput., vol.C-20, pp.1573-1578, 1971.
11. E.D. Mastrovito, "VLSI design for multiplication over finite fields $GF(2^m)$," In Lecture Notes in Computer Science 357, pp.297-309. Springer-Verlag, 1989.
12. A.J. Menezes, "Elliptic Curve Public Key Cryptosystems," Kluwer Academic Publishers, 1993.
13. V.S. Miller, "Use of elliptic curve curves in cryptography," Advances in Cryptology, Proc.Crypto'85, Springer-Verlag, pp.417-426, 1986.
14. NIST, "FIPS 186-2 draft, Digital Signature Standard (DSS)," 2000. http://csrc.nist.gov/fips/fips186-2.pdf
15. G. Orlando and C. Paar, "A Super-Serial Galois Fields Multiplier for FPGAs and its Application to Public-Key Algorithms," in Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM '99, April 1999.
16. M. Rosner, "Elliptic Curve Cryptosystems on Reconfigurable Hardware," Master's Thesis, Worcester Polytechnic Institute, 1998. http://www.ece.wpi.edu/Research/crypt/theses/index.html

A High-Performance Reconfigurable Elliptic Curve Processor for $GF(2^m)^\star$

Gerardo Orlando[1] and Christof Paar[2]

[1] General Dynamics Communication Systems
77 A St., Needham MA 02494-2892, USA
gerardo.orlando@gd-cs.com
[2] ECE Department, Worcester Polytechnic Institute
100 Institute Road, Worcester, MA 01609, USA
christof@ece.wpi.edu

Abstract. This work proposes a processor architecture for elliptic curves cryptosystems over fields $GF(2^m)$. This is a scalable architecture in terms of area and speed that exploits the abilities of reconfigurable hardware to deliver optimized circuitry for different elliptic curves and finite fields. The main features of this architecture are the use of an optimized bit-parallel squarer, a digit-serial multiplier, and two programmable processors. Through reconfiguration, the squarer and the multiplier architectures can be optimized for any field order or field polynomial. The multiplier performance can also be scaled according to system's needs. Our results show that implementations of this architecture executing the projective coordinates version of the Montgomery scalar multiplication algorithm can compute elliptic curve scalar multiplications with arbitrary points in 0.21 msec in the field $GF(2^{167})$. A result that is at least 19 times faster than documented hardware implementations and at least 37 times faster than documented software implementations.

1 Introduction

This work proposes a scalable elliptic curve processor architecture (ECP) which operates over finite fields $GF(2^m)$. One of its key features is its suitability for reconfigurable hardware. Unlike traditional VLSI hardware, reconfigurable devices such as Field Programmable Gate Arrays (FPGA) do not possess fixed functionality after fabrication but can be reprogrammed during operation. The scalability of the ECP architecture and the flexibility of reconfigurable hardware afford implementations the following benefits:

Architecture Efficiency. The complexity of finite field arithmetic architectures depends greatly on whether arithmetic for one specific field is being implemented, or for general finite fields. The most dramatic example is perhaps squaring in $GF(2^m)$ using standard basis. For a specific field, squaring can be performed in one clock cycle, whereas a general architecture usually

* This research was supported in part by NFS CAREER award CCR-9733246.

Ç.K. Koç and C. Paar (Eds.): CHES 2000, LNCS 1965, pp. 41–56, 2000.

requires $m/2$ clock cycles (where $m \geq 160$ for elliptic curves cryptosystems) [BG89]. Consequently, one algorithmic option that we explore in this paper relies on the bit-parallel computation of squares, resulting in extremely efficient implementations. The use of reconfigurable hardware allows applications to use an optimized squarer for every finite field.

Scalability. Depending on the application, different levels of security may be required. The main factor that determines the security of elliptic curve cryptosystem is the size of the underlying finite field. For instance, NIST announced recently a list of curves ranging from 163–571 bits [NIS99]. Realizing such a wide operand range efficiently in traditional hardware is a major challenge, whereas the ECP's architectural scalability and the FPGAs reconfigurability allow optimized processor instantiations for any field size. Moreover, the fine-grained scalability of the ECP's architecture provides a wide range of time-area, performance-cost architectural options. Section 5 provides some examples.

Algorithm Agility. It is a design paradigm of modern security protocols that cryptographic algorithms can be negotiated on a per-session basis. With the proposed ECP, it is possible through reconfiguration to (1) switch algorithm parameters and (2) to switch to another type of public-key algorithm.

Resource Efficiency. The vast majority of security protocols use public-key algorithms in the beginning of a session for tasks such as entity authentication and key establishment and private-key algorithms for bulk data encryption after that. With reconfigurable platforms, it is possible to reuse the same device for both tasks.

The remainder of the paper is structured as follows. Section 2 summarizes the previous works on elliptic curve implementations. Section 3 provides the most crucial mathematical and algorithmic background needed to understand the ECP. Section 4 describes the ECP architecture and its main components. Section 5 describes prototype implementations and results. Section 6 summarizes the conclusions.

2 Previous Work

A number of software and hardware implementations have been documented for the computation of point multiplication, which is the basic operation used by elliptic curve cryptographic systems. Among the most significant hardware implementations are [AMV93,Ros98,GSS99,SES98]. The ones in [AMV93,SES98] use off-the-shelf processors to perform elliptic curve operations and accelerators to perform finite field arithmetic. The implementation in [AMV93] uses an ASIC accelerator and the one in [SES98] uses an FPGA accelerator. The implementations in [Ros98,GSS99] are standalone elliptic curve processors in FPGAs. Both [Ros98,GSS99] define roadmaps for full-size, secure elliptic curve implementations but do not document successful implementations of them.

The implementations in [AMV93,GSS99,SES98] use normal basis representation. They use bit-serial multipliers, which require about m clock cycles to

compute a multiplication in $GF(2^m)$ and compute squares with cyclic shifts. (The use of digit-serial multipliers, which are used in this work, is mentioned in [GSS99] but the documented implementations use bit-serial multipliers.)

The hardware implementation documented in [Ros98] uses standard basis representation. This implementation is suitable for composite fields $GF((2^u)^v)$ where $u * v = m$. Its core-processing element is a hybrid multiplier which computes a multiplication in v clock cycles. This multiplier is also used to compute squares. It should be pointed out that recent developments demonstrate that some forms of composite fields give rise to elliptic curves that posses cryptographic weaknesses [GHS00].

Among the best performing software implementations which are reported in open literature are [SOOS95,LD99]. The performance of these implementations, as demonstrated in Section 5, rival that of the traditional hardware implementations previously mentioned. The main reasons for their high performance are their use of very efficient algorithms that are optimized for modern processors and the availability of processors with wide words that operate at very high clock rates.

The elliptic curve processor architecture introduced in this work exhibits the features of the aforementioned hardware and software implementations. Its hardware architecture is scalable and its processing units, like the ones used by the software implementations, are programmable. In addition, its architecture is neither restricted to use polynomials on extension degrees of a special form, as is the case for [Ros98], nor it favors particular fields, as is the case for [AMV93,GSS99,SES98] that favor fields for which Gaussian normal bases exist. It is also, to the authors' knowledge, the only standalone elliptic curve processor architecture that has been rendered into a full-size, secure elliptic curve implementation in FPGA technology.

3 Mathematical Background

3.1 Elliptic Curves Algorithms and Choice of Field Representation

This section provides a brief description of the elliptic curve algorithms used by the elliptic curve processor (ECP). The first algorithm is the double-and-add algorithm for scalar multiplications using projective coordinates as defined in [P1398]. The other algorithm is the projective coordinates version of the Montgomery scalar multiplication method described in [LD99]. The distinctive characteristics of these two algorithms are that the double-and-add algorithm adds and doubles elliptic curve points, while the Montgomery method adds and doubles only the x coordinates of two points, P_1 and P_2, where $P_2 = P_1 + P$ and P is the point that is being multiplied. Since the relationship between P_1 and P_2 is maintained throughout the multiplication, the addition of P_1 and P_2 yields the point $2P_1 + P$. From this detail and Algorithm 2 in the Appendix, one can verify that the intermediate points P_1 obtained during the computation of kP correspond to the intermediate points obtained with the double-and-add algorithm. At the end of the multiplication process, the x coordinate of kP is given

by the x coordinate of P_1 and the y coordinate is derived from the x coordinates of P_1 and P_2 and from the coordinates of P. The two multiplication methods previously discussed are presented in Algorithm 1 and 2 in the Appendix. Note that these algorithms, as the rest of this document, assume that the elliptic curve equation is defined as $y^2 + xy = x^3 + ax^2 + b$. These algorithms also assume that the binary representation of k is given by $k = \sum_{i=0}^{l-1} k_i 2^i$ with $k_{l-1} \neq 0$. The computational complexity of these algorithms is summarized in Table 1.

Table 1. Complexity of point multiplication in $GF(2^m)$ $(a, b \neq 0)$

Complexity	Montgomery	Double-and-Add (average)
#Squares	$5(m-1)+3$	$7(m-1)+1$
#Mult.	$6(m-1)+10$	$10.5(m-1)+3$
#Inverses	1	1

From Table 1 it is clear that an efficient method for squaring will have a considerable impact on the overall performance. Through the use of reconfigurable hardware it is possible to compute a square in one clock cycle for any field order even though a standard basis representation is being used. It appears very difficult to achieve the same behavior with traditional ASIC hardware platforms. An alternative is a normal basis representation, but this comes at the cost of a more complex multiplication architecture. In particular, normal basis multipliers can be prohibitively expensive for fields for which optimum normal bases do not exist. For an ECP with flexible finite field support, normal basis representation appear not to be the best choice.

It is important to note that the point multiplication algorithms consist of a main function, the double_and_add or the montgomery_scalar_multiplication functions in the algorithms shown in the Appendix. These main functions call point addition, point multiplication, coordinate conversion, and other functions as subroutines. In turn, these subroutines call finite field arithmetic subroutines. This hierarchical view is helpful for understanding the processor architecture described in Section 4.

3.2 $GF(2^m)$ Field Arithmetic

This section provides a brief introduction to $GF(2^m)$ finite field arithmetic. The reader is referred to [LN94] for in-depth study of this topic.

For all practical purposes, the computation of elliptic curve point double and a point addition is realized with algorithms involving field additions, squares, multiplications, and inversions. This work considers arithmetic in fields of characteristic two, $GF(2^m)$, using a standard basis representation. This basis representation is also known as polynomial or canonical basis representations. A field $GF(2^m)$ is isomorphic to $GF(2)[x]/(F(x))$, where $F(x) = x^m + \sum_{i=0}^{t} f_i x^i$ is a

monic irreducible polynomial of degree m with coefficients $f_i \in \{0, 1\}$. Here each residue class is represented by the polynomial of least degree in its class.

A standard basis representation uses the basis defined by the set of elements $\{1, \alpha, \alpha^2, \ldots, \alpha^{m-1}\}$, where α is a root of the irreducible polynomial $F(x)$. In this basis, field elements are represented as polynomials in α of degree less than m with coefficients 0 or 1; for example, an element A is represented as $A = \sum_{i=0}^{m-1} a_i \alpha^i$ with coefficients $a_i \in \{0, 1\}$. In hardware, the field elements are represented by binary m-tuples as in $(a_{m-1}, a_{m-2}, \ldots a_0)$.

The addition of two elements requires the modulo 2 addition of the coefficients of the field elements. In hardware, a bit-parallel adder requires m XOR gates, and an addition can be generally computed in one clock cycle.

The squaring of a field element $A = \sum_{i=0}^{m-1} a_i \alpha^i$ is ruled by Equation (1). A bit-parallel realization of this squarer requires at most $(r-1)(m-1)$ gates [Wu99,PFSR99], where r represents the number of non-zero coefficients of the field polynomial.

$$A^2 \equiv \sum_{i=0}^{m-1} a_i \alpha^{2i} \bmod F(\alpha) \tag{1}$$

The multiplication of two field elements A and B can be expressed as shown in Equation (2). This equation is arranged so that it facilitates the understanding of the digit-serial multiplier used by the ECP. This multiplier is of the type introduced in [SP97], and it is described here in Section 4.

In Equation (2), B is expressed in k_D digits ($1 \leq k_D \leq \lceil m/D \rceil$) as follows: $B = \sum_{i=0}^{k_D-1} B_i \alpha^{Di}$, where $B_i = \sum_{j=0}^{D-1} b_{Di+j} \alpha^j$ and D is the digit size in bits. Note that when m/D is not an integer, B is extended to an integer number of digits ($k_D = \lceil m/D \rceil$) by setting its most significant coefficients to 0 ($b_m = b_{m+1} = \ldots = b_{k_D * D-1} = 0$).

$$\begin{aligned} AB &\equiv (A \sum_{i=0}^{k_D-1} B_i \alpha^{Di}) \bmod F(\alpha) \\ &\equiv (\sum_{i=0}^{k_D-1} B_i (A\alpha^{Di} \bmod F(\alpha))) \bmod F(\alpha) \end{aligned} \tag{2}$$

The ECP lacks inversion circuitry. This work recommends the computation of inversions with repeated multiplications using the algorithms described in [IT88,Van99]. These algorithms compute inverses with $\lfloor \log_2(m-1) \rfloor + W(m-1) - 1$ multiplications [BSS99], where $W(m-1)$ represents the number of non-zero coefficients in the binary representation of $m-1$.

4 Processor Architecture

To compute kP efficiently one needs a blend of efficient algorithms and hardware architectures. Efficient algorithms are needed to compute point multiplication

and field operations. One also needs a platform that supports the efficient computation of such algorithms. This work proposes a processor architecture optimized for the use of efficient elliptic curve algorithms, which is also well suited for implementations in reconfigurable hardware.

The elliptic curve processor (ECP), shown in Figure 1, consists of three main components. These components are the main controller (MC), the arithmetic unit controller (AUC), and the arithmetic unit (AU). The MC is the ECP's main controller. It orchestrates the computation of kP and interacts with the host system. The AUC controls the AU. It orchestrates the computation of point additions, point doublings, and coordinate conversions. The AU performs the $GF(2^m)$ field additions, squares, multiplications, and inversions under AUC control. For the point multiplication algorithms given in the Appendix, the MC executes the double_and_add and the montgomery_scalar_multiplication functions, the AUC performs all the other subroutines, and the AU is the hardware that computes the finite field operations.

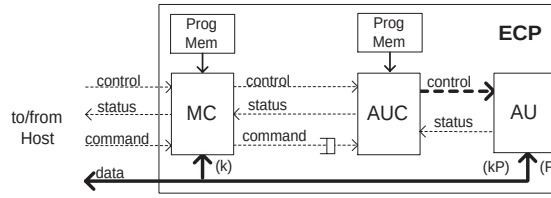


Fig. 1. Elliptic curve processor architecture

The following is a typical sequence of steps for the computation of kP in the ECP using the double-and-add algorithm and projective coordinates. First, the host loads k into the MC, loads the coordinates of P into the AU, and commands the MC to start processing. Then, the MC does its initialization, which includes finding the most significant non-zero coefficient of k. The MC then commands the AUC to perform its initialization, which includes the conversion of P from affine to projective coordinates. During the computation of kP, the MC scans one bit of k at time starting with the second most significant coefficient and ending with the least significant one. In each of these iterations, the MC commands the AU/AUC to do a point double. If the scanned bit is a 1, it also commands the AU/AUC to do a point addition. For each of these point operations, the AUC generates the control sequence that guides the AU through the computation of the required field operations. After the least significant bit of k is processed, the MC commands the AU/AUC to convert the result back to affine coordinates. When the AU/AUC finishes this operation, the MC signals to the host the completion of the kP operation. Finally, the host reads the coordinates of kP from the AU.

The ECP incorporates a set of techniques that maximizes resource utilization and speed. The most evident feature is concurrency. The ECP uses two loosely

coupled controllers, the MC and the AUC controllers, that execute their respective operations concurrently. These are very simple processors that execute one instruction per clock cycle. The AU uses concurrency. The AU incorporates a multiplier, a squarer, and a register file, all of which can operate in parallel on different data.

Another technique is pipelining. The regular architecture of the ECP allows it to use pipeline stages to reduce the critical path delay of the hardware and thus increase its operational frequency. The ECP incorporates pipelining in the AU and assures its maximum utilization with the AUC. The AUC maximizes pipeline utilization by minimizing pipeline fills and flushes. For example, the AUC can start loading operands for the next multiplication before the current one finishes.

The last main technique is the use of a large register set. The ECP's large register set supports algorithms that rely on precomputations. There are many such algorithms. Here we consider two examples. An example is the fixed window point multiplication algorithm. This algorithm requires on average $m + 2^{w-1}$ point doubles, $\lfloor m/w \rfloor + 2^{w-1}$ point additions, and the storage of 2^w points. Another algorithm is an adaptation of a fixed base exponentiation method introduced in [BGMW93] for operations involving a fixed point. This algorithm requires on average $\lfloor m/w \rfloor + 2^w$ point additions, the storage of $\lceil m/w \rceil$ points, and no point doubles. In the previous expressions, w is the window size, which is a measure of the number of bits of k processed in parallel. It must be pointed out that these optimizations can be used with the projective coordinate equations for point double and point addition defined in [P1398] but not with the ones defined in [LD99]. As this later algorithm requires that the relationship $P2 = P1 + P$ be maintained throughout the point multiplication process, while the aforementioned optimizations rely on precomputing absolute multiples of a point; for example, $1P$, $2P$, ... , $(2^w - 1)P$.

To illustrate the benefits of precomputation, consider an implementation for $GF(2^{167})$ using the projective coordinates defined in [P1398] and $w = 4$. Compared to the traditional double-and-add algorithm, the fixed window algorithm is approximately 1.1 times faster and the fixed point algorithm is over 2.5 times faster.

4.1 Arithmetic Unit

The AU, shown in Figure 2, is the unit responsible for field arithmetic. It consists of a register file, a least significant digit first (LSD) multiplier, a squarer, an accumulator, and a zero test circuit. The AU arranges these components in a streamlined, pipelined configuration that exhibits low fan out. The architecture contains two feedback paths that allow fast availability of operands to the multiplier, the squarer, and the register file.

The AU components operate under AUC control. The AUC's control extends to all the components shown in Figure 2. This fine control allows the AUC to extract maximum throughput from the AU by paralleling functions and managing pipeline delays.

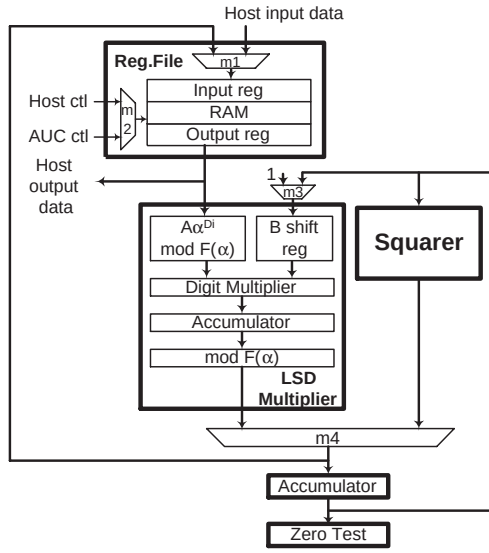


Fig. 2. Arithmetic unit

The multiplier and the squarer support the computation of field additions, squares, multiplications, and inversions. The addition of A and B is done by first computing $A * 1$ and then adding to it the product $B * 1$. The 1 operand can be supplied by the multiplexer m3 or the register file. This addition method exploits the ability of the LSD multiplier to accumulate products and eliminates the need for an adder. Field inversions are computed with repeated multiplications using the inversion algorithms described in [IT88,Van99].

The register file stores operands, precomputed values, and temporary values. It accepts input operands, such as the coordinates of P and the elliptic curve parameters a and b from the host system. It also accepts the results from the multiplier or the squarer selected by the multiplexer m4. It outputs operands to the multiplier and results to the host system. The basic components of the register file are the input and output registers and the RAM memory. RAM memory supports a large number of registers and the input and output register resolve access contentions to it.

The accumulator stores results from the multiplier and the squarer. It supplies the input operand of the squarer and one of the input operands of the multiplier. The zero test circuit, upon command, samples the content of the accumulator and compares it with zero. It maintains its result until another test is issued.

The AU employs a bit-parallel squarer [Wu99,PFSR99]. In the ECP's architecture, this squarer is capable of computing a square in one clock cycle. This squarer is a rendition of Equation (1) using XOR gates. For the field polynomials recommended for cryptographic applications [P1398,ANS98,ANS99], the

squarer complexity is at most $(m+t+1)/2$ gates for irreducible trinomials $F(x) = x^m + x^t + 1$ and $4(m-1)$ gates for pentanomials $F(x) = x^m + x^{t1} + x^{t2} + x^{t3} + 1$ [Wu99]. Moreover, for trinomials the critical path delay is at most two gate delays [Wu99].

The AU uses an LSD multiplier of the type introduced in [SP97]. This semi-systolic multiplier computes products according to Equation (2) using Algorithm 3. This multiplier computes a product sum $AB + C \bmod F(\alpha)$ within $\lceil m/D \rceil$ clock cycles. More precisely, the product is computed in k_D clock cycles, where k_D $(1 \leq k_D \leq \lceil m/D \rceil)$ represents the number of digits of B. The performance and consequently complexity of this multiplier is a function of the digit size D [SP97].

Algorithm 3: LSD multiplication

Inputs: $A = \sum_{i=0}^{m-1} a_i \alpha^i$ $B = \sum_{i=0}^{k_D-1} B_i \alpha^{Di}$, where $B_i = \sum_{j=0}^{D-1} b_{Di+j} \alpha^j$ Output: $C = (AB + C) \mod F(\alpha)$

$C = 0$ or the previous value of C
for $i = 0$ to $k_D - 1$ do
$C = B_i(A\alpha^{Di} \mod F(\alpha)) + C$
end for
$C = C \mod F(\alpha)$

As previously described, the ECP takes advantage of the accumulation property of its multiplier to compute additions. The addition $A+B$ requires two clock cycles when it is necessary to compute $A*1$ and then add to it the product $B*1$. It requires only one clock cycle when adding to the result of the previously computed multiplication or addition. In this last case one of the operands is already in the multiplier's accumulator.

A block diagram of the LSD multiplier is included in Figure 2 along with the other components of the AU. Its components are the B shift register, the $A\alpha^{Di} \mod F(\alpha)$, the digit multiplier, the accumulator, and the mod $F(\alpha)$ circuits. The B shift register delivers one digit of the B operand in each clock cycle. The $A\alpha^{Di} \mod F(\alpha)$ circuit computes an element $A\alpha^{Di} \mod F(\alpha)$ in each clock cycle from A for $i = 1$ or from the previously computed $A\alpha^{D(i-1)} \mod F(\alpha)$ for $i = 2, \ldots, k_D - 1$. The digit multiplier computes a product $B_i(A\alpha^{Di} \mod F(\alpha))$ in each clock cycle and the accumulator adds it to the cumulative sum of the previously computed products. The accumulated result is reduced by the mod $F(\alpha)$ circuit. The architecture of the multiplier is regular with only the reduction operations (mod $F(\alpha)$) dependent on the field polynomials.

The complexity of this multiplier, assuming no pipelining of the digit multiplier circuit, is approximately $2Dm + 7m$ gates and $3m$ registers for $m >> D$. The digit multiplier circuit is a main contributor to the complexity and performance of the multiplier. Its gate complexity is proportional to the digit size,

$2Dm$ gates, and, when it is implemented with binary trees, its critical path delay is $\lceil \log_2 2D \rceil$ gate delays.

Note that all the estimates given in this section assume 2-input gates, account for system I/O, and assume optimum field polynomials according to the definition given in [SP97]. These are field polynomials $F(x) = x^m + \sum_{i=0}^{t} f_i x^i$ for which $m - t \geq D$. Over 99% of the field polynomials in [P1398,ANS98,ANS99] satisfy this condition for digit sizes up to $D = 50$ and fields in the range $160 \leq m \leq 1024$.

5 Prototype Implementations

Three ECP prototypes were built to verify the suitability of the ECP architecture for reconfigurable FPGA logic. These prototypes support elliptic curves over the field $GF(2^{167})$, which is an attractive field for secure cryptosystems, with this field being defined by the field polynomial $F(x) = x^{167} + x^6 + 1$. However, we would like to stress that the ECP can be reconfigured with optimized architectures for any field $GF(2^m)$.

Each prototype used a 16-bit MC processor with 256 words of program memory, a 24-bit AUC processor with 512 words of program memory, and 128 registers, each of which is 167 bits wide. They also provided 32-bit I/O interface to the host system. To verify the scalability of the ECP architecture, each of the prototypes used an LSD multiplier with a different digit size. The prototypes used LSD multipliers with digit sizes equal to 4, 8, and 16. To verify the ECP's ability to handle multiple algorithms, the operation of the prototypes was verified with the two elliptic curve algorithms described in the Appendix. The implementation of these two algorithms demonstrates the ability of the ECP to adopt new, highly efficient algorithms. For example, an ECP can be deployed with one algorithm today and then updated with a better algorithm in the future.

The prototypes were implemented using the Xilinix's XCV400E-8-BG432 (Virtex E) FPGA. The prototypes were coded in VHDL. They were synthesized with Synopsis' FPGA Express 3.0 and Xilinx's Design Manager M2.1i. The details of these prototype implementations are discussed in the following subsections.

5.1 ECP Algorithms and Programming

The ECP prototypes were tested with two programs. One of the programs implemented the projective coordinates version of the Montgomery scalar multiplication algorithm and the other the projective coordinates version of the traditional double-and-add algorithm, none of which relies on precomputations. It should be noted that use of algorithms that rely on precomputation is supported by the ECP and their use will typically result in faster implementations than the ones documented here.

The number of clock cycles required to compute kP for each of the programs is summarized in Table 2. Because each step of the Montgomery algorithm requires the computation of a point addition and a point double, this table groups

these two operations in a single row. For the double-and-add algorithm, independent rows for point addition and point double are provided because each step of the algorithm requires a point double but not necessarily a point addition.

Note that the entries in Table 2 contain terms multiplied by $\lceil 167/D \rceil$, where D is the digit size of the multiplier being used. These terms reflect the number of $GF(2^{167})$ multiplications, each of which is executed in $\lceil 167/D \rceil$ clock cycles. The constant terms in the table account for squares, additions and processing overhead. Each square is computed in one clock cycle. Each additions is computed in one clock cycle if one of the operands is already in the multiplier's accumulator or in two clock cycles if that is not the case. The overhead processing time varies with each operation and it is accounted for each operation in the table. The times for coordinate conversions includes the computation of inverses using the inversion algorithm described in [Van99].

For both elliptic curve algorithms, the MC program used 56% of the MC's program memory. The AUC program used 90–98% of the AUC's program memory depending on the algorithm and the digit size. The high AUC memory utilization is due to the in-line coding of the point double and point add functions, which are by far the most frequently used operations. This is evident from the low overhead reported in Table 2 for these functions. To conserve memory, in-line coding was not used for infrequently executed functions such as coordinate conversion. Consequently, these operations exhibit high overhead.

Table 2. Number of clock cycles required to compute kP over $GF(2^{167})$

Operation	Double-and-Add # Clock Cycles	Montgomery # Clock Cycles
Point Double	$5\lceil 167/D \rceil + 25$	$6\lceil 167/D \rceil + 17$
Point Add	$11\lceil 167/D \rceil + 31$	
Coor. Conv., etc.,	$13\lceil 167/D \rceil + 575$	$20\lceil 167/D \rceil + 764$
kP	$(10.5\lceil 167/D \rceil + 47.5) * 166 + 13\lceil 167/D \rceil + 575$	$(6\lceil 167/D \rceil + 24) * 166 + 20\lceil 167/D \rceil + 764$

Table 3 approximates the number of cycles required for the computation of point multiplication for arbitrary $GF(2^m)$ fields. The approximations are based exclusively on the number of multiplications and the number of clock cycles required to compute them with an LSD multiplier with digit size D. This table assumes that inverses are computed using one of the algorithms defined in [IT88,Van99]. The inversion is assumed to require $\lfloor \log_2(m-1) \rfloor + W(m-1) - 1$ multiplications [BSS99], where $W(m-1)$ represents the number of non-zero coefficients in the binary representation of $m-1$.

Table 3. Number of clock cycles required to compute kP over $GF(2^m)$

Operation	Double-and-Add # Clock Cycles	Montgomery # Clock Cycles
Point Double	$5\lceil m/D \rceil$	$6\lceil m/D \rceil$
Point Add	$11\lceil m/D \rceil$	
Coor. Conv.	$(3 + (\lfloor \log_2(m-1) \rfloor + W(m-1) - 1))\lceil m/D \rceil$	$(10 + (\lfloor \log_2(m-1) \rfloor + W(m-1) - 1))\lceil m/D \rceil$
kP	$(10.5(m-1) + 3 + (\lfloor \log_2(m-1) \rfloor + W(m-1) - 1))\lceil m/D \rceil$	$(6(m-1) + 10 + (\lfloor \log_2(m-1) \rfloor + W(m-1) - 1))\lceil m/D \rceil$

5.2 Performance and Comparisons

This section summarizes the performance of the ECP prototype implementations and shows how it compares against leading software and hardware implementations.

Table 4 summarizes the performance of the ECP prototypes for the two elliptic curve algorithms. The results in this table illustrate that the Montgomery method is about 1.7 times faster than the traditional double-and-add algorithm. One can deduce from Table 1 that this is a direct result of the number of multiplications required by each algorithm ($\approx 10.5/6$), as the processing time for additions, squares, and inversions is almost negligible.

Table 4 also shows that the speedup increases as the digit size increases. The increase is not proportional to the digit size. What happens is that as the digit size increases, the multiplication processing time decreases proportionally. Consequently, the additions, the squarings, and the overhead processing costs increase relative to that of multiplications. Another contributing factor is the modest reduction in clock rate as the digit size increases and thus the size of the ECP. For the prototypes, an appreciable reduction in clock rate occurs as the digit size increased from 4 to 8. The clock rate remained fairly constant as the digit size increased from 8 to 16.

Table 4. Point multiplication performance of ECP prototypes

Digit Size	Clock (MHz)	Montgomery (msec)	double-and-add (msec)	Speedup rel. to $D = 4$
4	85.7	0.55	0.96	1
8	74.5	0.35	0.61	1.8
16	76.7	0.21	0.36	3.0

Table 5 lists the performance of leading published software (SW) and hardware (HW) implementations along with that of the fastest ECP prototype im-

plementation. The data in this table correspond to k values whose binary representation contains roughly the same number of 1's and 0's. Table 5 shows that the performance of software implementations on platforms with wide words and high clock rates rival that of traditional hardware implementations. It also shows that the performance of the fastest ECP implementation is at least 19 times faster than that of traditional hardware implementations and 37 times faster than software implementations.

Table 5. Performance of leading software and hardware implementations

Implementation	SW/ HW	Fields	Platform	Point Mult. (msecs)	Speedup rel. to ECP $D = 16$
Montgomery [LD99]	SW	$GF(2^{163})$	UltraSparc 64-bit,300MHz	**13.5**	64
Almost Inv. [SOOS95]	SW	$GF(2^{155})$	DEC Alpha 64-bit,175MHz	**7.8**	37
ASIC Coprocessor [AMV93]	HW	$GF(2^{155})$	VLSI 40 MHz	**3.9** est.	19
FPGA Coprocessor [SES98]	HW	$GF(2^{155})$	Xilinx FPGA XC4020XL,15 MHz	**18.4** est.	88
Composite fields [Ros98]	HW	$GF(((2^4)^2)^{21})$ $GF((2^8)^{21})$	Xilinx FGPA XC4062,16MHz	**4.5** est.	21
ECP $D = 16$	HW	$GF(2^{167})$	Xilinx FPGA XCV400E,76.7MHz	**0.21**	1

5.3 Logic Complexity

The logic complexity of the ECP prototypes is summarized in Table 6 in terms of the main components of modern FPGAs. These components are lookup tables (LUT) which are used as programmable gates, flip-flops (FF), and Block RAM which are configurable 4k-bit RAMs [Xil99]. The normalized complexity of the ECP prototypes is approximately $228+6.6m+(\lceil 2D/3 \rceil -1)m$ LUTs, $224+9.2m$ FF, and $4+\lceil m/32 \rceil$ 4k-bit Block RAMs for $m >> D$, 4-input LUTs, 32-bit Block RAMs, and D a multiple of 4. Note that the complexity is a function of the digit size D, which as mentioned previously is the main parameter that defines the performance and complexity of the ECP, and the size of the finite field (m). Interestingly, of all the logic elements only LUT logic complexity varies largely as a function of D. The multiplier's digit multiplier circuit is responsible for this variability as its size varies proportionally with the digit size.

The prototype implementations used between 15% and 28% of the LUTs (depending on the digit size), 16% of the FFs, and 25% of the Block RAMs available in the XCV400E-8-BG432 FPGA. Together, the AUC and the MC

Table 6. Logic complexity of ECP prototypes

Digit Size	#LUT	#FF	# Block RAM
4	1627	1745	10
8	2136	1753	10
16	3002	1769	10

processors, ignoring the complexity of the register that holds the k operand, used less than 13% of the logic resources and 40% of the memory elements. In turn, the AU used 76–87% of the LUTs, 59% of the flip-flops, and 60% of the memory elements. The remaining resources were used by system I/O logic. This breakdown shows that the ECP prototype implementations devoted most of its resources to arithmetic processing.

6 Conclusions

This work introduced a new elliptic curve processor architecture. This is a scalable and programmable processor architecture that exploits reconfigurability to deliver optimized solutions for different elliptic curves and finite fields. The ECP architecture is characterized by two loosely coupled processors responsible for the algorithmic functions of point multiplication and by a streamlined, pipelined finite field arithmetic unit that can be optimized for each finite field.

This work demonstrated that the ECP can attain high processing speeds in FPGA logic with three prototype implementations. The fastest prototype implementation was capable of computing a point multiplication in the field $GF(2^{167})$ at least 19 times faster than documented hardware implementations and 37 times faster than documented software implementations. Moreover, because the ECP is programmable as well as configurable, these prototype implementations can be programmed to use future, more efficient elliptic curve algorithms, and their size and performance can be tailored, through reconfiguration, to meet future needs.

References

AMV93. G.B. Agnew, R.C. Mullin, and S.A. Vanstone. An implementation of elliptic curve cryptosystems over $F_{2^{155}}$. *IEEE Journal on Selected Areas in Communications*, 11(5):804–813, June 1993.

ANS98. ANSI X9.62-1999. Public Key Cryptography For The Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA), January 1998. Approved January 7, 1999.

ANS99. ANSI X9.63-1999. Public Key Cryptography For The Financial Services Industry: Key Agreement and Key Transport Using Elliptic Curve Cryptography, January 1999. Working Draft.

BG89. T. Beth and D. Gollmann. Algorithm engineering for public key algorithms. *IEEE Journal on Selected Areas in Communications*, 7(4):458–466, 1989.

BGMW93. E.F. Brickell, D.M. Gordon, K.S. McCurley, and D.B. Wilson. Fast exponentiation with precomputation. In *Lecture Notes in Computer Science 658: Advances in Cryptology — EUROCRYPT '92*, pages 200 – 207. Springer-Verlag, Berlin, 1993.

BSS99. I. Blake, G. Seroussi, and N.P. Smart. *Elliptic Curves in Cryptography*. Cambridge University Press, Cambridge, UK, first edition, 1999.

GHS00. P. Gaundry, F. Hess, and N.P. Smart. Constructive and destructive facets of Weil descent on elliptic curves. available at `http://www.hpl.hp.com/techreports/2000/HPL-2000-10.html`, January 2000.

GSS99. L. Gao, S. Shrivastava, and G. Sobelman. Elliptic curve scalar multiplier design using FPGAs. In C. Koc and C. Paar, editors, *Workshop on Cryptographic Hardware and Embedded Systems (CHES '99)*, volume LNCS 1717. Springer-Verlag, August 1999.

IT88. T. Itoh and S. Tsujii. A fast algorithm for computing multiplicative inverses in $GF(2^m)$ using normal bases. *Information and Computation*, 78:171–177, 1988.

LD99. J. Lopez and R. Dahab. Fast multiplication on elliptic curves over $GF(2^m)$ without precomputation. In C. Koc and C. Paar, editors, *Workshop on Cryptographic Hardware and Embedded Systems (CHES '99)*, volume LNCS 1717. Springer-Verlag, August 1999.

LN94. R. Lidl and H. Niederreiter. *Introduction to finite fields and their applications*. Cambridge University Press, Cambridge, UK, revised edition, 1994.

NIS99. NIST. Recommended elliptic curves for federal government use. available at `http://csrc.nist.gov/encryption`, May 1999.

P1398. P1363. *Standard Specifications for Public-key Cryptography (Draft Version 8)*. IEEE, October 1998.

PFSR99. C. Paar, P. Fleischmann, and P. Soria-Rodriguez. Fast arithmetic for public-key algorithms in Galois fields with composite exponents. *IEEE Transactions on Computers*, 48(10):1025–1034, October 1999.

Ros98. M. Rosner. Elliptic curve cryptosystems on reconfigurable hardware. Master's thesis, ECE Dept., Worcester Polytechnic Institute, Worcester, USA, May 1998.

SES98. S. Sutikno, R. Effendi, and A. Surya. Design and implementation of arithmetic processor $F_{2^{155}}$ for elliptic curve cryptosystems. In *The 1998 IEEE Asia-Pacific Conference on Circuits and Systems*, pages 647–650, November 1998.

SOOS95. R. Schroeppel, H. Orman, S. O'Malley, and O. Spatscheck. Fast key exchange with elliptic curve systems. In D. Coppersmith, editor, *Advances in Cryptogaphy, Crypto 95*, volume LNCS 963. Springer-Verlag, 1995.

SP97. L. Song and K. K. Parhi. Low-energy digit-serial/parallel finite field multipliers. *Journal of VLSI Signal Processing Systems*, 2(22):1–17, 1997.

Van99. S.A. Vanstone. Efficient implementation of elliptic curve cryptography, June 1999. Certicom Corporation Seminar.

Wu99. H. Wu. Low complexity bit-parallel finite field arithmetic using polynomial basis. In C. Koc and C. Paar, editors, *Workshop on Cryptographic Hardware and Embedded Systems (CHES '99)*, volume LNCS 1717. Springer-Verlag, August 1999.

Xil99. Xilinx. *The Programmable Logic Data Book*. Xilinx, Inc., 1999.

A Relevant Algorithms

Algorithm 1: Double-and-add scalar multiplication using projective coordinates

```
double_and_add(x, y, k)
  (X, Y, Z) = conv_projective(x,y)
  (X_0, Y_0, Z_0) = (X, Y, Z) /* P_0 = P */
  for i = l − 2 downto 0 do
    (X, Y, Z) = double(X, Y, Z) /* P = 2P */
    if k_i = 1 then /* P = P + P_0 */
      (X, Y, Z) = add(X_0, Y_0, Z_0, X, Y, Z)
    end if
  end for
  (x, y) = conv_affine(X, Y, Z)
return (x, y)
```

```
double(X, Y, Z)
  /* if P = O then return O */
  if (X, Y, Z) = O then return(O)
  else /* P ≠ O return 2P */
    Z_2 = X * Z^2
    X_2 = (X + b^{1/4} Z^2)^4
    U = Z_2 + X^2 + YZ
    Y_2 = X^4 Z_2 + U X_2
  endif
return(X_2, Y_2, Z_2)
```

```
conv_projective(x, y)
return (X = x, Y = y, Z = 1)
```

```
conv_affine(X, Y, Z)
return(x = X/Z^2, y = Y/Z^3)
```

```
add(X_0, Y_0, Z_0, X_1, Y_1, Z_1)
  /* if P_1 = O then return P_0 */
  if (X_1, Y_1, Z_1) = O then
    return(X_0, Y_0, Z_0)
  /* else if P_0 = −P_1 then return O */
  else if (X_0, Y_0, Z_0) = −(X_1, Y_1, Z_1)
    then return(O)
  /* else if P_0 = P_1 then return 2P_0 */
  else if (X_0, Y_0, Z_0)=(X_1, Y_1, Z_1) then
    (X_2, Y_2, Z_2) = double(X_0, Y_0, Z_0)
  else /* return P_2 = P_0 + P_1 */
    U_0 = X_0 Z_1^2
    S_0 = Y_0 Z_1^3
    U_1 = X_1 Z_0^2
    W = U_0 + U_1
    S_1 = Y_1 Z_0^3
    R = S_0 + S_1
    L = Z_0 W
    V = R X_1 + L Y_1
    Z_2 = L Z_1
    T = R + Z_2
    X_2 = a Z_2^2 + TR + W^3
    Y_2 = T X_2 + V L^2
  endif
return(X_2, Y_2, Z_2)
```

Algorithm 2: Montgomery scalar multiplication using projective coordinates

```
montgomery_scalar_multiplication(x, y, k)
  /* P_1 = P,  P_2 = 2P */
  (X_1, Z_1, X_2, Y_2) = conv_projective(x, y)
  for i = l − 2 downto 0 do
    if k_i = 1 then
      /*P_1.X = P_1.X + P_2.X, P_2.X = 2P_2.X*/
      (X_1, Z_1) = madd(X_1, Z_1, X_2, Z_2, x)
      (X_2, Z_2) = mdouble(X_2, Z_2)
    else
      /*P_2.X = P_1.X + P_2.X, P_1.X = 2P_1.X*/
      (X_2, Z_2) = madd(X_2, Z_2, X_1, Z_1, x)
      (X_1, Z_1) = mdouble(X_1, Z_1)
    end if
  end for
return(compute_xy(X_1, Z_1, X_2, Z_2, x, y))
```

```
conv_projective(x, y)
 X_1 = x;   Z_1 = 1
 X_2 = x^4 + b;   Z_2 = x^2
return(X_1, Z_1, X_2, Z_2)
```

```
madd(X_1, Z_1, X_2, Z_2, x)
 X_1 = X_1 Z_2 X_2 Z_1 + x(X_1 Z_2 + X_2 Z_1)^2
 Z_1 = (X_1 Z_2 + X_2 Z_1)^2
return(X_1, Z_1)
```

```
mdouble(X, Z)
return(X = X^4 + bZ^4, Z = X^2 Z^2)
```

```
compute_xy(X_1, Z_1, X_2, Z_2, x, y)
 x_k = X_1/Z_1
 y_k = ((X_1/Z_1 + x)(X_2/Z_2 + x)+
       x^2 + y) * (X_1/Z_1 + x)/x + y)
return(x_k, y_k)
```

Fast Implementation of Elliptic Curve Defined over $GF(p^m)$ on CalmRISC with MAC2424 Coprocessor

Jae Wook Chung, Sang Gyoo Sim, and Pil Joong Lee

Dept. of Electronic and Electrical Eng., POSTECH
{jwchung,sim}@oberon.postech.ac.kr
pjl@postech.ac.kr

Abstract. In this paper, we propose fast finite field and elliptic curve (EC) algorithms useful for embedding cryptographic functions on high performance device such that most instructions take just one cycle. In such case, the integer multiplications and additions have the same computational cost so that the computational cost analyses that were previously done in traditional manner may be invalid and in some cases the new algorithms should be introduced for fast computation. In our implementation, column major method for field multiplication and BP inversion algorithm are used for fast field arithmetic, and mixed coordinates method is used for efficient EC exponentiation. We give here analyses on various algorithms that are useful for implementing EC exponentiation on CalmRISC microcontroller with MAC2424 coprocessor, as well as new exact analyses on BP (Bailey-Paar) inversion algorithm and EC exponentiation. Using techniques shown in this paper, we implemented EC exponentiation for various coordinate systems and the best result took 122ms, assuming 50ns clock cycle.

1 Introduction

Since Koblitz[8] and Miller[11] first introduced elliptic curve cryptography (ECC), many works[7,10,2] have shown that ECC can be very efficiently embedded into restricted hardware such as smart cards. During the past few years, most people believed that elliptic curve defined over $GF(2^m)$ was the only useful one for hardware implementation since it can be implemented with only simple bit operations. $GF(p)$ and $GF(p^m)$ were popular in computer software implementation but they were not in hardware implementation because a math coprocessor is required for its implementation in smart cards and it significantly increases the cost.

However ECC is not restricted to smart cards. There can be many hardware applications that already have a fast microcontroller with math coprocessor. One such application is a portable MP3 player, it needs a high performance microcontroller which supports fast integer multiplication and division to decode MP3 data and it also needs cryptographic services to prevent unauthorized copy of

Ç.K. Koç and C. Paar (Eds.): CHES 2000, LNCS 1965, pp. 57–70, 2000.

MP3 files. In this case, $GF(p)$ or $GF(p^m)$ is more likely the better than $GF(2^m)$, since they can utilize the fast multiplication and division instructions. $GF(p)$ and $GF(p^m)$ both are good choices for that kind of application, but $GF(p^m)$ seems to be a better choice because there is no need to implement complex multiple precision routines and it utilizes the full capability of the microcontroller. Not only is it easy to implement but also more efficient, since there are no carry propagation and the inversion in $GF(p^m)$ is far more efficient than that in $GF(p)$. Many works [10,2,1] have shown that $GF(p^m)$ is very suitable choice for computer software implementation.

We have implemented EC over $GF(p^{10})$ in CalmRISC microcontroller with MAC2424 coprocessor. CalmRISC is a very fast 8-bit RISC microcontroller and MAC2424 is a high performance math coprocessor that can compute 24-bit signed multiplication just in one cycle and that provides efficient division step instruction. Since integer multiplication and division are the critical operations in $GF(p^m)$, such devices provide the best platform for implementing EC defined over $GF(p^m)$.

This paper focuses on implementing EC defined over $GF(p^m)$ in CalmRISC microcontroller with MAC2424 math coprocessor. In particular, we have used $GF(p^{10})$, satisfying OEF (Optimal Extension Field [2]) conditions, where $p = 2^{16} - 165 =$ `0xff5b` and the irreducible polynomial being $f(x) = x^{10} - 2$.

2 Processor Features

2.1 CalmRISC Microcontroller

CalmRISC is Samsung's 8-bit low power RISC microcontroller that follows Harvard style. Both instruction and data can be fetched simultaneously without causing a stall using separate paths for memory access. CalmRISC has a 3-stage pipeline:

1. Instruction Fetch (IF)
2. Instruction Decode/Data Memory Access (ID/MEM)
3. Execution/Writeback (EXE/WB)

The first stage (or cycle) is IF, where the instruction pointed to by the program counter is read into the instruction register (IR). The second stage is ID/MEM, where the fetched instruction (stored in IR) is decoded and the data memory access is performed, if necessary. The final stage is Execution and Writeback stage (EXE/WB), where the required ALU operation is executed and the result is written back into the destination registers. Since CalmRISC instructions are pipelined, the next instruction fetch is not postponed until the current instruction is completely finished, but it is performed immediately after the current instruction fetch is done.

Most of CalmRISC instructions are 1-word instruction, while branch instructions such as long "call" and "jump" instructions are a 2-word instruction. Thus the number of clocks per instruction (CPI) is 1 except for long branches, which take 2 clock cycles per instruction.

2.2 MAC2424 Math Coprocessor

MAC2424 is a 24-bit high performance fixed-point DSP coprocessor for CalmRISC microcontroller. Main datapaths are constructed to 24-bit width, but it can also perform 16-bit data processing efficiently in 16-bit operation mode.

There are two modes of operation in MAC2424: 24-bit mode operation and 16-bit mode operation.

24-Bit Mode Operation.

- Signed fractional/integer 24 x 24-bit multiplication in single cycle
- 24 x 24-bit multiplication and 52-bit accumulation in single cycle
- 24-bit arithmetic operation
- Two 48-bit multiplier accumulator with 4-bit guard
- Two 32K x 24-bit data memory spaces

16-Bit Mode Operation.

- Four-Quadrant fractional/integer 16 x 16-bit multiplication in single cycle
- 16 x 16-bit multiplication and 40-bit accumulation in single cycle
- 16-bit arithmetic operation with 8-bit guard
- Two 32-bit multiplier accumulator with 8-bit guard
- Two 32K x 16-bit data memory spaces

2.3 Programming Environment

CalmSHINE is a C compiler for CalmRISC and MAC2424. It also supports assembly language. Thus architecture specific low-level instructions (such as 24-bit by 24-bit multiplication and accumulation) can be utilized via assembly language. Non-architecture specific functions may be written in C language.

3 Finite Field Arithmetic

Optimization of finite field arithmetic is very critical to the overall performance of EC operations. In this section, we describe algorithms for implementing efficient finite field arithmetic. In our implementation, we use $GF(p^m)$ where $p =$`0xff5b` (16 bits), $m = 10$ and $f(x) = x^{10} - 2$ as an irreducible polynomial. Although CalmRISC supports 24-bit by 24-bit multiplication, it is signed multiplication and memory access is very inefficient in 24-bit mode due to the memory alignment. This is why we use 16-bit p.

3.1 Modular Reduction

Modular reduction is used very frequently and it is the bottleneck of the performance of finite field arithmetic. In most computer software implementations, the modular reduction using simple bit shifts and additions is a popular choice and provides a very good performance when p is a pseudo-Mersenne number. This is due to the fact that the division instruction is very slow for most hardware. However this is not the case for MAC2424, since every operation is simple and it takes only one cycle except long-branch operations. Thus modular reduction using division step instruction is desirable for MAC2424. Moreover it has an advantage that the intermediate values do not need to move around between the registers. During the division steps in MAC2424, the dividend and divisor keep their position until the division ends. In our implementation, the modular reduction by repeated division step instruction takes 39 cycles, while the modular reduction by bit shifts and additions takes 90 cycles.

3.2 Field Multiplication and Squaring

We considered three different algorithms for finite field multiplication, Karatsuba-Offman algorithm (KOA), column major method and row major method. First we consider KOA. KOA works by reducing the number of multiplications while increasing the number of cheap additions/subtractions by the recursively. In general, it gives about $10 \sim 20\%$ performance enhancement for most architecture. However the computational cost for multiplication and addition/subtraction is exactly the same for MAC2424, so reducing the number of multiplication with sacrificing the number of addition/subtraction does not help.

Row major method is just a schoolbook method, so we skip the description here. Column major method is described as follows. This is not a general method but it is for our specific case where $f(x)$ is binomial ($f(x) = x^m - \alpha$), and with this algorithm the polynomial reduction and multiplication can be done simultaneously.

Algorithm 1 (Column Major Multiplication).

Input: $A(x)$ *and* $B(x) \in GF(p^m)$
Output: $C(x) = A(x)B(x)$ *mod* $f(x)$ $(f(x) = x^m - \alpha)$
for $k = 0$ *to* $m - 1$ *do followings*

1. $z \leftarrow 0$, $i \leftarrow m - 1$, $j \leftarrow k + 1$
2. *while* $i > k$, $z \leftarrow z + a_i b_j$, $i \leftarrow i - 1$, $j \leftarrow j + 1$
3. $z \leftarrow z \cdot \alpha$, $j \leftarrow 0$
4. *while* $i \geq 0$, $z \leftarrow z + a_i b_j$, $i \leftarrow i - 1$, $j \leftarrow j + 1$
5. $c_k \leftarrow z$ *mod* p

Row major method and column major method both may be good choices because the required number of operation is both equal, but the row major

method has the disadvantage of storing full one intermediate row in a temporary memory. Moreover column major is preferable since MAC2424 can multiply-and-accumulate simultaneously in one cycle. So the column major method of field multiplication is definitely a better choice for MAC2424. Algorithm 1 uses m^2+m-1 multiplication instructions and m modular reductions with modulus p. Algorithm 1 can be similarly applied to field squaring, so $\frac{m(m+1)}{2}+m-1$ multiplication instructions and m modular reductions with modulus p are needed for field squaring. Modular reduction is performed only m times because product of two subfield elements can be safely accumulated multiple times in MAC2424's accumulator and α is small enough ($\alpha = 2$) that z in Algorithm 1 never overflows. This means we don't need to reduce the intermediate values, instead we need to reduce just the final values. In our implementation, field multiplication takes 723 cycles and field squaring takes 717 cycles. The ratio of field squaring to field multiplication is almost close to 0.9 in our case. This is due to the fact that the most of the time is taken in modular reduction and that MAC2424 can multiply very fast.

3.3 Field Inversion

There have been many research efforts on finite field inversion algorithm. Well-known algorithms are extended Euclidean algorithm, almost inversion algorithm, and their variants. The efficiency of a finite field inversion algorithm can be roughly measured by counting the subfield inversion it uses since the subfield inversion is the most time consuming job among the subfield arithmetic. Even for MAC2424, subfield inversion could not be done fast. It takes 670 cycles in our implementation. Among the various finite field inversion algorithms we consider IM (Inversion with Multiplication) [10] and BP [1] algorithms since only they require just one subfield inversion. Here we review the IM algorithm and the BP algorithm.

Algorithm 2 (IM Inversion Algorithm). *Initialize* $B \leftarrow 0$, $C \leftarrow 1$, $F \leftarrow f(x)$, $G \leftarrow A(x)$

1. *If* $deg(F) = 0$ *then* $B \leftarrow B \cdot (F_0^{-1} \bmod p)$, *return* B.
2. *If* $deg(F) < deg(G)$ *then exchange* F, B *with* G, C.
3. $j = deg(F) - deg(G)$
   (a) *If* $j \neq 0$ *do the followings.*
   $\alpha \leftarrow G_{deg(G)}^2 \bmod p$,
   $\beta \leftarrow F_{deg(F)} G_{deg(G)} \bmod p$,
   $\gamma \leftarrow G_{deg(G)} F_{deg(F)-1} - F_{deg(F)} G_{deg(G)-1} \bmod p$,
   $\{F,\ B\} \leftarrow \alpha\{F,\ B\} - (\beta x^j + \gamma x^{j-1})\{G,\ C\}$.
   (b) *If* $j = 0$ *do the followings.*
   $\{F,\ B\} \leftarrow G_{deg(F)}\{F,\ B\} - F_{deg(F)}\{G,\ C\}$
4. *Goto step 2.*

In Algorithm 2, capitalized variables represent the polynomial representation with indeterminate x, $deg(F)$ denotes the degree of polynomial F and F_i is the coefficient of x_i in F.

The BP algorithm is exponentiation-based inversion algorithm using the fact that A^p where $A(x) = a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + \cdots + a_0 \in GF(p^m)$ can be computed very efficiently only if the irreducible polynomial for $GF(p^m)$ is a binomial of the form $f(x) = x^m - \alpha$. The following equation shows that only $m-1$ subfield multiplications are required for p-th power of a field element. Note that $\alpha^{\lfloor pi/m \rfloor} \bmod p$ and $pi \bmod m$ $(i = 1, \ldots, m-1)$ should be pre-computed beforehand.

$$\begin{aligned} A(x)^p &= a_{m-1}x^{p(m-1)} + a_{m-2}x^{p(m-2)} + \cdots + a_0 \\ &= \sum_{i=0}^{m-1} a_i \alpha^{\lfloor pi/m \rfloor} x^{pi} \bmod m \end{aligned} \tag{1}$$

One might have noticed that the p-th power repeated by i times can be collapsed to one p^i-th power which have the same computational cost with p-th power. Now we have all apparatus for BP algorithm:

Algorithm 3 (BP Inversion Algorithm). *Computes $A(x)^{-1}$ as follows.* $A(x)^{-1} = (A(x)^r)^{-1}A(x)^{r-1}$ *mod* $f(x)$ *where* $r = \frac{p^m-1}{p-1} = 1+p+p^2+\cdots+p^{m-1}$

In Algorithm 3, $(A(x)^r)^{-1}$ is always a subfield inversion. In this algorithm, computing $A(x)^{r-1}$ is very critical to the performance of finite field inversion. Since $r-1$ is in a special form we can compute $A(x)^{r-1}$ efficiently by addition chain and p^i-th power. The efficient method was already shown in [1], but we want to show that the analysis shown in [1] and [10] is incorrect.

Table 1. Example of Computing A^{r-1} for $r = p + p^2 + \cdots + p^5$, $m = 6$

Computing A^{r-1} where $r = p + p^2 + \cdots + p^5$	
Our method	Bailey & Paar's method
$B \leftarrow A^p = A^{(10)}$	$B \leftarrow A^p = A^{(10)}$
$T_1 \leftarrow AB = A^{(11)}$	$T_1 \leftarrow BA = A^{(11)}$
$T_2 \leftarrow T_1^{p^2} = A^{(1100)}$	$B \leftarrow T_1^{p^2} = A^{(1100)}$
$T_2 \leftarrow T_1T_2 = A^{(1111)}$	$B \leftarrow BT_1 = A^{(1111)}$
$T_2 \leftarrow T_2^{p^2} = A^{(111100)}$	$B \leftarrow B^p = A^{(11110)}$
$B \leftarrow T_2B = A^{(111110)}$	$B \leftarrow BA = A^{(11111)}$
	$B \leftarrow B^p = A^{(111110)}$

The required number of p^i-th power in BP algorithm is not always $\lfloor log_2(m-1) \rfloor + H_W(m-1)$ where $H_W(\cdot)$ is Hamming weight. Instead, the number of p^i-th power is at least one less than this when m is even except for $m = 2$. Table 1 shows that one p^i-th power can be reduced for $m = 6$ considering this fact. In general, if m is even, the exact number of p^i-th power is:

$$\#p^i\text{-th power} = \begin{cases} \lfloor log_2(m-1) \rfloor + 1 & \text{if } m \text{ is 2's power} \\ \lfloor log_2(m-1) \rfloor + H_W(m) - 1 & \text{if } 2|m \text{ , not 2's power} \end{cases} \quad (2)$$

Table 2. Computational Cost Analysis for BP and IM Inversion Algorithm

Algorithm		#Multiplication	#Reduction	#Inv
IM ($f(x) = x^m - w$)		$3m^2 - m - 7$	$m^2 + 4m - 8$	1
BP	Original analysis	$t_1(m) \cdot (m^2 + 2m - 2) + 3m - 1$	$t_1(m) \cdot (3m - 2) + 2m$	1
	New analysis	$t_1(m) \cdot (m^2 + m - 1)$ $+t_2(m) \cdot (m-1) + 2m + 1$	$t_1(m) \cdot (2m - 1)$ $+t_2(m) \cdot (m-1) + m + 2$	1
	special case	„	$t_1(m) \cdot m$ $+t_2(m) \cdot (m-1) + m + 1$	1
	$t_1(m) = \lfloor log_2(m-1) \rfloor + H_W(m-1) - 1$ $t_2(m) = \begin{cases} \text{Eqn.(2)} & \text{if } m \text{ is even} \\ t_1(m) + 1 & \text{if } m \text{ is odd} \end{cases}$			

Table 2 shows the new exact analyses of BP algorithm and it is compared with IM algorithm. Note that we corrected minor counting errors that were shown in previous works[1,10]. We also show analyses for the 'special case' when the product of two subfield elements can be accumulated without overflow, and when α is small, that is our case. In that case, we can save $m - 1$ modular reduction in field multiplication and 1 modular reduction in final field multiplication that only computes the constant term of $A(x)^r$ from $A(x)^{r-1}$ and $A(x)$.

In Table 2, the term "multiplication" means general multiplication performed by the processor, not the field or subfield multiplication. According to Table 2, the complexity of BP looks greater than that of IM. However if m is not too large, BP can provide better performance. When $m = 10$, that is the specific case for our implementation, IM requires 283 multiplication and 132 modular reduction, and BP requires 493 multiplication and 87 modular reduction (note that one less number of p^i-th power is required for $m = 10$ than the analysis shown in Table 2). Recall that, for MAC2424 the number of modular reduction is more significant (costs much more) than that of multiplication. Hence we conclude that BP algorithm will perform much better in our case ($283+132\times39 = 5431$(IM) $>$ $493 + 87 \times 39 = 3886$(BP)). In addition, not only does the BP algorithm take fewer cycles but also it is simple to implement as looping or branching is not needed. More improvement is possible for BP algorithm when m has a factor of 2. In our specific case, p^i-th power is computed as follows.

Algorithm 4 (p-th Power Algorithm for Our Specific Case). $B(x) = A(x)^p$

1. *Array X is pre-computed as*
   $X = \{$`1, 0x5AC3, 0x7B13, 0x9EEF, 0x7E9E, 0xFF5A(= -1), 0xA498, 0x8448, 0x606C, 0x80BD`$\}$
2. *For $i = 0$ to 9*
   $B_i = B_{(ip \ mod \ m)} = A_i \cdot X_i \ mod$ `0xff5b`

Algorithm 4 requires 8 multiplications, 8 modular reductions and 1 subtraction. Note that multiplying -1 is equal to subtracting from p. The following is the pre-computed value X for p^2-th power and p^4-th power algorithm for our specific case, respectively. To compute p^2-th or p^4-th power we only need to substitute the X in Algorithm 4 with the following Xs.

For p^2-th power: $X = \{$`1, 0x7B13, 0x7E9E, 0xA498, 0x606C, 1, 0x7B13, 0x7E9E, 0xA498,0x606C`$\}$
For p^4-th power: $X = \{$`1, 0x7e9e, 0x606c, 0x7b13, 0xa498, 1, 0x7e9e, 0x606c, 0x7b98, 0xa498`$\}$

The above each pre-computed array X has two 1s, so only 8 multiplications and 8 modular reductions are needed to compute p^2-th or p^4-th power. And even more, $X_{[1-4]}$ is identical to $X_{[6-9]}$, thus the memory can be saved.

Now we are ready to construct the most efficient method to compute $A(x)^{r-1}$ for our specific case, which leads to the least number of field multiplications and fully utilizes the above facts. The following algorithm efficiently computes $A(x)^{r-1}$ and we used this in our actual implementation.

$$
\begin{array}{lll}
T_1 \leftarrow A^p & (T_1 = A^p) & \text{(exp)} \\
T_2 \leftarrow A^p \cdot A & (T_2 = A^{1+p}) & \text{(mul)} \\
T_3 \leftarrow T_2^{p^2} & (T_3 = A^{p^2+p^3}) & \text{(exp)} \\
T_3 \leftarrow T_3 \cdot T_2 & (T_3 = A^{1+p+p^2+p^3}) & \text{(mul)} \\
T_4 \leftarrow T_3^{p^4} & (T_4 = A^{p^4+p^5+p^6+p^7}) & \text{(exp)} \\
T_3 \leftarrow T_3 \cdot T_4 & (T_3 = A^{1+p+p^2+p^3+p^4+p^5+p^6+p^7}) & \text{(mul)} \\
T_3 \leftarrow T_3^{p^2} & (T_3 = A^{p^2+p^3+p^4+p^5+p^6+p^7+p^8+p^9}) & \text{(exp)} \\
T_2 \leftarrow T_3 \cdot T_1 & (T_2 = A^{p+p^2+p^3+p^4+p^5+p^6+p^7+p^8+p^9}) & \text{(mul)}
\end{array}
$$

As shown above $A(x)^{r-1}$ is done by 4 field multiplications and 4 p^i-th powers. As it can be seen, since m is even, the number of field multiplication is equal to that of p^i-th power.

3.4 Performance of Field Arithmetic

Table 3 shows our finite field $GF(p^{10})$ implementation results. The cycles for each functions were measured using Samsung's CalmSHINE compiler and all

finite field functions were written in assembly. All cycles include the functional overhead such as parameter loading and data movements when entering and leaving the functions. In Table 3, 'I/M' is the ratio of field inversion to the field multiplication and 'S/M' is the ratio of field squaring to the field multiplication.

Table 3. Finite Field Implementation Result

Operation	Required Cycles
Add	187
Sub	141
Mult	723
Square	667
Sub_Inv	670
Mod	39
Inversion	5378
I/M	7.4
S/M	0.9

4 Elliptic Curve Arithmetic

In this section we discuss the method of optimizing EC exponentiation using mixed coordinate system. We optimize the EC exponentiation by combining mixed coordinate system from [5] and the Lim-Hwang's method [10].

4.1 Signed Window Algorithm for EC Exponentiation

Signed window method is known to be the most efficient method for computing EC exponentiation (EC scalar multiplication) excluding the fixed base exponentiation algorithms. Let k be a positive integer, and suppose we want to compute kP where P is an arbitrary point on an elliptic curve. Then k can be expressed as follows.

$$k = 2^{k_0}(2^{k_1}(\cdots 2^{k_{v-1}}(W[v] + W[v-1]) + W[v-2]\cdots) + W[0]) \tag{3}$$

where $W[i]$ is odd, $-2^w + 1 \leq W[i] \leq 2^w - 1$ and $w \leq k_i$. To compute EC exponentiation with signed window method, first pre-compute $P_i = iP$ ($i = \pm 1, \pm 3, \cdots, \pm(2^w - 1)$) and then evaluate the following equation using the pre-computed values.

$$kP = 2^{k_0}(2^{k_1}(\cdots 2^{k_{v-1}}(2^{k_v} P_{W[v]} + P_{W[v-1]}) + P_{W[v-2]}\cdots) + P_{W[0]}) \tag{4}$$

However pre-computation is not needed for $-2^w + 1 \leq W[i] \leq -1$ since negating EC point can be done easily, that is computing $P_{W[i]} = -P_{-W[i]}$ takes negligible time. It is also possible to implement an EC subtraction function to get rid of the redundant EC negating time using a small portion of additional program memory.

The first k_v doublings, in case $W[v] < 2^{w-1}$, can be more efficiently computed [5]. There have been an analysis on this in [5], however it is incorrect and we want to correct it here.

First we need to consider the probability that the bit size of $W[v]$ equals j. This can be easily computed and the following equation shows it.

$$Pr(|W[v]| = j) = \begin{cases} \frac{1}{2^{w-1}} & \text{if } j = 1 \\ \frac{1}{2^{w-j+1}} & \text{if } 2 \leq j \leq w \end{cases} \tag{5}$$

Use the fact that the above modification reduces $w + 1 - j$ doublings and increases 1 addition, and that we do not apply the above modification when $j = w$. Then can be easily verified that the average number of doublings reduced is $\frac{3}{2} - \frac{1}{2^{w-1}}$ and the average number of addition increased is $\frac{1}{2}$. Note that the average value of k_v is $w + 2$, so $w + 2$ doublings are needed in an average case if we don't use the above modification.

4.2 Mixed Coordinates System

We use mixed coordinates system to speed up computation of EC exponentiations. For a given rational integer k and an elliptic curve point P, we can evaluate the EC exponentiation kP by the following steps.

$$\begin{aligned} &T_0 = P_{W[v]} \\ &T_{i+1} = 2^{k_{v-i}} T_i + P_{W[v-i-1]} \text{ for } i = 0, 1, \ldots, v-1 \\ &kP = 2^{k_0} T_v \end{aligned} \tag{6}$$

The EC exponentiation kP is computed by repeating basic step $T_{i+1} = 2^{k_{v-i}} T_i + P_{W[v-i-1]}$, which is equal to $T_{i+1} = 2T' + P_{W[v-i-1]}$ where $T' = 2^{k_{v-i}-1} T_i$. If we represent the elliptic curve points $(T_i, 2T', P_{W[v-i-1]})$ as coordinates (C_1, C_2, C_3), the computational cost for a basic step is

$$(k_{v-i} - 1) \cdot t(2C_1) + t(2C_1 = C_2) + t(C_2 + C_3 = C_1).$$

In this paper, we denote affine coordinate as A [6,12], projective coordinate P [9,6], Jacobian coordinate J [3], modified Jacobian coordinate J^m [10] and Chudnovsky-Jacobian coordinate J^C [3]. Note that we use a different Modified Jacobian coordinate system. The Modified Jacobian coordinate shown in [10] is better because it reduces one field addition/subtraction in EC addition.

Let us now discuss suitable coordinate systems for C_1, C_2 and C_3. Since doublings in C_1 are repeated most frequently, we should choose C_1 such that $t(2C_1)$ is the smallest, thus we select J^m as C_1.

Since pre-computation $P_{W[i]}$ is done during online time in signed window method, i.e. the resulting values are not saved in auxiliary memory for another exponentiation, the pre-computation time is included in total elapsed time. Thus we should select coordinate C_3 suitable to compute the values $P_{w[i]}$. Cohen *et al.*[5] proposed to use either affine coordinate or Chudnovsky-Jacobian coordinate as C_3 and to select one by comparing $t(J^C + J^C)$ and $t(A + A)$. However, since the ratio I/M is relatively small, we chose $C_3 = A$. Then there are two pre-computation methods. First, we can compute $Pi = iP(i = 1, 3, 5, \ldots, 2^w - 1)$ in affine coordinate by simple method by repeating $P_{i+2} = P_i + P'$ for $i = 1, 3, 5, \ldots, 2^w - 3$ where $P_1 = P$ and $P' = P + P$. Here, the total computational cost is $t(2A) + (2^{w-1} - 1) \cdot t(A + A) = 2^{w-1}\mathrm{I} + 2^w\mathrm{M} + (2^{w-1} + 1)\mathrm{S}$. To reduce the number of inversion in $F(p^m)$, we can apply 'Montgomery trick of simultaneous inversion [4]' with sacrificing the number of multiplications and squares. The total cost in that case is $w\mathrm{I} + (5 \cdot 2^{w-1} + 2w - 10)\mathrm{M} + (2^{w-1} + 2w - 3)\mathrm{S}$. Table 4 shows the expected computational cost for these two methods. In Table 4, the computational costs for pre-computation in case of $C_3 = J^c$ were also shown for comparison.

Table 4. Computational Cost for Various Pre-computation Methods

Method	Computational cost
Affine(simple)	8I + 16M+9S = 83.3M
Affine(Mont. trick)	4I + 38M + 13S = 79.3M
Chudnovsky-Jacobian 1	77M + 26S = 100.4M
Chudnovsky-Jacobian 2	I + 55M + 23S = 83.1M

In Table 4, Montgomery's trick is shown to be the best choice. However we didn't use the Montgomery trick, since online pre-computation time is just a very small part of EC exponentiation, and it does not significantly improves the EC exponentiation time. In addition, the Montgomery's method requires much more program memory than the simple method without giving much improvement in performance. We chose to use simple method in online pre-computation.

Let us discuss suitable coordinate for C_2. Since we selected modified Jacobian coordinate and affine coordinate for C_1 and C_3 respectively, coordinate for C_2 should minimize $(k_{v-i} - 1) \cdot t(2J^m) + t(2J^m = C_2) + t(C_2 + A = J^m)$, that is, it should minimize $t(2J^m = C_2) + t(C_2 + A = J^m)$. Although there are 5 candidates for C_2, Table 5 shows computational amounts to compute a basic step (Eqn. 6) using 3 candidates of least cost. In Table 5, we assumed window size $w = 4$.

In Table 5, the 1-bit gap between the two neighboring diminished windows is considered to be the worst case (i.e. $k_i = w + 1$ for $i = 1, 2, \ldots, v$), and the 2-bit

Table 5. Candidates for Best Mixed Coordinates System and their Analyses

Coordinate	Cost	Worst case ($k_{v-1}=5$)	Average case ($k_{v-1}=6$)
(J^m, J, A)	$(7.6k_{v-1}+12.5)$M	50.5M	58.1M
(J^m, J^c, A)	$(7.6k_{v-1}+12.5)$M	50.5M	58.1M
(J^m, J^m, A)	$(7.6k_{v-1}+13.5)$M	51.5M	59.1M

gap is considered to be the average case (i.e. $k_i = w+2$ for $i = 1, 2, \ldots, v$). According to Table 5, we can select either Jacobian coordinate(J) or Chudnovsky-Jacobian coordinate(J^c) for C_2. Since Chudnovsky-Jacobian coordinate uses 2 more finite field $F(p^m)$ elements than Jacobian, it is inefficient in storage. Thus we select Jacobian coordinate for C_2. Consequently, for $(C_1, C_2, C_3) = (J^m, J, A)$, $w = 4$ and $|k| = 160$, we can compute an EC exponentiation kP with following computational cost in average.

$$\begin{aligned} t(2A) + (2^{w-1}-1)\cdot t(A+A) + \frac{1}{2}\cdot t(A+A=J^m) \\ + (w + \frac{1}{2^{w-1}} - \frac{1}{2})\cdot t(2J^m) + t(J+A=J^m) \\ + \{\frac{|k|-w+1-(\frac{1}{2})^{w-1})}{w+2} - 1\}\cdot \left\{ \begin{array}{l} (w+1)\cdot t(2J^m) \\ +t(2J^m=J) + t(J+A=J^m) \end{array} \right\} \\ \approx 8I + 849.7M + 763.7S \approx \ 1596\text{M} \quad (7) \end{aligned}$$

In worst case, we can compute kP with the following cost.

$$\begin{aligned} t(2A) + (2^{w-1}-1)\cdot t(A+A) + t(A+A=J^m) + t(2J^m=J) + t(J+A=J^m) \\ + \{\frac{|k|-1}{w+1} - 1\}\cdot \{w\cdot t(2J^m) + t(2J^m=J) + t(J+A=J^m)\} \\ \approx 8I + 895.4M + 792S \approx \ 1667\text{M} \quad (8) \end{aligned}$$

5 Implementation Results

We implemented elliptic curve exponentiation in CalmRISC with MAC2424 coprocessor using all algorithms shown in previous sections. All finite field functions were written in assembly language since time critical low-level instructions cannot be programmed in high-level language, and all elliptic curve functions were written in C language on top of the finite field functions. Table 6 shows our implementation of elliptic curve exponentiation in various coordinate systems. Note that the result shown in Table 6 was measured using CalmSHINE C compiler. CalmSHINE compiler measures the clock cycle for each function exactly, however it can be done only in 'debug build mode' and CalmSHINE compiler

does very poor code optimization in 'debug build mode'. This is why the result shown in Table 6 is slower than what is expected. In real implementation with optimized codes, it will perform much better. Referring to Table 6, mixed coordinates is the best with almost 10% of improvement over fastest single coordinate system (Modified Jacobian).

Table 6. Implementation Result of Elliptic Curve Exponentiation

EC Exponentiation Result		
Coordinate	Cycles	Time
(J^m, J, A)	2448265	122ms
(A, A, A)	3632657	182ms
(J^m, J^m, J^m)	2711543	135ms

6 Conclusions

In this paper, we proposed optimized algorithms for implementing EC in CalmRISC with MAC2424 math coprocessor, in which all instructions take just one clock cycle, and we showed implementation results and full analyses on their performances. We also gave new exact alalyses on BP inversion algorithm and EC exponentiation. In our implementation, we used column major method for field multiplication and slightly improved BP algorithm for field inversion. Mixed coordinates using Lim-Hwang's Modified Jacobian coordinate was applied for for efficient EC exponentiation. Our implementation of EC exponentiation took about 122ms (assuming one cycle takes 50ns), which is about 10% of improvement over single coordinate system. This result can be much better in real implementation with CalmSHINE's optimized compile mode. Although the algorithms shown in this paper is focused on our specific case, it can be easily applied to other environments where all basic arithmetic instructions have the same computational cost.

Acknowledgement

This project was supported mainly by Samsung Electronics Co. Ltd., and partially by Brain Korea 21 and Com^{2}MaC-KOSEF.

References

1. Bailey, D., Paar, C.: Efficient Arithmetic in Finite Field Extensions with Application in Elliptic Curve Cryptography, To appear in *Journal of Cryptology* (Available at `http://ece.wpi.edu/People/faculty/cxp.html`)
2. Bailey, D. V. and Paar, C.: Optimal extension field for fast arithmetic in public key algorithms, *Advances in Cryptology-Crypto'*98, Lecture Notes in Computer Science, Vol 1462. Springer-Verlag, (1998), 472-485.
3. Chudnovsky, D. V. and Chudnovsky, G. V.: Sequences of numbers generated by addition in formal groups and new primality and factorization tests, *Advances in Applied Math.*, Vol. 7. (1986), 385-434.
4. Cohen, H.: *A course in computational algebraic number theory*, Graduate Texts in Math., Vol. 138. Springer-Verlag, (1993).
5. Cohen, H., Miyaji, A. and Ono, T.: Efficient Elliptic Curve Exponentiation Using Mixed Coordinates, *Advances in Cryptology-Asiacrypt'98*, Lecture Notes in Computer Science, Vol. 1514. Springer-Verlag, (1998), 50-65.
6. IEEE P1363: Standard Specifications for Public Key Cryptography, Working Draft 12, Nov. (1999).
7. Itoh, K., Takenaka, M., Torll, N., Temma, S. and Kurihara, Y.: Fast Implementation of Public-Key Cryptography on a DSP TMS320C6201, *Cryptographic Hardware and Embedded Systems*'99, Lecture Notes in Computer Science, Vol. 1717. Springer Verlag, (1999), 61-72.
8. Koblitz, N.:, Elliptic Curve Cryptosystems, *Math. Comp.*, Vol. 48. pp. (1987), 203-209.
9. Koyama, K. and Tsuruoka, Y.: Speeding up elliptic cryptosystems by using a signed binary window method, *Advances in Cryptology-Proceedings of Crypto*'92, Lecture Notes in Computer Science, Vol. 740. Springer-Verlag, (1993), 345-357.
10. Lim, C. H. and Hwang, H. S.: Fast Implementation of Elliptic Curve Arithmetic in $GF(p^n)$, *Public Key Cryptography*, Lecture Notes in Computer Science, Vol. 1751. Springer-Verlag, (2000), 405-421.
11. Miller, V. S.: Use of Elliptic Curves in Cryptography, *Advances in Cryptology-Proceedings of Crypto* '85, Lecture Notes in Computer Science, Vol. 218. Springer-Verlag, (1986), 417-426.
12. Silverman, J. H.: *The Arithmetic of Elliptic Curves*, GTM 106. Springer-Verlag, New York (1986).

Protecting Smart Cards from Passive Power Analysis with Detached Power Supplies

Adi Shamir

Dept. of Applied Math.
The Weizmann Institute of Science
Rehovot 76100, Israel
shamir@wisdom.weizmann.ac.il

Abstract. Power analysis is a very successful cryptanalytic technique which extracts secret information from smart cards by analysing the power consumed during the execution of their internal programs. It is a passive attack in the sense that it can be applied in an undetectable way during normal interaction with the smart card without modifying the card or the protocol in any way. The attack is particularly dangerous in financial applications such as ATM cards, credit cards, and electronic wallets, in which users have to insert their cards into card readers which are owned and operated by potentially dishonest entities.
In this paper we describe a new solution to the problem, which completely decorrelates the external power supplied to the card from the internal power consumed by the chip. The new technique is very easy to implement, costs only a few cents per card, and provides perfect protection from passive power analysis.
Keywords: Smart cards, power analysis, SPA, DPA.

1 Introduction

Hundreds of millions of smart cards are used today in thousands of applications which include cellular telephony, pay TV, computer access control, storage of medical information, identification cards, stored value cards, credit cards, etc. These cards are typically used by executing cryptographic computations based on secret keys embedded in their non-volatile memories. The goal of an attacker is to extract these secret keys from the tamper resistant card in order to modify the card's contents, to create a duplicate card, or to generate an unauthorized transaction.

We distinguish between two types of attacks:

1. *An active attack*, in which the smart card chip can be extracted, modified, probed, partially destroyed, or used in unusual environments. Active attacks leave clearly visible signs of tampering, and thus they are usually applied to stolen cards, or in situations in which the owner of the smart card is interested in defeating its security (e.g., in pay TV or telephony applications). They include fault attacks [BDL], probing attacks [KK], chip microsurgery with focused ion beam (FIB) devices, etc. They typically require considerable amount of time, sophisticated equipment and detailed knowhow of

Ç.K. Koç and C. Paar (Eds.): CHES 2000, LNCS 1965, pp. 71–77, 2000.

the physical design of the chip. They are extremely powerful in extracting system-wide information about the smart card system, but are rarely used to extract individual user keys due to their cost and complexity.

2. *A passive attack*, in which the smart card can only be externally watched during its normal interaction with a (possibly modified) smart card reader. This is the preferred attack when the owner of the smart card is interested in preserving its security, e.g., in financial applications: An ATM card can be used to withdraw cash from a foreign cash dispensing machine operated by an unfamiliar financial institution, a credit card can be used to pay for merchandise in a mafia-affiliated store, and a mondex-like card can be used to transfer money to a purse owned by a dishonest taxi driver. In all these cases, smart cards which will be misused, retained, returned late, or returned damaged by active attacks will be immediately reported by the card owner to the card issuer, who will launch an investigation. Passive attacks include timing attacks [K], glitch attacks [KK], and power analysis [KJJ]. They require little sophistication and minimal investment, and can be carried out against a large number of individual cards by a small number of rogue card readers.

Timing and glitch attacks pose little risk to well designed smart card applications, since it is easy to protect the software and hardware elements of smart cards against them. However, power analysis is very easy to implement and very difficult to avoid. It is based on the observation that the detailed power consumption curve of a typical smart card (which describes how the externally supplied current changes over time) contains a huge amount of information about its operation. With sufficiently sensitive measuring devices, it is possible to watch the exact sequence of events (in the form of individual gates which switch on or off) during the execution of the microcode of each instruction. For example, the power consumption profiles of the addition and multiplication operations are completely different, the power consumed by writing 0..0 and 1..1 to memory are noticably different, and it is possible to visually extract the secret key of an RSA operation by determining which parts look like a modular squaring and which parts look like a modular multiplication.

In the Simple Power Analysis (SPA) variant of this attack, the attacker studies a single power consumption curve to obtain statistical information about the identity of the instructions and the Hamming weight of data words read from or written into memory at any given clock cycle. An example of the power consumed by a typical smart card during the execution of a DES encryption operation (at two time scales) is described in Fig. 1, which is taken from [KJJ]: at the top we can identify the 16 rounds of DES, the initial and final permutations, and other large scale structural details of the implementation; at the bottom, we can see the (noisy) details of the execution of a single round of DES.

The Differential Power Analysis (DPA) variant of this attack is even more powerful: the attacker studies multiple power consumption curves recorded from different executions with different inputs, and uses statistical differences between particular subsets of executions to find in an automated way particular key bits.

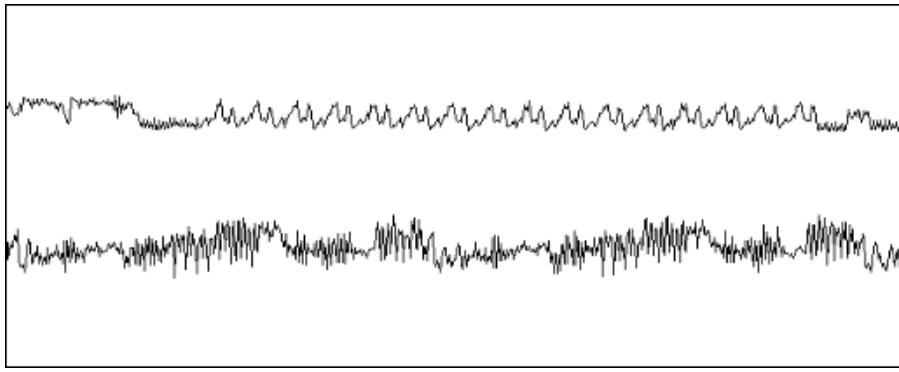


Fig. 1. The current supplied to standard smart cards

Kocher had publicly stated that with this DPA technique he managed to break essentially all the types of smart cards deployed so far by financial institutions.

Power analysis is usually a passive attack, since the smart card need not be modified in any way and cannot possibly know that its power supply is being monitored.

2 Previous Protective Techniques

After the publication of Kocher's SPA/DPA techniques, researchers and smart card manufacturers started looking for solutions. Attempts to make the power consumed by smart cards absolutely uniform by changing their physical design failed, since even small nonuniformity in the power consumption curve could be captured by sensitive digital oscilloscopes and analysed to reveal useful information. In addition, forcing all the instructions to switch the same number of gates on or off at the same points in time is a very unnatural requirement, which increases the area and total power consumption of the microprocessor, and slows it down.

Another proposed solution was to add a capacitor across the power supply lines on the smart card to smooth the power consumption curve. However, physical limitations restricted the size of the capacitor, and enough nonuniformity was left in the power consumption curve to make this a very partial solution, especially against DPA.

A related technique is to add to the smart card chip a sensor which measures the actual current supplied to the chip, and tries to actively equalize it by controling an additional current sink. However, the local changes in the power supply curve are so rapid that any compensation technique is likely to lag behind and leave many power spikes clearly visible.

Other proposed techniques include software-based randomization techniques, hardware-based random noise generators, unusual instructions, parallel execu-

tion of several instructions, etc. However, randomized software does not help if the attacker can follow individual instructions, and hardware noise can be eliminated by averaging multiple power consumption curves, and thus they provide only limited protection against a determined attacker with sensitive measuring devices.

A different solution is to replace the external power supply by an internal battery on the smart card. If the power pads on the smart card are not connected to the chip, the power consumption cannot be externally measured in a passive attack by the card reader. However, the thickness of a typical smart card is just 0.76 mm. Since such thin batteries are expensive, last a very short time, and are difficult to replace, this is not a practical solution.

An alternative solution is to use a rechargeable battery in each smart card. Such a battery can be charged by the external power supply whenever the card is inserted into a card reader, and thus we do not have to replace it so often. However, thin rechargeable batteries drain quickly even when they are not in use, and thus in normal intermittent use there is an unacceptably long charging delay before we can start powering the card from its internal battery. In addition, typical rechargeable batteries deteriorate after several hundred charging cycles, and thus the card has to be replaced after a relatively small number of intermittent transactions.

3 The New Proposal

In this paper we propose a new method which uses a simple "airgap" to completely decorrelate the power supplied to the card from the power consumed by the card. The basic idea is to use two capacitors as the power isolation element. During half the time capacitor 1 is (regularly) charged by the external power supply and capacitor 2 is (irregularly) discharged by supplying power to the smart card chip, and during the other half the roles of the two capacitors are reversed.

The behaviour of the capacitors is defined by a simple switch control unit and four power transistors which are added to the smart card chip (see Fig. 2). The preferred cyclic sequence of actions is:

1. The first capacitor is disconnected from external power.
2. The first capacitor is connected to the chip.
3. The second capacitor is disconnected from the chip.
4. The second capacitor is connected to the external power.

With this behaviour the smart card chip is always powered by at least one capacitor, but the external power supply is never connected directly to the internal chip. The supplied current has the uniform and predictable form described in Fig. 3, whereas the consumed current can continue to have the highly irregular shape of Fig. 1. The capacitors are connected via diodes to prevent leakage from the charged capacitor to the discharged capacitor during the brief moments in which they are connected in parallel to the chip.

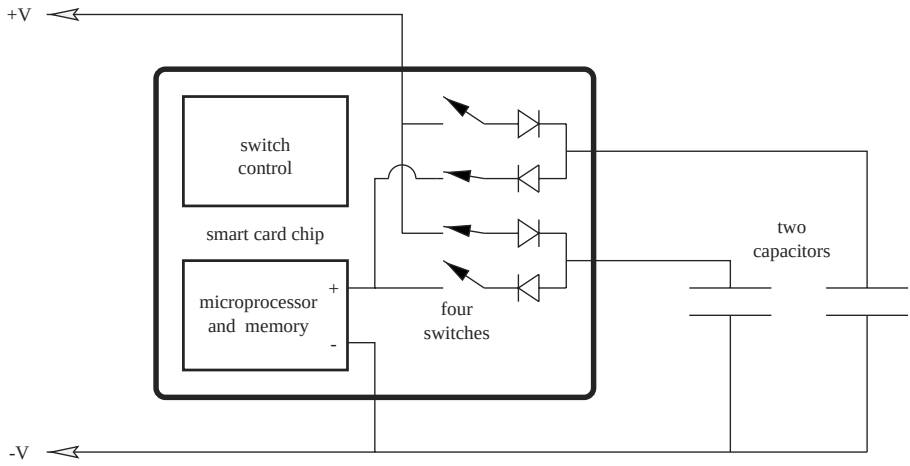


Fig. 2. Schematic diagram of a smart card with a detached power supply

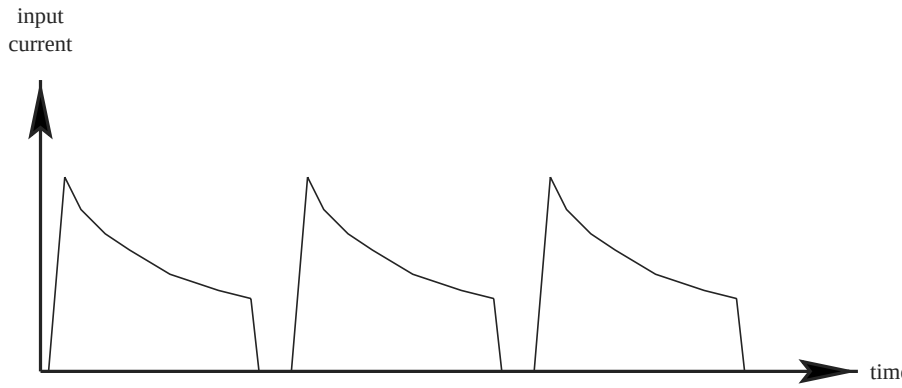


Fig. 3. The current supplied to smart cards with detached power supplies

The recommended size of each capacitor is about 0.1 microfarad. Low voltage capacitors of this type are commercially available in sizes as small as 2x2x0.4 mm, and thus they can be placed as external components next to the smart card chip in its plastic cavity. Alternatively, we can embed the capacitors in the card material itself by using alternate layers of plastic and aluminum in its 0.76 mm thickness and over its full surface area. Another possibility is to build the capacitors as extra metalic layers in the chip during its manufacturing process, but this would force the capacitor to be very small, and complicate the chip's manufacturing process. In large scale manufacturing, the addition of the two capacitors and the switch control adds just a few cents to the cost of the smart card.

An alternative design uses only one capacitor, which is alternately connected to the chip and to external power by two power transistors under the control of a simplified switch logic. The disadvantage of this approach is that the chip has to be halted and disconnected from power after each discharging cycle, which slows down its operation and can cause problems with some types of volatile on-chip memories.

The capacitor switchover should be triggered by counting a fixed number of instructions, rather than by comparing the dropping voltage of the discharging capacitor to some fixed threshold. The only information a passive attacker can infer is the total charge consumed by all the chip operations during the discharging period, which determines the initial current at the beginning of the next charging cycle. We can reduce this residual leakage by making the discharge period as long as possible. A simple calculation shows that a standard 0.1 microfarad capacitor can supply the 5 milliamperes required by a typical smart card chip for a period of 20 microseconds with a voltage drop of just 1 volt (say, from 6 volts to 5 volts). At the standard smart card clock rate of 5 megahertz, the chip performs about 100 instructions in this period, and thus the residual information which can be obtained by a passive attacker is the total power consumed by the chip during about 100 consecutive instructions. This is much less informative than the exact sequence of microcode events for each instruction, but it is still slightly vulnerable to DPA attacks on large numbers of sampled executions.

To make the smart card completely immune to passive power attacks, we have to add another simple element. Its role is to discharge the capacitor in an externally unobservable way to some fixed voltage after it is disconnected from the chip and before it is connected to the power supply (these are the intrapulse periods in Fig. 3). For example, the external power supply charges the capacitor from 4.5 to 6 volts, the chip discharges it during exactly 100 clock cycles to 5 ± 0.3 volts, and the switchover circuitry discharged it through an additional power transistor to exactly 4.5 volts during exactly 10 additional clock cycles before connecting it to the external power supply. In this case power measurements are completely useless, since the charging capacitors are always in exactly the same state at the same points in time regardless of the program executed or the data processed on the chip.

It is important to realize that power information can leak not only through the power lines, but also through the I/O line of the smart card chip which is used to send and receive data in a serial mode. This potential problem was ignored in most of the literature on power attacks, even though it can be used to attack chips whose power supplies were made immune to power attacks. In our proposed scheme, the voltage fluctuations of this line can leak information about the current power supplied by the capacitors. A simple solution to this problem is to disallow I/O operations (and temporarily ground or float the I/O line) during the execution of sensitive cryptographic subroutines.

The new capacitor approach is conceptually similar to the previously proposed battery approach, but it has the following important advantages:

- Capacitors are physically smaller than batteries, and are easier to embedded on the chip or in the plastic card next to the chip.
- Capacitors are cheaper than batteries, and cost just a few cents.
- Capacitors can be recharged an unlimited number of times, while batteries deteriorate after several hundred charging cycles.
- Capacitors do not have the memory effects of rechargeable batteries, and can be recharged without side effects even if they are not fully discharged.
- Capacitors can be charged in a fraction of a second, and thus intermittent use is not a problem.
- When we alternately charge and discharge capacitors, the average current consumed from the power supply is roughly equal to the average current consumed by the chip. Standard card readers may be unable to supply the large initial current needed if we want to charge the battery during the first second and then use it to power the chip for ten seconds.

The only disadvantage of the capacitor approach is that it can supply power to the chip only for several hundred clock cycles before its voltage becomes too low, and in each clock cycle the supplied voltage drops by about 0.01 volts. However, this voltage drop is not likely to interfere with the normal operation of the smart card chip, and we can repeatedly recharge the capacitors from the external power supply in order to execute an arbitrarily long computation.

Both the capacitor and the battery approaches are useless against an active attacker who can cut them off, replace them with other components, or measure the internal power consumption of the chip. However, the general problem of protecting smart cards in a cost effective way against active probing and microsurgery attacks seems to be currently unsolvable, and thus we do not try to address it in this paper.

References

BDL. D. Boneh, R. A. Demillo and R. J. Lipton, *On the Importance of Checking Cryptographic Protocols for Faults*, Proceedings of Eurocrypt 97, Springer-Verlag, 1997, pp 37-51.

K. P. Kocher, *Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems*, Proceedings of Crypto 96, Springer-Verlag, 1996, pp 104-113.

KK. O. Kommerling and M. Kuhn, *Design Principles for Tamper Resistant Smartcard Processors*, Proceedings of USENIX Workshop on Smartcard Technology, USENIX Association, pp. 9-20, 1999.
`http://www.cl.cam.ac.uk/~mgk25/sc99-tamper[-slides].pdf`.

KJJ. P. Kocher, J. Jaffe, and B. Jun, *Introduction to Differential Power Analysis and Related Attacks*, `http://www.cryptography.com/dpa/technical/index.html`, 1998.

Smartly Analyzing the Simplicity and the Power of Simple Power Analysis on Smartcards

Rita Mayer-Sommer

Electrical Engineering Division, ETH Zürich, Switzerland
rmayerso@ee.ethz.ch

Abstract. A new kind of cryptanalytic attacks, targeted directly at the weaknesses of a cryptographic algorithm's physical implementation, has recently attracted great attention. Examples are timing, glitch, or power-analysis attacks. Whereas in so-called *simple power analysis* (SPA for short) only the power consumption of the device is analyzed, *differential power analysis* (DPA) additionally requires knowledge of ciphertext outputs and is thus more costly. Previous investigations have indicated that SPA is little threatening and moreover easy to prevent, leaving only DPA as a serious menace to smartcard integrity. We show, however, that with careful experimental technique, SPA allows for extracting sensitive information easily, requiring only a single power-consumption graph. This even holds with respect to basic instructions such as register moves, which have previously not been considered critical. Our results suggest that SPA is an effective and easily implementable attack and, due to its simplicity, potentially a more serious threat than DPA in many real applications.

1 Introduction

It is the cryptanalyst's objective to obtain as much critical information as possible out of a cryptosystem, while keeping his effort and the risk of being detected at a minimum. In contrast to the design of a cryptographic algorithm, where security constitutes the central purpose, its ultimate physical implementation always depends on circuit implementation. Security aspects, as compared to efficiency, simplicity, or power consumption criteria, do still only play a marginal role in circuit design.

Kocher *et al.* [4] have proposed the following two kinds of so-called power-analysis attacks: simple power analysis (SPA), where the opponent tries to recover information about the secret key by simply measuring the power consumption of the computing device, and the more complex differential power analysis (DPA). Whereas the difficulty in SPA remains in the necessity for the attacker to know at which precise instant power consumption contains relevant information, DPA is more demanding in terms of the supplementary information needed. Above all, it requires a much larger number of experiments than does SPA.

In contrast to DPA, SPA merely requires the power consumption characteristics of one execution of the algorithm. However, SPA was previously considered

Ç.K. Koç and C. Paar (Eds.): CHES 2000, LNCS 1965, pp. 78–92, 2000.

unrealistic due to the destructive effect of noise deteriorating the measured signal. It was believed that, if anything at all, only conditional jump instructions testing for key bits might lead to successful SPA.

The results of our experiments, carried out on a smartcard-type microprocessor, stand in contrast to these beliefs and show that simple power analysis is an effective and easily implementable, hence serious, attack. This is even true for much simpler instructions than previously speculated, such as move operations, which cannot possibly be avoided by software countermeasures.

The outline of this paper is as follows. Section 2 provides an overview to the state of the art in power analysis and positions our results in this context (Section 2.4). In Section 3, we describe our experimental technique and the obtained results in detail. In Section 4, we draw the conclusions from the outcome of our experiments.

2 Power-Analysis Attacks to Cryptosystems

Power analysis is a physical attack to smartcard-based cryptosystems. It exploits the fact that the power dissipation of an electronic circuit depends on the actions performed in it. More specifically, the current flowing through the power lines of an operating microprocessor is dependent on the processed data. The following paragraphs describe and compare the types of power analysis that are currently examined. The hypotheses and results we discuss can be found in the recent publications [4], [6], [3].

2.1 Simple Power Analysis and Differential Power Analysis

Power analysis differs from most physical attack methods (see [1] and [2]) in many respects. First, it is not invasive and can thus be performed in a few instants; therefore, it can be used if a card-based action performed by an ordinary user is to be imitated by the eavesdropper, causing direct damage to the individual. Furthermore, the information side channel constituted by operation-related consumption can be accessed quite easily and without requiring a lot of specific knowledge about circuit or software implementation[1].

Those reasons make power analysis a type of attack which must be considered as a menace in case the eavesdropper is able to extract some information from the easily created side channel. Our objective has been to determine the amount of that information.

Kocher *et al.* [4] describe the two techniques which use the power-dissipation characteristics as a provider of side-channel information. Simple power analysis implies that the cryptanalyst measures the power consumption of the device operated during encryption or decryption, and evaluates the measured values (sampled at adequate instants, whose timing must be known or found by the

[1] Another question is how much information about the system is needed for exploiting the consumption information properly, but for now we just discuss accessibility of such a side channel.

attacker) directly, in order to correlate them with the key itself. On the other hand, differential power analysis is an attack which requires the availability of multiple power-consumption characteristics and ciphertexts out of a large number of diverse plaintext inputs.

The main advantage of DPA over SPA, apart from the fact that the attacker does not need to know implementational details of the target code (yet he must provide himself with all the other information necessary for performing DPA: a large number of ciphertexts and consumption graphs), is that the averaging process reduces the noise energy in the measured consumption signals. As the problem of extracting side-channel information from power-dissipation characteristics mainly lies in the many orders of magnitude between the absolute consumption values and the data-dependent differences between them, the influence of noise on SPA measurements can present an obstacle. Nevertheless, we could show that simple precautions in the measurement circuit, such as use of shielded cables and avoidance of ground loops, can raise the signal-to-noise ratio of the data obtained by SPA to an acceptable point.

The advantage of SPA over DPA is its low requirement in terms of amount of experiments and degree of device corruption. It certainly requires some insight into the structure of the implemented code, but extracting information about the program code with the help of microprobing tools is not a big obstacle for an experienced attacker (see [5] for details).

2.2 Physical Background of Simple Power Analysis

The power dissipation in CMOS cells such as logic gates, flip-flops, or latches mainly depends on changes of components' states rather than on the states themselves; e.g., for an inverter whose input voltage, applied to the connected gates of its cascaded PMOS and NMOS transistors, switches from high to low, the establishment of a transient short-circuit is induced. The rise of current in such a case is much larger than static dissipation. An in-depth analysis of short-circuit power consumption for a simple inverter cell is made in [7].

From these considerations, one might conclude that not the actual contents of the data bus, but rather the change in state of the internal registers from one instruction to the next would be measurable by power analysis. Nevertheless, our experiments enabled us to make both types of observations: the conductive properties of the data bus disclosed information about the absolute Hamming weight[2] of the transported data, and the rise in current induced by a change of state in internal registers was representative of the amount of bits that had changed in the data stored at this location, i.e., the transition counts[3]. Those two types of information are generated and retrieved independently, and combining them for cryptanalytic means is definitely interesting.

[2] The Hamming weight of a binary string is the number of ones that occur in it.

[3] The transition count between two consecutively processed data strings is the Hamming weight of their pointwise XOR-sum.

2.3 Previous Results on Simple Power Analysis

Messerges, Dabbish, and Sloan [6] specifically indicate the dangers of SPA and DPA on the DES encryption process. They provide an estimate on the reduction of the brute-force search space for the eight DES key bytes, for the case where the Hamming weights of these eight bytes are given, and also for the case where additionally the eight parity bits are known: without any supplementary information, there are 2^{56} possibilities, compared to about 2^{45} in the first and about 2^{38} in the second case.

They acknowledge that finding out the Hamming weights alone may be of little help, especially when larger keys than in DES are employed, but that this type of knowledge can get quite useful as soon as the key bytes are shifted, like during DES encryption.

In [6], the dangers are mentioned that may arise from knowledge of transition counts between key data and the data bus' contents previously to key data being transferred onto it. It is indicated that an attacker might easily find out what was written on the bus right before it was loaded with the crucial key byte, because this data is usually some fixed address or an instruction opcode. In this concern, our cryptanalytic methods – observing different but comparable processes (as are typical for execution of the encryption rounds), possibly separated by many instructions, and extracting the differences between them – take a different turn. What is proposed in [6] is an "instantaneous" analysis where real-time transitions are observed and evaluated. In [6], it is claimed that in such a case, the attacker requires some detailed knowledge about the source code of the algorithm's implementation (more precisely, that not only the code structure, but also the addresses of accessed registers and memory have to be known[4]).

The measurements exposed in [6] reflect the change in power dissipation when a bus which at first contains a memory address is loaded with various data values. Our results extend those measurements by showing that it is not necessary, nor at all helpful, to know storage addresses in order to find absolute Hamming weight values of data.

In our view, it must be proven that Hamming weight information for key bytes can be found by SPA. The two papers we discussed so far make this assumption and affirm that in principle "it can be done". Yet, they also claim that SPA is only possible for conditional branching instructions. Still, the corresponding quantitative results are not exposed. This triggered our desire to determine how effectively Hamming weights, and not merely transition counts, really can be found.

[4] We observe that those addresses could be generated randomly for every single smartcard (a kind of "fingerprint" addressing); to find out which addresses a certain card uses, it would thus be necessary to extract its specific source code – not always an easy task. It is more likely that the attacker is merely informed about the general structure of the code.

Biham and Shamir [3] propose a method which enables the attacker to identify the key-scheduling process during encryption by simple power measurements, when he has no access to information about algorithm implementation or timing. They show that the reasons for the vulnerability of DES, SAFER, and SAFER+ to this attack are uneven cyclic shifts and the way original key bits are grouped into subkey bytes. The authors found that during the key-scheduling process in DES, knowledge of the Hamming weights of the subkey bytes provides the key in a direct manner.

An important point is the implicit statement in [3] that SPA is an attack at least as dangerous as DPA if the cryptographic algorithm is designed in a way which makes it vulnerable to an attacker who has Hamming-weight information (and not just in the case where the implementor of the algorithm is not cautious about power analysis, and may create conditional jumps testing for key bits).

2.4 Our Results

We implemented the very simplest kind of power-analysis attack by observing the chip's power dissipation directly. Our main aim hereby was the extraction of information about data arguments of instructions. The method we employed to obtain maximum information from a microprocessor's power consumption characteristic was to compare a number of data-related processes identical except for one of the data or instruction properties we wished to examine (e.g., Hamming weight or Hamming-weight change, transition count, or absolute value of data or storage location; types of instructions or contents of instruction arguments; number of bits changing from high to low and inversely; more generally, the different types of changes that may take place). Then, we had to find out which of those properties could be at the origin of the observed variations.

In [4], it is claimed that SPA is easily made impossible by avoiding the use of key bits in conditional branching or jump instructions, whose dissipation characteristics distinguish themselves clearly from other operations. Yet, our results, obtained without involving conditional branching on sensitive data, indicate that even simple move instructions can reveal critical information.

Additionally, our results show that if the device is operated at sufficiently low frequency and high supply voltage[5], it is not even necessary to average noise out of the consumption characteristics in order to obtain key information. This implies that indeed a single experiment delivers enough insight to obtain key-relevant information.

Although DPA can represent a powerful attack on cryptosystems, as it directly aims at obtaining key bits, it is not necessarily a "very low cost" attack in the sense of easy feasibility. As the menace constituted by an attack is inversely proportional to the expense required for its performance, successful SPA should be regarded as especially dangerous due to its low cost.

[5] In an SPA scenario, operating frequency and supply voltage are considered to be under the control of the attacker.

3 Experimental Method

We chose the PIC16C84 chip [8] as the processor for our experiments, which were run at 4 MHz and 4.5 V supply voltage. This particular processor is similar in structure to most of the microprocessors in use for smartcard systems.

The method we used to investigate data dependency of the PIC's power dissipation was to design test routines in the processor's assembly language, making it perform certain instructions with varying data arguments. We then acquired the power-consumption characteristics generated during program execution; in the next step, we found "zones" of high correlation between data and consumption, which we further investigated. The conclusions drawn from these investigations constitute the results of our query.

3.1 Data Dependency in Move Instructions

In order to evaluate changes in power dissipation due to writing different data values into a certain memory location or register, we executed the following assembly-language program (see [8] for details) as an infinite loop:

```
; define registers VAL, PORTB, PORTA, REG:
VAL    equ 0x08
PORTA  equ 0x05
PORTB  equ 0x06          ; PORTA, PORTB: output ports
REG    equ 0x0c

start
      clrf    REG
      movlw   D'255'
      movwf   VAL        ; 0: move 255 to source value register

loopstart
      movfw   VAL, 0     ; 1: move new value to accumulator
      nop                ; 2
      nop                ; 3
      movwf   REG        ; 4: ! move value from accumulator
      nop                ; 5           to internal register !
      nop                ; 6
      movwf   PORTB      ; 7: move value to PORTB
      bsf     PORTA, 0   ; 8: set strobe bit (LSB of PORTA)
      bcf     PORTA, 0   ; 9: clear strobe bit
      clrf    PORTB      ;10: clear data in port B
      decfsz  VAL        ;11: decrease value,
      goto    loopstart  ;12           back to loopstart if !=0
      decf    VAL        ;13: set value to 255
      goto    start
```

This way, the numbers which are consecutively written into `REG` range from `255` to `0` (cf. instruction 4), decreasing by steps of one. We also examined variants of this program, where the transferred values were either increased by steps of 3, or decreased by steps of 1. Additionally, we ran the same program structure, replacing the `movwf` in instruction 4 by other commands combining moves and logical operations.

3.2 Finding Data Dependency

We acquired analog data representing the processor's power dissipation, sampled at 200 MHz (i.e., 50 samples per card cycle), by measuring the voltage over a probing resistor connected between the microprocessor's ground pin and the overall circuit's ground. We then wanted to find the instants during execution of the previously described program loops where data dependency of power consumption could be observed. At this point, assumptions were drawn as to what changes in which properties of data could induce measurable variations in power dissipation.

We investigated the data dependency of the acquired voltage samples in the following way: for every data value (as we examined an eight-bit processor, those values range from 0 to 255), one loop of the test program was run. Data dependency is likely to occur at several instants, e.g., those where data is written to the output ports of the processor; yet, this process is much less interesting than the write operation triggered by the `movwf`-instruction, which transfers a value from the accumulator to one of the internal registers, or inversely. In order to see clearly at what exact instant during the execution of this command the dependency between power dissipation and data is maximal, a correlation factor (correlating the measured values to the investigated data properties) was computed for every list of length 256, containing a measure of power dissipation at a certain stage of the loop execution for every data value. Thus, for every sampling moment k (the range of k is dependent on the number of assembly instructions per loop and sampling rate) during the loop execution, there exists a list v_k,

$$v_k = [v_k(0), v_k(1), \ldots, v_k(255)],$$

where $v_k(j)$ is the voltage measured over the probing resistor at the moment of the k^{th} sample during execution of the loop with data argument j. We are now interested in correlations between $v_k(j)$ and certain properties $p(j)$ of the data j for fixed k.

There are various ways to compute correlations between two quantities. For instance, one might be in the situation of wanting to evaluate the degree of correlation between two sets of data samples with an unknown joint distribution. However, in the present case, numerous data sets are compared against one another, so the relative rather than the absolute value of correlation is interesting; we were primarily interested in detection of local maxima. Therefore, setting the average current consumption at the moment of the k^{th} sample and the average of the investigated data property over all data arguments to $\overline{v_k}$ and $\overline{p}$ respectively,

we simply computed the Pearson correlation factor r_k,

$$r_k = \frac{\sum_j (v_k(j) - \overline{v_k}) \cdot (p(j) - \overline{p})}{\sqrt{\sum_j (v_k(j) - \overline{v_k})^2} \cdot \sqrt{\sum_j (p(j) - \overline{p})^2}}$$

and compared the correlation factors for different moments k during program loop execution.

In accordance with our expectations, we found that correlation between power dissipation and Hamming weight of processed data indeed occurs. This fact can be stated after inspection of the different correlation graphs which have been drawn with respect to direct data, Hamming weight, and transition counts. The second type of correlation graph contains valuable information in the sense of clear peaks indicating high correlation during the `movwf`-instruction. In addition, inspecting the transition-counts correlation graph we found that in the same instruction, there is also an instant where power consumption is proportional to the number of bits that were inverted from the previously transferred data value to the current one. But we may not yet be led to the conclusion that Hamming weights and transition counts are the only data properties correlated to consumption; it is always possible that we "overlooked" certain other kinds of correlation.

A typical graph of correlation values with respect to Hamming weight is shown in Figure 1. From the correlation graph, we extracted local peaks, indicating that "something interesting" might be happening at the k^{th} instant during loop execution. We then inspected the corresponding v_k and evaluated if indeed data dependency could be observed. Examples of typical v_k's, extracted in the described manner, are given in Figures 2-10.

3.3 Noise Level of the Acquired Signals

Figures 2-10 indicate the striking similarities between the components of v_k and the Hamming weights (Fig. 2-4, 7-10) or the transitions counts (Fig. 5, 6) of the sequence of processed data. Yet, the visualization of this similarity is just an intuitive hint that Hamming information is leaked; the noise level of the obtained signal still had to be examined.

Thus, we now evaluated whether the quality of the extracted measurement data was at all high enough for making assumptions about Hamming weights of the processed instruction arguments. In the given case, this evaluation primarily consisted of the question whether the data consumption values could be grouped in a unique manner, so that they would form clusters of points which could be assigned a single Hamming weight, and whether the noise induced by the measurement was low enough in order to make those attributions in a correct manner.

We made the separation into nine clusters of points and observed that the averages of every cluster are separated by voltages of about $\Delta V \approx 5\,\text{mV}$. Those cluster distances remain constant for all Hamming-weight values except for zero,

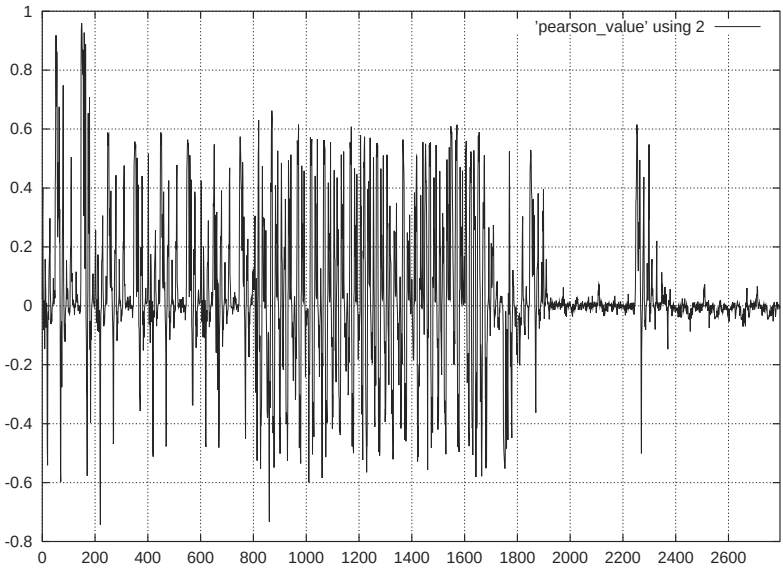


Fig. 1. Pearson correlation factors during program loop execution. Variation of correlation factors can be observed during the last five loop instructions: two nop's, movwf[data], two nop's (200 samples per instruction).

which induces much lower consumption. Thus, the maximum admitted noise level is $n_{max} = \Delta V/2$; every noise contribution higher than this will lead to an erroneous conclusion about the Hamming weight of processed data.

In our experiments, we were indeed able to locate v_k's where Hamming weight attribution could be done in an unequivocal manner. For a real attacker things are different: unless he already knows the timing of the investigated process, he cannot find the proper instant k without the help of correlations. Yet, we found that power dissipation at this crucial instant (where best indication of Hamming weights is given) is characterized by maximal correlation between current consumption and data, minimal distortion of power consumption by processes other that the loading of key bytes on the internal data bus, and minimum variance among the clusters of consumption samples. Thus, even if the attacker is a priori unable to locate the desired instant k, he might reach this aim by using those properties of the acquired data.

4 Concluding Remarks

We have shown that SPA can be done with extremely simple infrastructure and adequate experimental technique. Even basic assembly instructions such as register moves provide information about Hamming weights of on-bus data *and* transition counts between data items written into memory locations or registers.

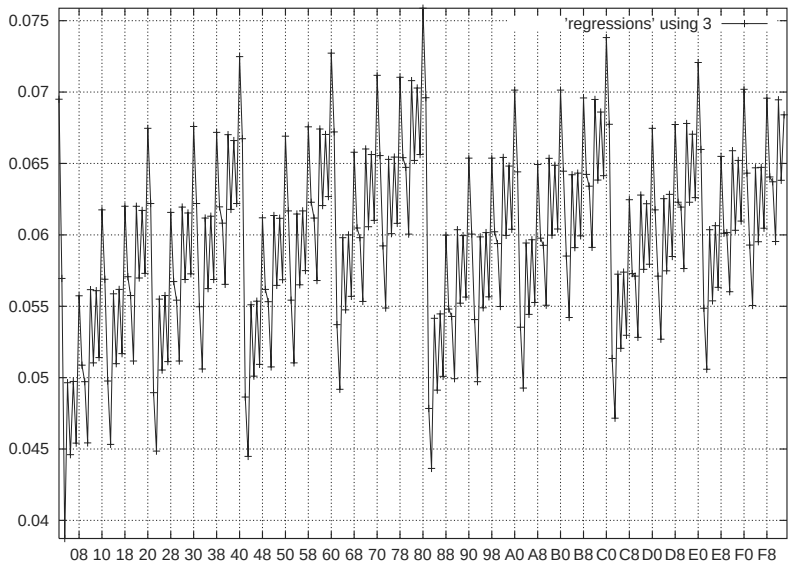


Fig. 2. Voltage values for 256 loop executions. Here, v_k is extracted at the instant of highest correlation between Hamming weight of the data sequence and power dissipation (during instruction 4). x-axis: data values j; y-axis: $v_k(j)$ in [V]. A zoom-in is shown in Figure 3.

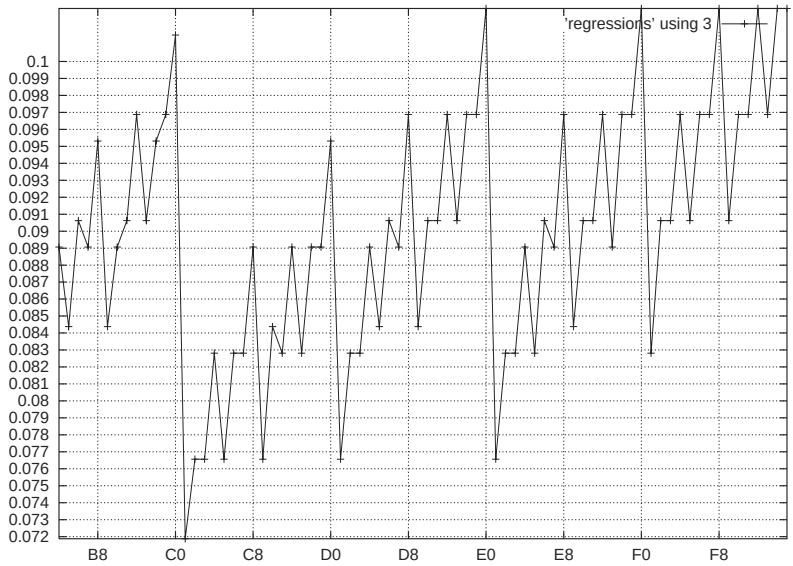


Fig. 3. The values $v_k(j)$ for j ranging from 180 to 256, increasing by steps of 1.

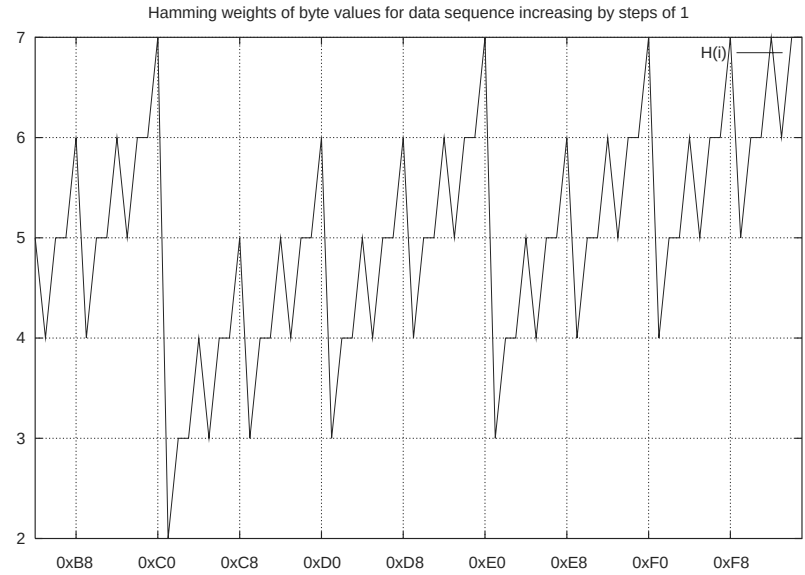


Fig. 4. For comparison, this figure gives the computed Hamming weights of the data processed in the investigated loops.

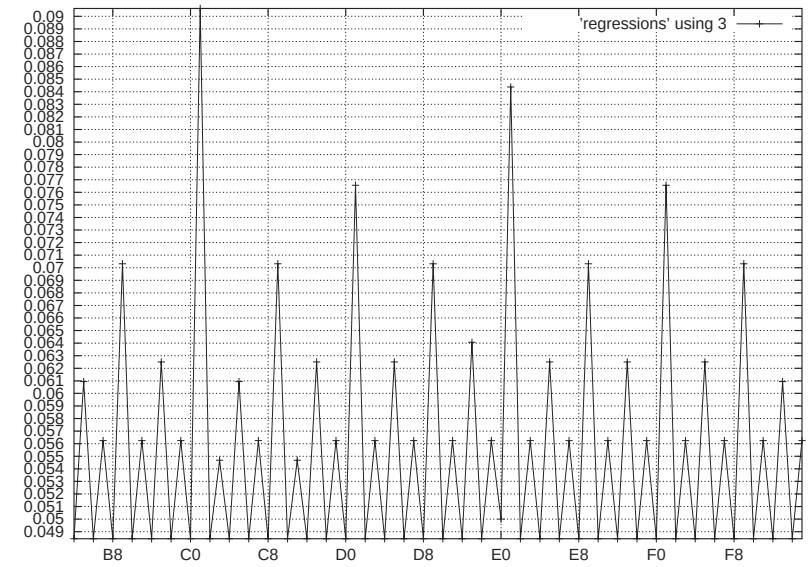


Fig. 5. Measured voltage values at instants of highest correlation between transition counts and power consumption, for j going from 180 to 255, increasing by steps of 1.

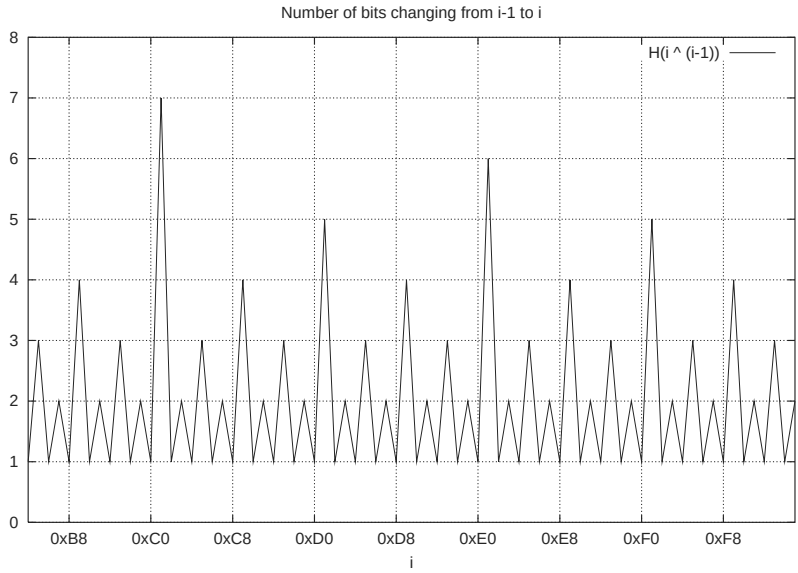


Fig. 6. Computed transition counts for data going from 180 to 255, increasing by steps of 1.

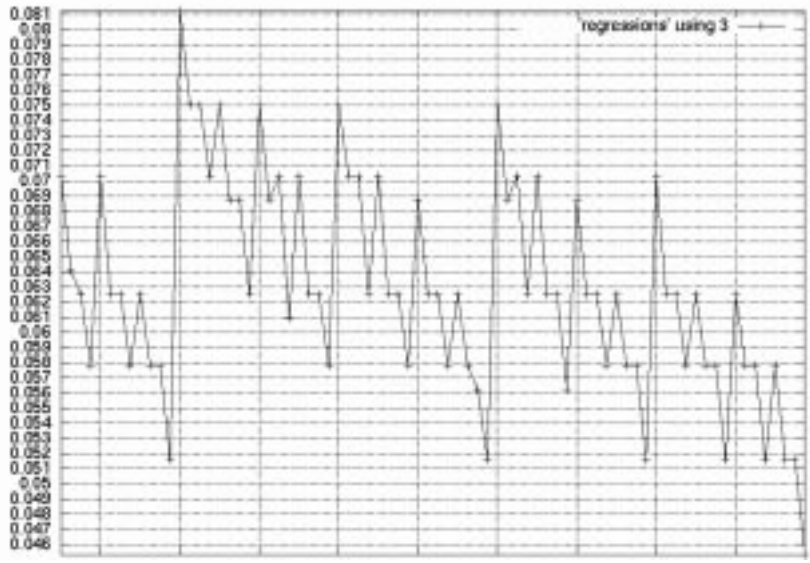


Fig. 7. The values $v_k(j)$ for data j transferred by the move-instruction ranging from 76 down to 0, decreasing by steps of 1.

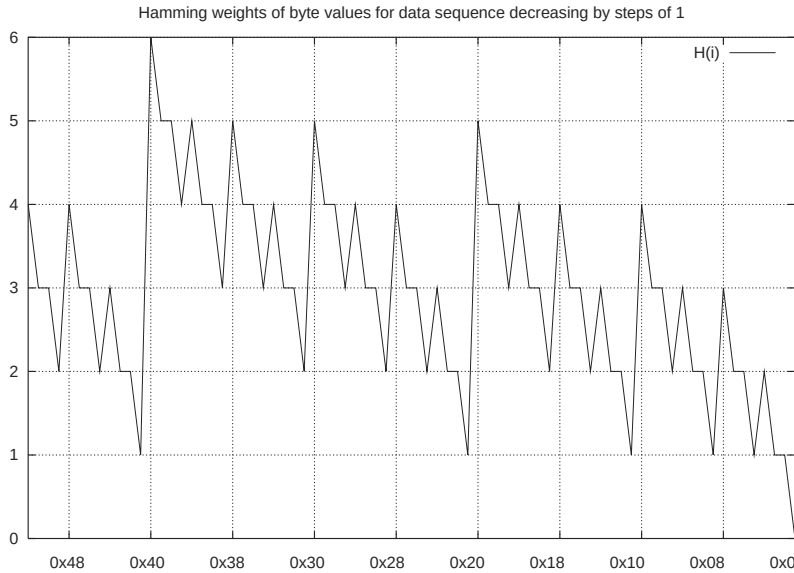


Fig. 8. Computed Hamming weights for data ranging from 76 down to 0, decreasing by steps of 1.

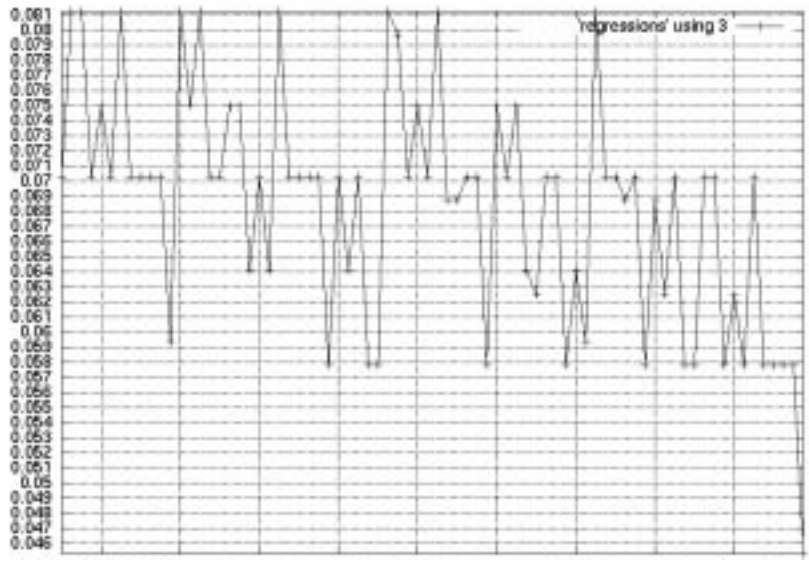


Fig. 9. Measured voltage values for transferred data j ranging from 228 to 0, decreasing by steps of 3.

Fig. 10. Computed Hamming weights for data ranging from 228 to 0, decreasing by steps of 3.

The SPA attacker does not require conditional jumps on sensitive data in order to obtain this information, contrary to what was supposed until now. Using appropriate noise shielding, we could show that a single experiment suffices in order to draw the desired key information from the power consumption of a smartcard processor performing cryptographic operations.

When examining instructions other than `movwf`, we have observed that the extracted values sequence is identical, and hence independent of the instruction type; thus, `iorwf, xorwf, rrf, subwf` all yield resembling v_k's when correlated to Hamming weight and transition count. In order to explain this, we indicate that the "crude" data is transferred over the internal bus before getting involved with mathematical or logical operations in the ALU. This data transfer is, at certain instants, the sole reason for characteristic power consumption values; whatever takes place inside the arithmetic and logic unit of course causes data- and operation-dependent power dissipation, but it is not easily possible nor at all necessary to analyze the power consumption of this type of activity.

It was our objective to find out whether Hamming weights and transition counts are really yielded by SPA, and the question can be answered by a clear yes. Even if we assume that nothing but what we found can be found at all, this still is a menace to smartcard holders' security if the attacker is able to synchronize with the implemented software.

Acknowledgments

This research was carried out during the author's stay with the Security Group, Computer Lab of Cambridge University, UK. The author would like to thank Ross Anderson, Stefan Wolf, Markus Kuhn, Anna Lysyanskaya, and Hubert Kaeslin for stimulating discussions and proofreading, and the anonymous referees for helpful comments. This project was supported by NDS Technologies Israel Ltd.

References

1. R. J. Anderson and M. G. Kuhn, Low Cost Attacks on Tamper Resistant Devices, *5th International Workshop on Security Protocols*, 1997.

2. R. J. Anderson and M. G. Kuhn, Tamper Resistance - a Cautionary Note, *Proceedings of the Second USENIX Workshop on Electronic Commerce*, 1996.
3. E. Biham and A. Shamir, Power Analysis of Key Scheduling of the AES Candidates, *Proceedings of the Second AES Candidate Conference*, 1999.
4. P. Kocher, J. Jaffe, and B. Jun, Differential Power Analysis, *Advances in Cryptology – Proceedings of CRYPTO '99*, Lecture Notes in Computer Science, Vol. 1666, Springer-Verlag, 1999.
5. M. Kuhn and O. Kömmerling, Design Principles for Tamper-Resistant Smartcard Processors, *Proceedings of the USENIX Workshop on Smartcard Technology*, 1999.
6. T. S. Messerges, E. A. Dabbish, and R. H. Sloan, Investigations of Power Analysis Attacks on Smartcards, *Proceedings of the USENIX Workshop on Smartcard Technology*, 1999.
7. H. J. M. Veendrick, Short-Circuit Dissipation of Static CMOS Circuitry, *IEEE Journal of Solid-State Circuits*, Vol. SC-19, No.4, 1984
8. Datasheet of the PIC16C84 Processor, available at `http://www.microchip.com`.

Power Analysis Attacks and Algorithmic Approaches to their Countermeasures for Koblitz Curve Cryptosystems

M. Anwar Hasan

Department of Electrical and Computer Engineering
University of Waterloo, Waterloo, Ontario, N2L 3G1, Canada
ahasan@ece.uwaterloo.ca

Abstract. Because of their shorter key sizes, cryptosystems based on elliptic curves are being increasingly used in practical applications. A special class of elliptic curves, namely, Koblitz curves, offers an additional but crucial advantage of considerably reduced processing time. In this article, power analysis attacks are applied to cryptosystems that use scalar multiplication on Koblitz curves. Both the *simple* and the *differential* power analysis attacks are considered and a number of countermeasures are suggested. While the proposed countermeasures against the simple power analysis attacks rely on making the power consumption for the elliptic curve scalar multiplication independent of the secret key, those for the differential power analysis attacks depend on randomizing the secret key prior to each execution of the scalar multiplication.

1 Introduction

If cryptographic systems are not designed properly, they may leak information that is often correlated to the secret key. Attackers who can access this leaked information are able to recover the secret key and break the cryptosystem with reasonable efforts and resources. In the recent past, attacks have been proposed that use the leaked side channel information such as timing measurement, power consumption and faulty hardware (see for example [1], [2], [3], [4], [5], [6]). These attacks are more related to the implementation aspects of cryptosystems and are different from the ones that are based on statistical properties of the cryptographic algorithms (i.e., differential and linear cryptanalysis attacks [7], [8], [9]).

In [2] and [4], Kocher et al. have presented attacks based on *simple* and *differential* power analysis (referred to as SPA and DPA respectively) to recover the secret key by monitoring and analyzing the power consumption signals. Such power signals can provide useful side channel information to the attackers. In [10], Kelsey shows how little side channel information is needed by an attacker to break a cryptosystem. In [3], Messerges et al. show how the side channel information can be maximized. In order to implement a good cryptosystem, the designer needs to be aware of such threats.

Ç.K. Koç and C. Paar (Eds.): CHES 2000, LNCS 1965, pp. 93–108, 2000.

In [11], a number of power analysis attacks against smartcard implementations of modular exponentiation algorithms have been described. In [12], power analysis attacks have been extended to elliptic curve (EC) cryptosystems, where both the SPA and the DPA attacks have been considered and a number of countermeasures including private key randomization and EC point blinding have been proposed. Generic methods to counteract power analysis attacks have been reported in [13].

Cryptosystems based on a special class of ECs, referred to as anomalous binary or Koblitz curves, were proposed by Koblitz in [14]. Such cryptosystems offer significant advantage in terms of reduced processing time. The latter, along with shorter key sizes, has made Koblitz curve (KC) based cryptosystems attractive for practical applications. However, the countermeasures available for random EC based cryptosystems do not appear to be the best solution to KC based cryptosystems.

In this article power analysis attacks are investigated in the context of KC based cryptosystems. The SPA attack is considered and its countermeasures at the algorithmic level are given. The proposed countermeasures rely on making the power consumption for the elliptic curve scalar multiplication independent of the secret key. Cryptosystems equipped with such countermeasures are however not secure enough against a stronger attack based on the DPA. In this article we also consider the DPA attack and describe how an attacker can maximize the differential signal used for the power correlation. To prevent DPA attacks against KC cryptosystems, we suggest a number of countermeasures which are the main results of this article. These countermeasures depend on randomizing the secret key prior to each execution of the scalar multiplication. They are suitable for hardware implementation, and compared to the countermeasures available in the open literature, their implementation appears to be less complex.

2 Preliminaries

An elliptic curve (EC) is the set of points satisfying a bivariate cubic equation over a field. For the finite field $\mathrm{GF}(2^n)$ of characteristic two, the standard equation for an EC is the Weierstrass equation

$$y^2 + xy = x^3 + ax^2 + b \tag{1}$$

where $a, b \in \mathrm{GF}(2^n)$ and $b \neq 0$. The points on the curve are of the form $P = (x, y)$, where x and y are elements of $\mathrm{GF}(2^n)$. Let E be the elliptic curve consisting of the solutions (x, y) to equation (1), along with a special point $\mathcal{O}$ called the point at *infinity*. It is well known that the set of points on E forms a commutative finite group under the following addition operation. (More on it can be found in [15], [16], [17], [18], and [19].)

Elliptic Curve Addition: Let $P = (x, y) \neq \mathcal{O}$ be a point on E. The inverse of P is defined as $-P = (x, x + y)$. The point $\mathcal{O}$ is the group identity, i.e., $P \uplus \mathcal{O} = \mathcal{O} \uplus P = P$, were $\uplus$ denotes the elliptic curve group operation (i.e.,

addition). If $P_0 = (x_0, y_0) \neq \mathcal{O}$ and $P_1 = (x_1, y_1) \neq \mathcal{O}$ are two points on E and $P_0 \neq -P_1$, then the result of the addition $P_0 \uplus P_1 = P_2 = (x_2, y_2)$ is given as follows.

$$x_2 = \begin{cases} \left(\frac{y_0+y_1}{x_0+x_1}\right)^2 + \frac{y_0+y_1}{x_0+x_1} + x_0 + x_1 + a, & P_0 \neq P_1, \\ x_0^2 + \frac{b}{x_0^2}, & P_0 = P_1, \end{cases} \tag{2}$$

$$y_2 = \begin{cases} \left(\frac{y_0+y_1}{x_0+x_1}\right)(x_0 + x_2) + x_2 + y_0, & P_0 \neq P_1, \\ x_0^2 + \left(x_0 + \frac{y_0}{x_0}\right) x_2 + x_2, & P_0 = P_1. \end{cases} \tag{3}$$

The above formulas for the addition rule require a number of arithmetic operations, namely, addition, squaring, multiplication and inversion over GF(2^n). (See, e.g., [20] and [21], for efficient algorithms for finite field arithmetic.) The computational complexities of addition and squaring are much lower than those of multiplication and inversion. To simplify the complexity comparison, our forthcoming discussion in this article ignores the costs of addition and squaring operations. Also, note that the formulas in (2) and (3) for point doubling (i.e., $P_0 = P_1$) and adding (i.e., $P_0 \neq P_1$) are different. The doubling requires one inversion, two *general* multiplications and one constant multiplication, whereas the adding operation costs one inversion and two general multiplications. Since, a field inverse is several times slower than a constant multiplication, we assume that the costs of elliptic curve point doubling and adding are roughly equal. If the points on E are represented using projective coordinates, one can however expect to see a considerable difference in these two costs and needs to treat them accordingly.

Elliptic Curve Scalar Multiplication: Elliptic curve scalar multiplication is the fundamental operation in cryptographic systems based on ECs. If k is a positive integer and P is a point on E, then the scalar multiplication kP is the result of adding k copies of P, i.e.,

$$kP = \underbrace{P \uplus P \uplus \cdots \uplus P}_{k \text{ copies}},$$

and $-kP = k(-P)$. Let $(k_{l-1},\ k_{l-2},\ \cdots,\ k_1,\ k_0)_r$ be a radix r representation of k, where k_{l-1} is the most significant symbol (digit) and each k_i, for $0 \leq i < l-1$, belongs to the symbol set s used for representing k. Thus

$$\begin{aligned} kP &= \left(\sum_{i=0}^{l-1} k_i r^i\right) P = (k_{l-1} r^{l-1} P) \uplus \cdots \uplus (k_1 r P) \uplus (k_0 P) \\ &= r\,(r\,(\cdots r\,(r(k_{l-1}P) \uplus k_{l-2}P) \uplus \cdots)\,k_1 P) \uplus k_0 P. \end{aligned}$$

Then one may use the following *multiply-radix-and-add* algorithm (also known as double-and-add algorithm for $r = 2$) to compute kP in l iterations.

Algorithm 1. Scalar multiplication by multiply radix and add
Input: k and P
Output: $Q = kP$

```
Q := O
for (j = l − 1; j ≥ 0; j − −) {
    Q := rQ
    if (k_j = 1)
        Q := Q ⊎ P
}
```

Remark 1. For practical purposes one can assume that k is represented with respect to the conventional binary number system (i.e., $r = 2$ and $k_j \in \{0, 1\}$) and that $l = n$ where n is the dimension of the underlying extension field. Then the above algorithm would require approximately $3n/2$ elliptic operations on average.

Note that the conventional binary system is non-redundant and k has only one representation. However, using a different number system which has redundancy in it, the integer k can be represented in more than one way. By choosing a representation of k that has fewer non-zeros, one can reduce the number of EC additions and hence speed-up the scalar multiplication. More on this can be found in [22] and the references therein.

Remark 2. If k is represented in the binary NAF (non-adjacent form [22]), where $r = 2$, $k_j \in \{-1, 0, 1\}$ and $k_j k_{j+1} = 0$, $0 \le j \le n$, then the average number of elliptic operations in Algorithm 1 is $\approx 4n/3$.

Koblitz Curves: In (1), if we set $b = 1$ and restrict a to be in $\{0, 1\}$, we have

$$y^2 + xy = x^3 + ax^2 + 1, \tag{4}$$

which gives a special class of ECs, referred to as Koblitz curves (KC). Let us denote the KC as E_a. (In the rest of this article, if a curve E as defined in conjunction with (1) is not of Koblitz type, then it is referred to as a random curve.)

In (4), since $a \in \mathrm{GF}(2)$, if (x, y) is a point on E_a, (x^2, y^2) is also a point on E_a. Using the addition rule given in the previous section, one can also verify that if $(x, y) \in E_a$, then the three points, viz., (x, y), (x^2, y^2) and (x^4, y^4) satisfy the following:

$$(x^4, y^4) \uplus 2(x, y) = (-1)^{1-a}(x^2, y^2). \tag{5}$$

Using (5), one can then obtain

$$\tau(x, y) = (x^2, y^2), \tag{6}$$

where, τ is a complex number which satisfies

$$\tau^2 - (-1)^{1-a}\tau + 2 = 0. \tag{7}$$

Equation (6) is referred to as the Frobenius map over GF(2). One important implication of (6) is that the multiplication of a point on E_a by the complex number τ can simply be realized with the squaring of the x and y coordinates of the point. As a result, if the scalar k is represented with radix τ and $k_i \in \{0, 1\}$, the above multiply-radix-and-add algorithm still can be used with $r = \tau$. The operation $Q := rQ$ would however then correspond to two squaring operations over GF(2^n). In a *normal* basis representation, squaring is as simple as a cyclic shift of the bits of the operand. Efficient squaring algorithms using the more widely used *polynomial* basis can be found in [23] and [20].

3 SPA Attack and Its Countermeasures

The elliptic curve scalar multiplication $Q = kP$, where both P and Q are points on the curve and k is an integer, is the fundamental computation performed in cryptosystems based on elliptic curves. In the elliptic curve version of the Diffie-Hellman key exchange, the scalar k is the *private* key which is an binary integer of about n bits long. The security of many public-key cryptosystems depends on the secrecy of the private key. In many applications, the key is stored inside the device. In the recent past, attacks have been reported in the open literature to recover the key by analyzing the power consumption of cryptosystems. Following [12], here we first briefly describe the simple power analysis (SPA) attack and a countermeasure against it (denoted as *simple countermeasure*). Although the countermeasure, which is a close-variant of its counterpart in modular exponentiation, is easy to implement, below we show that its straight-forward implementation gives away the computational advantage one would expect from the use of Koblitz curves. We then discuss two simple modifications for possible improvements.

3.1 SPA Attack

In general, power analysis attacks rely on the difference between power consumptions of the cryptosystem when the value of a specific partitioning function is above and below a suitable threshold. For example, when a cryptosystem is performing a simple operation (such as the Frobenius map), the power consumption may be related to the Hamming weight of the operand. Large differences in power consumptions may be identified visually or by simple analysis.

If the scalar multiplication is performed using the *multiply-radix-and-add* algorithm, a simple power analysis can be applied. In Algorithm 1, the operation $Q := rQ$ is performed in each iteration irrespective of the value of k_j. However, the step with $Q := Q \uplus P$ is processed if $k_j = 1$, which requires a number of time and power consuming operations, such as, GF(2^n) multiplication and inversion as shown in (2) and (3). This enables an attacker to easily analyze the power consumption signals, especially to detect the difference in power consumption (and time) and to eventually recover k_j. An attacker may need as low as one iteration of the multiply-radix-and-add algorithm to obtain k_j.

3.2 Coron's Simple Countermeasure

A straightforward countermeasure for the SPA attack is to make the execution of the elliptic curve addition independent of the value of k_j. This can be achieved by performing the elliptic addition in each iteration irrespective of the value of k_j and use the sum in the subsequent steps as needed. This is shown in the following algorithm. The latter, along with r replaced by 2, yields the SPA resistant scalar multiplication for random curves proposed by Coron [12].

Algorithm 2. SPA resistant scalar multiplication
Input: k and P
Output: $Q = kP$

```
for (j = l − 1; j >= 0; j − −){
    Q[0] := rQ[0]
    Q[1] := Q[0] ⊎ P
    Q[0] := Q[k_j]
}
Q:=Q[0]
```

Assuming that the difference in power consumption to access $Q[0]$ and $Q[1]$ is negligible, the power consumption for executing $Q[0] := Q[k_j]$ (and hence the above algorithm) does not depend of the value of k_j. As a result, the simple power analysis attack would not be effective to recover k.

As mentioned earlier, for the binary representation of k, the latter can be n bits long implying that the above algorithm would require $2n$ elliptic operations (doubling and adding). On the other hand, to take advantage of the simple Frobenius mapping associated with the Koblitz curve, k is usually represented with radix $r = \tau$. For such τ-adic representation, if we limit k_j, for $0 \leq j \leq l-1$ to be 0 and 1 only, then the value of l in Algorithm 2 can be $\approx 2n$ [24]. Thus, the algorithm would require about $2n$ elliptic operations (only addition, no doubling) and does not appear to provide computational advantages of using Koblitz curves over random curves. The following discussion however attempts to alleviate this problem.

3.3 Reduced Complexity Countermeasure

Since the solutions (x, y) to equation (4) are over GF(2^n), we have $x^{2^n} \equiv x$. Consequently,

$$\begin{aligned} \tau^n(x, y) &= (x^{2^n}, y^{2^n}) \equiv (x, y) \\ (\tau^n - 1)(x, y) &\equiv \mathcal{O}. \end{aligned} \tag{8}$$

Thus, for the scalar multiplication $Q = kP$, instead of using k, one can use $k \pmod{\tau^n - 1}$ and an n-tuple can be used to represent $k \pmod{\tau^n - 1}$ in radix τ. Let $\kappa = (\kappa_{n-1}, \kappa_{n-2}, \cdots, \kappa_0)_\tau$ denote the τ-adic representation of $k \pmod{\tau^n - 1}$, where $\kappa_i \in s$. The latter corresponds to the set of symbols used for representing the reduced k. Efficient algorithms exist to reduce k modulo $\tau^n - 1$

(see for example [24], [25]). As it will be shown later, in certain situations, it appears to be advantageous to use an expanded symbol set which results in a redundant number system. Assume that $s = \{s_0, s_1, \cdots, s_{|s|-1}\}$ with $s_0 < s_1 < \cdots < s_{|s|-1}$. For the sake of simplicity, if we also assume that s is symmetric around zero (e.g., $s = \{-1,\ 0,\ 1\}$), the following algorithm for $Q = kP$ is SPA resistant.

Algorithm 2a. SPA resistant scalar multiplication with reduced τ representation
Input: k and P
Output: $Q = kP$

```
for (j = n − 1; j >= 0; j − −){
    Q[0] := τQ[0]
    Q[1] := Q[0] ⊎ P;    Q[−1] := −Q[1]
    Q[2] := Q[1] ⊎ P;    Q[−2] := −Q[2]
    ⋮
    Q[(|s| − 1)/2] := Q[(|s| − 3)/2] ⊎ P;    Q[−(|s| − 1)/2] := −Q[(|s| − 1)/2]
        /* |s| is odd for symmetric s */
    Q[0] := Q[κ_j]
}
Q = Q[0]
```

The cost of calculating the additive inverse of an elliptic point is simply equal to the addition of two elements of $\mathrm{GF}(2^n)$ and it is quite small compared to an elliptic addition. As a result, the computational cost of the above algorithm is essentially $n(|s| - 1)/2$ elliptic operations (additions only). In terms of storage requirements, the above algorithm uses buffers to hold $|s|$ elliptic points. These buffers are accessed in each iteration. For high speed cryptosystems, these points can be buffered in the registers of the underlying processor. For practical applications, the number of registers needed for this purpose may be too high for many of today's processors. For example, if a Koblitz curve over $\mathrm{GF}(2^{163})$, recommended by various standardization committees, is used, then twelve 32-bit registers are needed to hold a single elliptic point (both x and y coordinates in uncompressed form). Assuming that there are only three symbols in the set s, the total number of 32-bit registers needed is thirty six.

Remark 3. If k is reduced mod $\tau^n - 1$ and represented in the signed binary τ-adic form [22], where $r = \tau$ and $s = \{-1, 0, 1\}$ and then the number of elliptic operations (i.e., additions) in Algorithm 2a. is n.

3.4 Countermeasure with Large Sized Symbol Set

With the increase of $|s|$, the cost of this algorithm increases linearly and the advantage of using the Frobenius map, rather than the *point doubling*, diminishes. What follows below is a way to reduce the cost of scalar multiplication for Koblitz curves using a few pre-computed elliptic curve points.

From (8), one can write

$$(\tau - 1)(\tau^{n-1} + \tau^{n-2} + \cdots + 1)(x, y) \equiv \mathcal{O}$$

implying that

$$(\tau^{n-1} + \tau^{n-2} + \cdots + 1)(x, y) \equiv \mathcal{O}. \tag{9}$$

Thus, in the context of the scalar multiplication one has

$$k \pmod{\tau^n - 1} \equiv \sum_{i=0}^{n-1} (\kappa_i - s_0 + 1)\tau^i$$

where s_0 is the smallest integer in s. Notice that $\kappa_i - s_0 + 1$ ensures that each symbol of the reduced k representation is a non-zero positive integer. Now, we have an efficient way to compute scalar multiplication as follows.

Algorithm 2b. Efficient SPA resistant scalar multiplication with reduced τ representation
Input: k and P
Output: $Q = kP$

```
P[0] = O
for (i = 0; i ≤ |s| − 1; i++){
    P[i + 1] := P[i] ⊎ P
}
Q := O
for (j = n − 1; j >= 0; j−−){
    Q := τQ ⊎ P[κ_j − s_0 + 1]
}
```

This algorithm requires a maximum of $n + |s|$ elliptic operations (in contrast to $n(|s| - 1)/2$ elliptic operations in Algorithm 2a.). More importantly, the number of registers needed in the loop of the above algorithm does not increase with the increase of the symbol set size. This may be advantageous for register constrained processors. The pre-computed points, namely, $P[i]$, for $0 \leq i \leq |s| - 1$, can be stored in a RAM. The latter is updated at the beginning of the algorithm and is accessed only once in each iteration.

If the activities on the address bus of the RAM can be monitored, it can be used by the attacker to reduce his efforts to recover κ_j. One way to prevent this kind of information leakage is to load the pre-computed points into random locations inside the RAM.

4 DPA Attack

SPA attacks would fail when the differences in the power signals are so small that it is infeasible to directly observe them and to apply simple power analysis. In such cases, an attacker can apply differential power analysis (DPA). The

DPA attacks are based on the same underlying principle of SPA attacks, but use statistical and digital signal processing techniques on a large number of power consumption signals to reduce noise and to strengthen the *differential* signal. The latter corresponds to the peak, if any, in the power correlation process. This signal is an indication whether or not the attacker's guess about a symbol of the n-tuple representation of the secret key is correct.

Kocher et al. first introduced the idea of DPA to attack DES [2], [4]. This DPA attack was strengthened by Messerges et al. in [3]. Coron applied the DPA attack against EC cryptosystems [12]. In this section, this attack is briefly described and possible strategies that the attacker can use to strengthen the differential signal are presented.

4.1 DPA Attack on EC Scalar Multiplication

Assume that a cryptosystem uses one of the scalar multiplication algorithms described in the previous section. Although, Algorithms 2, 2a. and 2b. are all SPA resistant, and iterations of each of these algorithms have equal amount of computational load irrespective of k_j or κ_i, the latter remains the same in all runs of the algorithm. One can take advantage of this to recover the scalar in the DPA attack as follows.

In order to apply the DPA attack, the algorithm is executed repeatedly (say, t times) with points $P_0, P_1, \cdots, P_{t-1}$ as inputs. During the execution of the algorithm, the power consumption is monitored for each iteration in which two elliptic curve points are added. In its i-th execution of the algorithm, let the power consumption signal of the j-th iteration be $S_{i,j}$, $0 \leq i \leq t-1$ and $0 \leq j \leq n-1$. Assume that the most significant $n-j'-1$ symbols, namely, $\kappa_{n-1}, \kappa_{n-2}, \cdots, \kappa_{j'+1}$ are known. In order to determine the next most significant symbol $\kappa_{j'}$, the attacker proceeds as follows.

In an attempt to analyze the power signals, a partitioning function is chosen by the attacker. This function, in its simplest form, is the same for all j and is of two value logic. The function's value depends on k, more specifically, for $j = j'$, it depends on $\kappa_{n-1}, \kappa_{n-2}, \cdots, \kappa_{j'}$. The true value, which is still unknown to the attacker, is generated within the devise which executes the scalar multiplication algorithm. Let us denote this value as $\gamma_{i,j'} \in \{0,1\}, 0 \leq i \leq t-1$. For the DPA to work for the attacker, there ought to be a difference in the power consumptions based on the two values. By guessing a value (say $\kappa'_{j'}$) for $\kappa_{j'}$, the attacker

1. comes up with his own value $\gamma'_{i,j'}, 0 \leq i \leq t-1$, for the partitioning function,
2. splits the power signals $S_{i,j'}$, for $0 \leq i \leq t-1$, into two sets: $S_0 = \{S_{i,j'} | \gamma'_{i,j'} = 0\}$ and $S_1 = \{S_{i,j'} | \gamma'_{i,j'} = 1\}$, and finally
3. computes the following differential signal

$$\delta(j') = \frac{\sum\limits_{S_{i,j'} \in S_0} S_{i,j'}}{\sum\limits_{i=0}^{t-1} \gamma'_{i,j'}} - \frac{\sum\limits_{S_{i,j'} \in S_1} S_{i,j'}}{\sum\limits_{i=0}^{t-1} (1 - \gamma'_{i,j'})}. \tag{10}$$

Notice that

$$\Pr(\gamma'_{i,j'} = \gamma_{i,j'}) \approx \begin{cases} \frac{1}{2}, \text{ if } \kappa'_{j'} \neq \kappa_{j'} \\ 1, \text{ if } \kappa'_{j'} = \kappa_{j'} \end{cases} \tag{11}$$

and for a sufficiently large value of t,

$$\lim_{t\to\infty} \delta(j') \approx \begin{cases} 0, \text{ if } \kappa'_{j'} \neq \kappa_{j'} \\ \epsilon, \text{ if } \kappa'_{j'} = \kappa_{j'} \end{cases} \tag{12}$$

where ϵ is related to the difference of the average power consumptions with $\gamma_{i,j'}$ being 0 and 1. This non-zero value of the differential signal indicates a correct guess for $\kappa_{j'}$. For the symbol set s, to obtain a non-zero differential signal the attacker needs to perform the differential power analysis $|s|/2$ times, on average.

5 Countermeasures against DPA Attacks

In this section we describe three countermeasures to prevent the DPA that an attacker can use in an effort to learn the (secret) scalar k of the Koblitz curve scalar multiplication $Q = kP$. The underlying principle is that if k is randomly changed each time it is used in the cryptosystem, the averaging out technique used in the DPA would not converge to an identifiable differential signal and the DPA attacks are expected to fail. The main challenge however is to change k to pseudo-random values with a reasonable cost and still providing the same Q.

The countermeasures presented below can be applied separately. However, when they are used together, one can expect to attain highest level of protection against power analysis attacks.

5.1 Key Masking with Localized Operations (KMLO)

For the sake of simplicity, in (4) assume that $a = 1$ (an extension using $a = 0$ is straight-forward). Then, using (7) one can write

$$2 = \tau - \tau^2 = -\tau^3 - \tau, \tag{13}$$

which shows two different representations of '2'. This in turn allows the τ-adic symbols of k to be replaced in more than one way on a window of three or more symbols. For example, using the above two representations of '2', the window of four symbols vis., $(\kappa_{i+3}, \kappa_{i+2}, \kappa_{i+1}, \kappa_i)$ can be replaced by $(\kappa_{i+3} \pm d_{i+3}, \kappa_{i+2} \pm d_{i+1}, \kappa_{i+1} \pm d_{i+1}, \kappa_i \pm d_i)$ where

$$\begin{array}{rcccc} & (d_{i+3}, & d_{i+2}, & d_{i+1}, & d_i) \\ = & (0, & \overline{1}, & 1, & \overline{2}) \\ = & (\overline{1}, & 0, & \overline{1}, & \overline{2}) \\ = & (\overline{1}, & 1, & \overline{2}, & 0) \end{array} \tag{14}$$

and $\overline{z} = -z$. If d_i's are allowed to take values outside the range [-2, 2], more combinations can be obtained. These combinations can be used to modify the

symbols of the window such that the resultant symbols belong to an expanded set s. Also, the windows can be overlapped, their starting positions can be changed and their sizes can be varied.

To implement this key masking scheme in hardware, one can use an n-stage shift register where each stage can hold one symbol of s. The key k reduced modulo $\tau^n - 1$ is initially loaded into the register. A masking unit will take a w-tuple vector ($w \geq 3$) consisting of any w adjacent symbols from the register and adds it to another vector derived from (13). (For $w = 4$, a set of possible vectors are given in (14).) The resultant vector then replaces the original w-tuple vector in the register. This process is repeated by shifting the contents of the register, possibly to mask all the symbols stored in the register. During this masking process, if a resultant symbol lies out side the set s, one can repeatedly apply (13) to restrict the symbol within s. Additionally, since $(\tau^{n-1}+\tau^{n-2}+\cdots+1)P \equiv 0P$,

$$Q = \left(\sum_{i=0}^{n-1} \kappa_i \tau^i\right) P \equiv \left(\sum_{i=0}^{n-1} \hat{\kappa}_i \tau^i\right) P \tag{15}$$

where $\hat{\kappa}_i = \kappa_i \pm c$, for $0 \leq i \leq n-1$, and c is an integer. Hence, a bias can be applied to each symbol of the key without any long addition (and hence without any carry propagation).

5.2 Random Rotation of Key (RRK)

Let

$$P' = \tau^r P, \tag{16}$$

where r is a random integer such that $0 \leq r \leq n-1$. Using (8), the elliptic curve scalar multiplication can be written as follows:

$$\begin{aligned}
Q &= kP \\
&= (\kappa_{n-1}\tau^{n-1} + \kappa_{n-2}\tau^{n-2} + \cdots + \kappa_0)P \\
&= \tau^{r-n}\left(\kappa_{r-1}\tau^{n-1} + \kappa_{r-2}\tau^{n-2} + \cdots + \kappa_r\right) P \\
&= \left(\kappa_{r-1}\tau^{n-1} + \kappa_{r-2}\tau^{n-2} + \cdots + \kappa_r\right) P' = \left(\sum_{i=0}^{n-1} \kappa_{(r-1-i) \bmod n}\ \tau^i\right) P' \\
&= \tau\,(\ \tau\,(\ \cdots \tau\,(\ \tau(\kappa_{r-1}P') \uplus \kappa_{r-2}P')\uplus \cdots) \uplus \kappa_{r+1}P') \uplus \kappa_r P'.
\end{aligned}$$

This leads to the following algorithm, where the operation "$P'[i] := s_i P'$ " can be replaced by "$P'[i] := (s_i - s_0 + 1)P'$ " if an elliptic addition of $\mathcal{O}$ has to be avoided.

Algorithm 3. Scalar Multiplication with Key Rotation
Input: k and P
Output: $Q = kP$

Pick up a random number $r \in \{0,\ 1,\ \cdots,\ n-1\}$
Compute $P' = \tau^r P$

$Q := \mathcal{O}$; $j = r$; $P'[i] := s_i P'$, $0 \le i \le |s| - 1$
for $(i = 0; i \le n - 1; i{+}{+})$ {
  $j := j - 1 \pmod n$
  $Q := \tau Q$
  $Q := Q \uplus P'[\kappa_j - s_0]$
}

Before starting the iterations, Algorithm 3 computes P'. From (16), we have

$$P' = (x^{2^r}, y^{2^r}) \tag{17}$$

where x and y are the coordinates of P. Let the *normal* basis representations of x and y be

$$x = (x_{n-1},\ x_{n-2},\ \cdots,\ x_0) \quad \text{and}$$
$$y = (y_{n-1},\ y_{n-2},\ \cdots,\ y_0),$$

respectively. Then, one can write

$$x^{2^r} = (x_{n-r-1},\ x_{n-r-2},\ \cdots,\ x_0, x_{n-1}, \cdots,\ x_{n-r}) \quad \text{and}$$
$$y^{2^r} = (y_{n-r-1},\ y_{n-r-2},\ \cdots,\ y_0, y_{n-1}, \cdots,\ y_{n-r})$$

which correspond to r-fold left cyclic shift of the representations of x and y, respectively. Thus, using a normal basis representation, one can easily compute $P' = \tau^r P$ with minimal risk of revealing the value of r against power attacks.

On the other hand, if x and y are represented with respect to a *polynomial* or other basis where the τ^r mapping is accomplished in an iterative way, measures must be taken so that the number of iterations does not reveal r.

5.3 Random Insertion of Redundant Symbols (RIRS)

The basis of this method for key randomization is that before each scalar multiplication a number of redundant symbols are inserted at random locations in the secret key sequence. In order to correctly generate the shared secret, these redundant symbols however must collectively nullify their own effects.

Before a particular scalar multiplication, assume that a total of n' redundant symbols denoted as f_i, for $0 \le i \le n' - 1$, are inserted into the original key sequence $\{\kappa_i\}$, for $0 \le i \le n - 1$. Thus the resultant sequence, denoted as $b = \{b_i\}$, for $0 \le i \le N - 1$, has $N = n + n'$ symbols. When an SPA resistant algorithm (like the ones in Section 3) is applied to do a scalar multiplication, the redundant symbols are paired, i.e., (f_0, f_1), (f_2, f_3), $\cdots$, $(f_{n'-2}, f_{n'-1})$ where n' is even. For the sake of simple implementation, redundant pairs are picked up in sequence, (i.e., first (f_0, f_1), then (f_2, f_3) and so on) and inserted at random adjacent locations starting from the most significant symbol of b. For example, with $n' \ge 4$, if the pair (f_{2l}, f_{2l+1}), for $0 \le l < n'/2 - 1$, is inserted at locations i and $i - 1$, then (f_{2l+2}, f_{2l+3}) is inserted at j and $j - 1$, for some i and j such that $N - 1 \ge i > j > 1$.

In order to implement this scheme, an N bit random number g is generated which has $\frac{n'}{2}$ non-adjacent one's. The locations of these 1's are aligned with the redundant symbols f_{2l+1}, for $0 \leq l \leq n'/2-1$, each of which corresponds to the second element of a pair of redundant symbols as shown below.

$$\begin{array}{lccccccc} \text{Location}: & N-1 & \cdots & N-r+1 & N-r & N-r-1 & N-r-2 & \cdots \\ g: & 0 & \cdots & 0 & 0 & 1 & 0 & \cdots \\ b: & \kappa_{n-1} & \cdots & \kappa_{n-r+1} & f_0 & f_1 & \kappa_{n-r} & \cdots \end{array} \tag{18}$$

In (18), the first pair (f_0, f_1) has been inserted at random location $N-r$. Assume that all b_i's belong an *expanded* symbol set $u = \{u_0, u_1, \cdots, u_{|u|-1}\}$. To perform scalar multiplication on the Koblitz curve, we can then state the following algorithm, where b'_i is an integer in the range [0,|u|-1] and is obtained as $b'_i = \imath$, for $0 \leq \imath \leq |u|-1$, given that $b_i = u_\imath$.

Algorithm 4. Scalar Multiplication with Random Insertions
Input: k and P
Output: $Q = kP$

Compute $P[i] = u_i P, \quad 0 \leq i \leq |u|-1$
Generate g and form b
$Q := \mathcal{O}$; $R[0] := \mathcal{O}$; $R[1] := \mathcal{O}$
for $(i = N-1; i \geq 0; i--)$ {
  $R[0] := \tau R, \quad R[1] := \tau^{-1} R$
  $Q := R[g_i] \uplus P[b'_i]$
}

In the above algorithm, let $Q^{(j)}$ denote the point Q at iteration $i = j$. In order to determine the value of the redundant symbols, without loss of generality we refer to (18). To obtain the correct Q at the end of iteration $i = N-r+2$, the algorithm should yield

$$Q^{(N-r-2)} = \kappa_{n-1}\tau^{r-1} + \kappa_{n-2}\tau^{r-2} + \cdots + \kappa_{n-r+1}\tau + \kappa_{n-r}. \tag{19}$$

In this regard, note that

$$Q^{(N-r)} = \kappa_{n-1}\tau^{r-1} + \kappa_{n-2}\tau^{r-2} + \cdots + \kappa_{n-r+1}\tau + f_0.$$

With $i = N-r-1$ and $i = N-r-2$ we have $g_{N-r-1} = 1$ and $g_{N-r-2} = 0$, respectively. In Algorithm 4, these two values of g_i's correspond to τ^{-1} and τ mappings, respectively, and yield the following:

$$\begin{aligned} Q^{(N-r-1)} &= \tau^{-1}(\kappa_{n-1}\tau^{r-1} + \kappa_{n-2}\tau^{r-2} + \cdots + \kappa_{n-r+1}\tau + f_0) + f_1, \\ Q^{(N-r-2)} &= \kappa_{n-1}\tau^{r-1} + \kappa_{n-2}\tau^{r-2} + \cdots + \kappa_{n-r+1}\tau + f_0 + f_1\tau + \kappa_{n-r}. \end{aligned} \tag{20}$$

Comparing (19) and (20), one can see that if f_1 is chosen to nullify the effect of f_0, then these two redundant symbols should have the following relationship

$$f_0 + \tau f_1 = 0. \tag{21}$$

On the other hand, if the pair (f_2, f_3) is to cancel (f_0, f_1), then the following should satisfy

$$f_2 + \tau f_3 + \tau^{m-1}(f_0 + \tau f_1) = 0 \tag{22}$$

assuming that f_2 is m positions away from f_1. The value of m can be varied. However for the sake of simpler implementation it can be fixed to a value which could be as small as unity.

5.4 Comments

- The three methods presented above use τ-adic representation of the key k. The first and the third methods can be extended to other bases of presentation of k.
- Each of the three proposed methods uses a look-up table which hold the points $u_i P$, for $0 \leq i \leq |u| - 1$, corresponding to the expanded symbol set u which contains all the symbols of the randomized key. The table contents are to be updated each time a new P (or P' in the random rotation of the key) is used. If the symbol set is symmetric then the look-up table may contain only $\{u_i P : u_i \geq 0, 0 \leq i \leq |u| - 1\}$. This is possible because the negative multiples of P can be obtained from the positive multiples using $-(x, y) = (x, x + y)$. Although, this technique reduces the table size by half, it introduces an extra step for the negative digits, which may reveal the signs of the digits unless proper measures are taken to protect them.

6 Comparison and Concluding Remarks

In the randomization technique of [12], a multiple of the total number of curve points $\mathcal{E}$ is added to k. Since $\mathcal{E}P = \mathcal{O}$,

$$Q = kP = (k + e\mathcal{E})P \tag{23}$$

where e is an integer. The realization of this key randomization scheme requires a large integer multiplier ($\mathcal{E}$ is about n bits long). If this multiplier is not already part of the system into which the power analysis resistant elliptic curve scalar multiplication is to be embedded, it will result in a considerable increase in the silicon area. On the other hand, the key masking/randomization scheme presented in Section 5 does not need such a multiplier.

Recently, Chari et al. have proposed generic methods [13] to countermeasure differential power analysis attacks. For instance, one can randomly pick up a pair of numbers k' and k'' such that $k = k' + k''$, and computes $k'P \uplus k''P$. A straightforward implementation of this scheme would require longer computation time since two scalar multiplications are involved. In order to reduce the computation time, if certain speed-up techniques (e.g., Shamir's trick) are used, then one would require relatively more complex implementation.

Although, the method proposed in [11] focuses on key randomization in modular exponentiation, one can attempt to extend it to elliptic curve scalar multiplication. It starts the computation of Q at a random symbol of k, but always

terminates the computation at the most significant symbol giving the adversary an opportunity to work backwards. For such an attack on Koblitz curve based cryptosystems, the adversary needs to compute the inverse of the τ mapping, which unlike the square root operation in modular exponentiation, can be quite simple. On the other hand, Algorithm 3 of this article also starts at random position (i.e., symbol), but unlike [11] it terminates adjacent to the random starting position. As a result, it is less vulnerable to the backward power analysis attack.

When applied to Koblitz curve based cryptosystems, the proposed countermeasures are expected to be less complex than the similar ones already exist. Nevertheless, their overall impacts on the cryptosystems need to be carefully investigated and possible trade-offs are to be identified for implementation in real systems. More importantly, these countermeasures are to be investigated against more advanced attacks.

Acknowledgment

This work was mostly done during the author's sabbatical leave with the Motorola Laboratories, Schaumburg, IL. The author thanks Larry Puhl for his encouragement to pursue this work, and acknowledges the stimulating discussions he had on this topic with Ezzy Dabbish, Tom Messerges and Brian King.

References

1. P. Kocher, "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Systems," in *Advances in Cryptology- CRYPTO '96, Lecture Notes in Computer Science*, pp. 104–113, Springer-Verlag, 1996.
2. P. Kocher, J. Jaffe, and B. Jun, "Introduction to Differential Power Analysis and Related Attacks." `http://www.cryptography.com/dpa/technical`, 1998.
3. T. Messerges, E. A. Dabbish, and R. H. Sloan, "Investigation of Power Analysis Attacks on Smartcards," in *Proceedings of USENIX Workshop on Electronic Commerce*, pp. 151–161, 1999.
4. P. Kocher, J. Jaffe, and B. Jun, "Differential Power Analysis," in *Advances in Cryptology- CRYPTO '99, Lecture Notes in Computer Science*, pp. 388–397, Springer-Verlag, 1999.
5. D. Boneh, R. A. Demillo, and R. J. Lipton, "On the Importance of Checking Cryptographic Protocols for Faults," in *Advances in Cryptology- CRYPTO '97, Lecture Notes in Computer Science*, pp. 37–51, Springer-Verlag, 1997.
6. E. Biham and A. Shamir, "Differential Fault Analysis of Secret Key Cryptosystems," in *Advances in Cryptology- CRYPTO '97, Lecture Notes in Computer Science*, pp. 513–525, Springer-Verlag, 1997.
7. E. Biham and A. Shamir, "Differential Cryptanalysis of DES-like Cryptosystems," *Journal of Cryptology*, vol. 4, pp. 3–72, 1991.
8. M. Matsui, "Linear Cryptanalysis Method for DES Cipher," in *Advances in Cryptology- EUROCRYPT '93, Lecture Notes in Computer Science*, pp. 386–397, Springer-Verlag, 1994.

9. E. Biham and A. Shamir, "Differential Cryptanalysis of the Full 16-round DES," in *Advances in Cryptology- CRYPTO '92, Lecture Notes in Computer Science*, pp. 487–496, Springer-Verlag, 1993.
10. J. Kelsey, "Side Channel Cryptanalysis of Product Ciphers," in *ESORICS, Lecture Notes in Computer Science*, pp. 487–496, Springer-Verlag, 1998.
11. T. Messerges, E. A. Dabbish, and R. H. Sloan, "Power Analysis Attacks on Modular Exponentiation in Smartcards," in *Proceedings of Workshop on Cryptographic Hardware and Embedded Systems*, pp. 144–157, LNCS, Springer-Verlag, 1999.
12. J.-S. Coron, "Resistance Against Differential Power Analysis for Elliptic Curve Cryptosystems ," in *Workshop on Cryptographic Hardware and Embedded Systems*, pp. 292–302, LNCS, Springer-Verlag, 1999.
13. S. Chari, J. R. R. C. S. Jutla, and P. Rohatgi, "Towards Sound Approaches to Counteract Power-Analysis Attacks," in *Advances in Cryptology- CRYPTO '99*, pp. 398–412, 1999.
14. N. Koblitz, "CM-Curves with Good Cryptographic Properties," in *Advances in Cryptology- CRYPTO '91, Lecture Notes in Computer Science*, pp. 279–287, Springer-Verlag, 1992.
15. V. S. Miller, "Use of Elliptic Curves in Cryptography," in *Advances in Cryptology-CRYPTO '85*, pp. 417–426, Springer, 1986.
16. N. Koblitz, "Elliptic Curve Cryptosystems," *Math. Comp.*, vol. 48, pp. 203–209, 1993.
17. A. J. Menezes, *Elliptic Curve Public Key Cryptosystems*. Kluwer Academic Publishers, 1993.
18. J. H. Silverman, *The Arithmetic of Elliptic Curves*, vol. 106. New York: Springer-Verlag, 1986.
19. I. F. Blake, G. Seroussi, and N. P. Smart, *Elliptic Curves in Cryptography*. Cambridge Univ Press, 1999.
20. R. Schroeppel, S. O'Malley, H. Orman, and O. Spatscheck, "A Fast Software Implementation for Arithmetic Operations in GF(2^n)," in *Advances in Cryptology-CRYPTO '95, Lecture Notes in Computer Science*, pp. 43–56, Springer, 1995.
21. H. Wu, M. A. Hasan, and I. F. Blake, "Highly Regular Architectures for Finite Field Computation Using Redundant Basis," in *Proceedings of Workshop on Cryptographic Hardware and Embedded Systems*, pp. 269–279, LNCS, Springer-Verlag, 1999.
22. D. Gordon, "A Survey of Fast Exponentiation Methods," *Journal of Algorithms*, vol. 27, pp. 129–146, 1998.
23. H. Wu, "Low Complexity Bit-Parallel Finite Field Arithmetic Using Polynomial Basis," in *Proceedings of Workshop on Cryptographic Hardware and Embedded Systems*, pp. 280–291, LNCS, Springer-Verlag, 1999.
24. J. Solinas, "An Improved Algorithm for Arithmetic on a Family of Elliptic Curves," in *Advances in Cryptology- CRYPTO '97, Lecture Notes in Computer Science*, pp. 357–371, Springer-Verlag, 1997.
25. T. Kobayashi, H. Morita, K. Kobayashi, and F. Hoshino, "Fast Elliptic Curve Algorithm Combining Frobenius Map and Table Reference to Adapt to Higher Characteristic," in *Advances in Cryptology- EUROCRYPT '99, Lecture Notes in Computer Science*, pp. 176–189, Springer-Verlag, 1999.

A Timing Attack against RSA with the Chinese Remainder Theorem

Werner Schindler

Bundesamt für Sicherheit in der Informationstechnik (BSI)
Godesberger Allee 183, 53175 Bonn, Germany
Werner.Schindler@bsi.bund.de

Abstract. We introduce a new type of timing attack which enables the factorization of an RSA-modulus if the exponentiation with the secret exponent uses the Chinese Remainder Theorem and Montgomery's algorithm. Its standard variant assumes that both exponentiations are carried out with a simple square and multiply algorithm. However, although its efficiency decreases, our attack can also be adapted to more advanced exponentiation algorithms. The previously known timing attacks do not work if the Chinese Remainder Theorem is used.
Keywords: Timing attack, RSA, Chinese Remainder Theorem, Montgomery multiplication.

1 Introduction

The central idea of any timing attack is to determine a secret parameter from differences between running times needed for various input values. At Crypto 96 Kocher introduced a timing attack on modular exponentiations ([6]). In [3] a successful attack against an early version of a Cascade smart card is described. In [9] these attacks are optimized and, moreover, the assumptions on the attacker's abilities are weakened considerably.

The attacks mentioned above recover an unknown exponent (e.g. a secret RSA key) bit by bit. They yet do not work if the Chinese Remainder Theorem (CRT) is used as it is essential to know the exact input values of the respective arithmetical operations at any instant. (Concerning the CRT there is no more than a rough idea sketched in [6] (Sect. 7) how to exploit time differences caused by the initial reduction $y \mapsto y (\mathrm{mod}\, p_j)$. However, this should be of little practical significance since the variance of the remaining hundreds of arithmetic operations usually is gigantic compared with this effect.)

In this paper we introduce and investigate a completely new type of timing attack. It enables the factorization of an RSA modulus n if the exponentiation with the secret exponent uses the CRT while the multiplications and squarings modulo the prime factors p_1 and p_2 are carried out with Montgomery's algorithm ([8]). The standard variant of our attack assumes that both exponentiations use a simple square and multiply algorithm. Although its efficiency decreases, our attack can also be adapted to more advanced exponentiation algorithms. Our

Ç.K. Koç and C. Paar (Eds.): CHES 2000, LNCS 1965, pp. 109–124, 2000.

attack is very robust and, for comparison, needs notably fewer time measurements than the attacks introduced in [6] and [3]. If exponentiation uses square and multiply under optimal circumstances the standard variant of our attack requires fewer than 600 time measurements to factor a 1024 bit modulus n. Making use of a lattice-based algorithm introduced by Coppersmith in [2] again nearly halves this number of time measurements. Further, one can verify during the attack with high probability whether the decisions have been correct so far. The only weakness of our attack is that it is a chosen-input attack.

First, we briefly describe and investigate Montgomery's algorithm and exponentiation with CRT. In Sect. 3 general assumptions are formulated and discussed and the central idea of our attack is illustrated. In Sect. 4 and 5 the standard variant of our attack is worked out, error probabilities are computed, and mechanisms for error detection and error correction are discussed. Sect. 6 presents results of practical experiments and in Sect. 7 we extend our attack to implementations which use more advanced exponentiation schemes than square and multiply. The paper ends with remarks on fields of application, possible countermeasures and concluding remarks.

2 Montgomery's Algorithm and the CRT

Let $(d_w d_{w-1} \dots d_0)_2$ denote the binary representation of $d \in \mathbb{N}$ where $d_w = 1$ denotes the most significant bit. If d denotes a secret key the computation of $y^d (\bmod m)$ usually requires hundreds of modular squarings and multiplications. Hence many implementations use Montgomery's algorithm ([8]) which transfers time-consuming modular multiplications modulo m to a modulus $R > m$ with $gcd(R, m) = 1$ which fits to the device's hardware architecture. (Usually $R = 2^\omega$ where ω is a multiple of 32 or 64.)

Let's have a closer look at Montgomery's algorithm. As usually, for $a \in Z$ the term $a (\bmod m)$ denotes the smallest nonnegative integer which is congruent to a modulo n while $R^{-1} \in Z_m := \{0, 1, \dots, m-1\}$ denotes the multiplicative inverse of R in Z_m. The integer $m^* \in Z_R$ satisfies the equation $RR^{-1} - mm^* = 1$ in Z. To simplify notation we introduce the mappings $\Psi, \Psi_* \colon Z \to Z_m$ defined by $\Psi(x) := (xR) (\bmod m)$ and $\Psi_*(x) := (xR^{-1}) (\bmod m)$. As easily can be checked Ψ and Ψ_* are bijective on Z_m and, moreover, inverse mappings; i.e. $\Psi_*(\Psi(x)) = x$ for all $x \in Z_m$. For $a' := \Psi(a)$ and $b' := \Psi(b)$ Montgomery's algorithm returns $s := \Psi_*(\Psi(a)\Psi(b)) = \Psi(ab)$. The subtraction $s - m$ in line 4 is called *extra reduction*.

Montgomery's algorithm

```
z:=a'b'
r:=(z(mod R)m*) (mod R)
s:=(z+rm)/R
if s≥m then s:=s-m
return s        (= Ψ*(a'b') = a'b'R^{-1} (mod m))
```

Remark 1. Many implementations use a more efficient multiprecision variant of Montgomery's algorithm than listed above (see e.g. [7], Algorithm 14.36, or

[10]). The factors a' and b' then are internally represented with respect to a basis h which fits perfectly to the hardware multipliers (typically, $h = 2^{32}$), that is, $a' := \sum_{j=0}^{t-1} a'_j h^j$ and $b' := \sum_{j=0}^{t-1} b'_j h^j$. Instead of the product $a'b'$ and the reduction modulo $R := h^t$ it suffices to calculate $a'_0 b', a'_1 b', \ldots, a'_{t-1} b'$ and to perform t reductions modulo h. However, whether an extra reduction is necessary does not depend on the chosen basis h but on $R := h^t$. (This can be verified with a simple induction proof.) Thus, as will become clear later, the concrete realization of Montgomery's algorithm is indeed of no significance for our attack.

Combined with Montgomery's algorithm the square and multiply exponentiation algorithm reads as follows:

Exponentiation algorithm 1
(Square and multiply using Montgomery's algorithm)

```
temp := Ψ(y)
for i=w-1 down to 0 do {
  temp := Ψ*(temp²)
  if (dᵢ=1) then temp := Ψ*(temp*Ψ(y))
  }
return Ψ*(temp)
```

Suggestively, we call operations $\Psi_*(\text{temp}^2)$ and $\Psi_*(\text{temp} * \Psi(y))$ *Montgomery multiplications* in the following. The number of extra reductions in Exponentiation algorithm 1 depends on the particular base y, or more directly, on $\Psi(y)$.

Lemma 1. *(i) Montgomery's algorithm requires an extra reduction step iff*

$$\frac{a'b'}{Rm} + \frac{a'b'm^* (\bmod R)}{R} \geq 1. \tag{1}$$

(ii) Let the random variable B be equidistributed on Z_m. Unless the ratio $R/gcd(R, \Psi(a))$ is extremely small

$$\text{Prob}\,(\textit{extra reduction in } \Psi_*(\Psi(a)B)) = \frac{\Psi(a)}{2R} \qquad \textit{for } a \in Z_m \tag{2}$$

holds and similarly

$$\text{Prob}\left(\textit{extra reduction in } \Psi_*(B^2)\right) = \frac{m}{3R}\ . \tag{3}$$

Proof. For the proof of (1) to (3) we refer the interested reader to [9]. We merely sketch the central idea to verify (2). Its left-hand side is equivalent to $\text{Prob}(Ac + (Acmm^* (\bmod 1)) \geq 1)$ where we temporarily use the abbreviations $A := B/m$ and $c := \Psi(a)/R$. Further, for fixed $v \in \mathbb{N}$ (e.g., $v = 32$) define the intervals $I_j := [j2^{-v}, (j+1)2^{-v})$ for $j < 2^v$. For realistic modulus size m under mild conditions $\text{Prob}(Acmm^* (\bmod 1) \mid A \in I_i) \approx 2^{-v}$ should be an excellent approximation for $i, j < 2^v$. Then the left-hand side of (2) approximately equals $\text{Prob}(Uc + V \geq 1)$ where U and V denote independent random variables being equidistributed on $[0, 1)$. The latter probability equals the right-hand side of (2).

Remark 2. We point out that the proof of (2) and (3) is not exact in a mathematical sense as it uses (though plausible!) heuristic arguments at two steps. However, results from practical experiments (thousands of pseudorandom factors and various moduli) match perfectly with (2) and (3). For none of these factors and moduli neither (2) nor (3) turned out to be wrong. Hence it seems to be resonable to use "=" instead of "≈".

Let $n = p_1 p_2$ denote an RSA-modulus with primes p_1 and p_2 while d denotes the secret exponent. Steps 1 to 3 below describe how to compute $y^d (\mathrm{mod}\, n)$ using the CRT and Montgomery's algorithm. The numbers d', d'', b_1 and b_2 and the parameters for the Montgomery multiplications $(\mathrm{mod}\, p_1)$ and $(\mathrm{mod}\, p_2)$ are precomputed in a setup Step carried out only once after loading (or generating within the device, resp.) p_1, p_2 and d. In particular, $d' := d(\mathrm{mod}\,(p_1 - 1))$ and $d'' := d(\mathrm{mod}\,(p_2 - 1))$ while $b_1 \equiv 1 \pmod{p_1}$ and $b_1 \equiv 0 \pmod{p_2}$, and similarly, $b_2 \equiv 0 \pmod{p_1}$ and $b_2 \equiv 1 \pmod{p_2}$.

CRT using Exponentiation algorithm 1

Step 1: a) $y_1 := y(\mathrm{mod}\, p_1)$
b) Compute $x_1 := (y_1)^{d'} (\mathrm{mod}\, p_1)$ with Exponentiation algorithm 1
Step 2: a) $y_2 := y(\mathrm{mod}\, p_2)$
b) Compute $x_2 := (y_2)^{d''} (\mathrm{mod}\, p_2)$ with Exponentiation algorithm 1
Step 3: Return $(b_1 x_1 + b_2 x_2)(\mathrm{mod}\, n)$

3 General Assumptions and the Central Idea

We assume that the attacker has access to a hardware device (smart card, PC etc.) which calculates the modular exponentiation with a secret RSA exponent using CRT with Montgomery multiplication. Below, we will formulate and discuss assumptions concerning the implementation. Then we derive the main theorem and explain the central idea of our attack.

Definition 1. *In analogy to Sect. 2 for $i = 1, 2$ we define the mappings $\Psi_i, \Psi_{i*}: Z \to Z_{p_i}$ by $\Psi_i(a) := (aR)(\mathrm{mod}\, p_i)$ and $\Psi_{i*}(a) := (aR^{-1})(\mathrm{mod}\, p_i)$. As usually, the greatest common divisor of integers a and b is denoted with $\gcd(a, b)$. The term $N(\mu, \sigma^2)$ denotes a normal distribution with mean (=expected value) μ and variance σ^2. A value taken on by a random variable X is called a* realization *of X.*

General Assumptions. a) The attacker is able to use the hardware device for chosen inputs and to measure the total time needed for an exponentiation.
b) A modular exponentiation $y^d (\mathrm{mod}\, n)$ is computed with the CRT using Exponentiation algorithm 1 stated at the end of Sect. 2.
c) The attacker knows the modulus n.
d) Both, Montgomery multiplications $(\mathrm{mod}\, p_1)$ and $(\mathrm{mod}\, p_2)$ use the same parameter value R.
e) Montgomery multiplications $(\mathrm{mod}\, p_1)$ and $(\mathrm{mod}\, p_2)$ require time c if no extra reduction is needed and $c + c_{\mathrm{ER}}$ otherwise.

f) Running times are reproducible, i.e. for fixed d and n the running time Time($y^d \pmod n$) does only depend on the base y but not on other (e.g. external) influences.
g) For a randomly chosen base y the cumulative time needed for all operations besides the Montgomery multiplications inside the loop of Exponentiation Algorithm 1 in Steps 1 and 2 of the CRT algorithm (in particular, the time needed for input and output operations, for the calculation of $y_i, \Psi_i(y_i), \Psi_{i*}(\text{temp})$ $(i = 1, 2)$ and for $b_1x_1 + b_2x_2 (\bmod n)$) may be viewed as realization of a $N(\mu_{\text{CRT}}, {\sigma_{\text{CRT}}}^2)$-distributed random variable.

Our attack is a chosen input attack which enables the factorization of the modulus n. It will turn out that it tolerates measurement errors and external influences and also works under less restrictive assumptions. In Sect. 7 it will be extended to CRT combined with more advanced exponentiation algorithms than Exponentiation algorithm 1. We will reference to the general assumptions using the abbreviation GA.

Remark 3. (i) Re GA d): The assumption that both, exponentiation $(\bmod p_1)$ and $(\bmod p_2)$ use the same parameter value R is usually fulfilled as both prime factors have the same number of bits.
(ii)Re GA e): Reductions modulo a power of 2 and divisions by a power of 2 can simply be realized by neglecting the high-value bits or as a shift operation, resp. Due to GA d) assumption GA e) usually should be fulfilled (see also [3]). However, slight deviations from constant running times could be interpreted as a part of the measurement error (see Remark 7).
(iii) Re GA f): Concerning external influences assumption GA f) should be fulfilled for smart cards (but not for multi-user-systems). Instead, there might be randomly chosen dummy operations masking the "true" running time (see Remark 7).

To simplify further notation we will use the following abbreviations and definitions

$$R^{-1} := R^{-1} (\bmod n), \qquad \beta := \sqrt{n/R^2} \tag{4}$$

$$T(u) := \text{Time}((uR^{-1} (\bmod n))^d (\bmod n)) \ . \tag{5}$$

The term $T(u)$ covers all the time needed to compute $(uR^{-1}(\bmod n))^d (\bmod n)$. The CRT delivers $(uR^{-1} (\bmod n))R \equiv u \pmod{p_i}$ so that Theorem 1 is an immediate corollary of Lemma 1. We recall that the right-hand sides of (6) and (7) are at least excellent approximations. Theorem 1 is crucial for our attack.

Theorem 1. *Let B_i denote a random variable being uniformly distributed on Z_{p_i}. Then for $i = 1, 2$*

$$\text{pr}_{i*} := \mathit{Prob}\left(\mathit{extra\ reduction\ in}\ \Psi_{i*}(B_i^2)\right) = \frac{p_i}{3R} \tag{6}$$

and, unless the ratio $R/gcd(R, u (\bmod p_i))$ is extremely small, also

$$\begin{aligned} \text{pr}_i(u) :=& \text{Prob}\left(\mathit{extra\ reduction\ in}\ \Psi_{i*}(\Psi_i(uR^{-1} (\bmod n))B_i) = \Psi_{i*}(uB_i)\right) \\ &= \frac{u (\bmod p_i)}{2R} \qquad \mathit{for}\ u \in Z_n \ . \end{aligned} \tag{7}$$

Figures 1 and 2 below illustrate the central idea of our attack. For base $uR^{-1} (\mathrm{mod}\, n)$ hundreds of Montgomery multiplications have to carried out with factors $u (\mathrm{mod}\, p_1)$ and $u (\mathrm{mod}\, p_2)$, respectively. ¿From (7) we conclude that the probability for an extra reduction within any of these multiplication is linear in the respective factor. Differences between running times required for modular exponentiations result from different numbers of extra reductions within the respective modular multiplications and squarings. Figure 2 plots the expected number of extra reductions (=E(#er)) as a function of u in a neighborhood of kp_i where k is an integer and $i \in \{1, 2\}$. Hence for $u_1 < u_2$ with $u_2 - u_1 \ll p_i$ the time difference $T(u_2) - T(u_1)$ should reveal whether the interval $\{u_1 + 1, \ldots, u_2\}$ contains an integer multiple of at least one prime factor p_i or not. In the first case $T(u_1)$ should be significantly larger than $T(u_2)$ while in the second case both running times should approximately be equal. In the following sections we will make this intuitive idea precise.

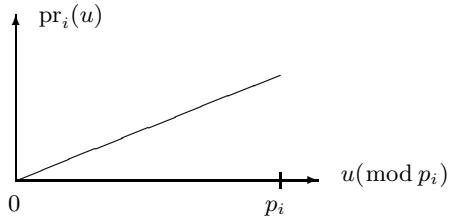


Fig. 1. Probability for an extra reduction with a random cofactor

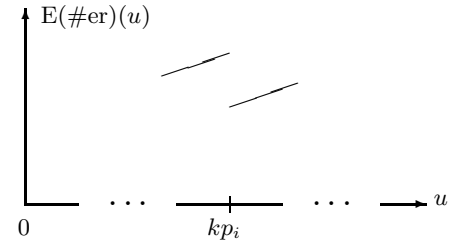


Fig. 2. The expected number of extra reductions is discontinuous at each integer multiple of p_i.

Our attacks falls in three phases: In Phase 1 an "interval set" $\{u_1 + 1, \ldots, u_2\}$ has to found which contains an integer multiple of p_1 or p_2. Starting from this set in Phase 2 a sequence of decreasing interval subsets has to be determined, each of which containing an integer multiple of p_1 or p_2. The decisions in Phase 1 and 2 are based on the respective time differences $T(u_2) - T(u_1)$. As soon as the actual subset is small enough Phase 3 begins where $\gcd(u, n)$ is calculated for all u contained in this subset. If all decisions within Phase 1 and 2 were correct then the final subset indeed contains a multiple of p_1 or p_2 so that Phase 3 delivers the factorization of n.

4 Basic Scheme

Let the exponent d' be a $(w_1 + 1)$ bit number with $(g_1 + 1)$ ones in its binary representation and similarly, the exponent d'' be a $(w_2 + 1)$ bit number with $(g_2 + 1)$ ones in its binary representation. Then within the for-loops of Exponentiation

algorithm 1 $w_1+w_2+g_1+g_2$ Montgomery multiplications are carried out, in particular, w_1 of type $\Psi_{1*}(\text{temp}^2)$ and g_1 of type $\Psi_{1*}(\text{temp} * \Psi_1(y_1))$ and, similarly, w_2 of type $\Psi_{2*}(\text{temp}^2)$ and g_2 of type $\Psi_{2*}(\text{temp} * \Psi_2(y_2))$. For $y = uR^{-1} (\bmod n)$ we will view the cumulative time needed for these Montgomery multiplications as a realization of a $N(\mu_{\text{MM}}(u), \sigma_{\text{MM}}(u)^2)$-distributed random variable. The temp-values are the intermediate results $\Psi_i(y_i), \Psi_i(y_i^2), \ldots, \Psi_i(y_i^d)$ which behave as realizations of random variables equidistributed on Z_{p_i}. Then (9) is an immediate consequence of Theorem 1. We point out that extra reductions in subsequent Montgomery multiplications are negative correlated and that under mild assumptions a pendant of the well-known central limit theorem also holds for dependent random variables. (The first assertion follows from the fact that after an extra reduction $\text{temp} < (p_i/R)p_i$ or $\text{temp} < (u(\bmod p_i)/R)p_i$, resp.) Consequently, due to GA g) the running time $T(u)$ then may be viewed as a realization of a $N(\mu(u), \sigma(u)^2)$-distributed random variable X_u with

$$\mu(u) = \mu_{\text{CRT}} + \mu_{\text{MM}}(u), \qquad \sigma(u)^2 = {\sigma_{\text{CRT}}}^2 + \sigma_{\text{MM}}(u)^2 \tag{8}$$

$$\mu_{\text{MM}}(u) \approx c \sum_{i=1}^{2} (w_i + g_i) + \text{c}_{\text{ER}} \sum_{i=1}^{2} (w_i \text{pr}_{i*} + g_i \text{pr}_i(u)) \tag{9}$$

Theorem 2.

$$\sigma_{\text{MM}}(u)^2 \approx {\text{c}_{\text{ER}}}^2 \sum_{i=1}^{2} [w_i \text{pr}_{i*}(1 - \text{pr}_{i*}) + g_i \text{pr}_i(u)(1 - \text{pr}_i(u)) \tag{10}$$

$$+2(g_i - 1)\text{cov}_{i;\text{MQ}}(u) + 2g_i \text{cov}_{i;\text{QM}}(u) + 2(w_i - g_i)\text{cov}_{i;\text{QQ}}] \quad \textit{with}$$

$$\text{cov}_{i;\text{MQ}}(u) = 2\text{pr}_i(u)^3 \text{pr}_{i*} - \text{pr}_i(u)\text{pr}_*, \quad \text{cov}_{i;\text{QM}}(u) = \frac{9}{5}\text{pr}_i(u)\text{pr}_{i*}^2 - \text{pr}_i(u)\text{pr}_*,$$

$$\text{cov}_{i;\text{QQ}} = \frac{27}{7}\text{pr}_{i*}^4 - \text{pr}_{i*}^2 \ .$$

Proof. Using the same notation as in the proof of Lemma 1 the left-hand side of (2) equals $\text{Prob}(Ac + Acmm^* (\bmod 1) \geq 1)$. As pointed out there slight deviations in the first summand should cause "vast" variations in the second, i.e. with respect to this particular probability both summands should behave as if they were independent and if the second was equidistributed on $[0, 1)$. Using (1) a similar assertion can be derived for (3). Under these assumptions we may view the temp values in Exponentiation algorithm 1 in Step $i \in \{1, 2\}$ as realizations of the random variables $S_0, S_1, \ldots$ which are recursively defined by $S_0 := \Psi_i(y_i)$ and $S_k := (S_{k-1}\Psi_i(y_i)/R + V_k) \,(\bmod 1)$ or $S_k := (S_{k-1}^2 p_i/R + V_k) \,(\bmod 1)$, resp., if the k^{th} Montgomery multiplication is a multiplication with $\Psi_i(y_i)$ or a squaring, resp. Here $V_1, V_2, \ldots$ denote independent random variables being equidistributed on $[0, 1)$. Further, define the $\{0, 1\}$-valued random variables $W_1, W_2, \ldots$ by $W_k := 1_{\{S_k < S_{k-1}\Psi_i(y_i)/R\}}$ or $W_k := 1_{\{S_k < S_{k-1}^2 p_i/R\}}$, resp. Then $W_k = 1$ iff an extra reduction is necessary in Montgomery multiplication k. As $V_1, V_2, \ldots$ are independent and equidistributed on $[0, 1)$ the same is true for $S_1, S_2 \ldots$. If ν

denotes the distribution of S_{g-1} (Dirac measure or equidistribution, resp.) then for $1 \le g < h$ we obtain the covariance $\mathrm{Cov}_{\mathrm{MQ}}(W_g W_h) =$

$$\int_{[0,1)^{h-g+2}} 1_{\{S_g < S_{g-1}\Psi_i(y_i)/R\}}(s_g) \cdot 1_{\{S_h < S_{h-1}^2 p_i/R\}}(s_h)\, ds_h \cdots ds_g \nu(ds_{g-1})$$
$$- \left(\int_{[0,1)^2} 1_{\{S_g < S_{g-1}\Psi_i(y_i)/R\}}(s_g)\, ds_g \nu(ds_{g-1}) \right) \cdot \left(\int_{[0,1)^2} 1_{\{S_h < S_{h-1}^2 p_i/R\}}(s_h)\, ds_h\, ds_{h-1} \right).$$

Here subscripts MQ means that index g belongs to a multiplication with $\Psi_i(y_i)$ and h to a squaring. The covariance $\mathrm{Cov}_{\mathrm{MQ}}(W_g W_h) = 0$ if $h > g+1$ and equals $\mathrm{cov}_{i;\mathrm{MQ}}(u)$ if $h = g+1 > 2$. Equivalent assertions hold for $\mathrm{Cov}_{\mathrm{QM}}(W_g W_h)$, $\mathrm{Cov}_{\mathrm{MM}}(W_g W_h)$ and $\mathrm{Cov}_{\mathrm{QQ}}(W_g W_h)$. Approximating the covariance of W_1 and W_2 by $\mathrm{cov}_{i;\mathrm{QM}}(u)$ or $\mathrm{cov}_{i;\mathrm{QQ}}$, resp., finishes the proof of Theorem 2 as the least-value bits of d' and d'' are 1.

Remark 4. (i) The proof of Theorem 2 is not exact in a mathematical sense as it uses the same heuristic arguments as Lemma 1. However, (10) matches with practical experiments. We point out that the variance $\sigma_{\mathrm{MM}}(u)^2$ does not affect the (single) decision rule defined below but its knowledge enables the choice of appropriate parameters s and N (see Sect. 5 and 7).

Now let $0 < u_1 < u_2 < n$ with $u_2 - u_1 < p_1, p_2$. Three cases are possible:

Case A: $\{u_1+1, \ldots, u_2\}$ does not contain a multiple of p_1 or p_2.
Case B: $\{u_1+1, \ldots, u_2\}$ contains a multiple of one of p_1 or p_2 but not of both.
Case C: $\{u_1+1, \ldots, u_2\}$ contains a multiple of both p_1 or p_2.

Clearly, the expected value of the time difference $T(u_2) - T(u_1)$ equals $E(X_{u_2} - X_{u_1}) = \mu(u_2) - \mu(u_1)$. In Case B p_j denotes the prime factor of which $\{u_1 + 1, \ldots, u_2\}$ contains a multiple. From (9) and Theorem 1 we obtain

$$E(X_{u_2} - X_{u_1}) = \begin{cases} \frac{c_{\mathrm{ER}}}{2R}\left(g_1(u_2-u_1) + g_2(u_2-u_1)\right) & \text{in Case A} \\ \frac{c_{\mathrm{ER}}}{2R}\left(g_j(u_2-u_1-p_j) + g_{3-j}(u_2-u_1)\right) & \text{in Case B} \\ \frac{c_{\mathrm{ER}}}{2R}\left(g_1(u_2-u_1-p_1) + g_2(u_2-u_1-p_2)\right) & \text{in Case C} \end{cases} \quad (11)$$

unless the ratios $R/\gcd(R, u_1(\bmod p_i))$ and $R/\gcd(R, u_2(\bmod p_i))$ are extremely small. (For randomly chosen u_1, u_2 this should not never occur in practice.) If $u_2 - u_1 \ll p_1, p_2$ the expected values differ considerably:

$$E(X_{u_2} - X_{u_1}) \approx \begin{cases} 0 & \text{in Case A} \\ -\frac{c_{\mathrm{ER}}}{2R}\left(g_j p_j\right) & \text{in Case B} \\ -\frac{c_{\mathrm{ER}}}{2R}\left(g_1 p_1 + g_2 p_2\right) & \text{in Case C.} \end{cases} \quad (12)$$

Note that secret RSA exponents are chosen randomly. (In many applications the public exponent is a fixed small value, e.g. 3 or $2^{16}+1$. However, d may be interpreted as a function of the randomly chosen prime factors p_1 and p_2.) It is therefore reasonable to assume

Assumption 1. $w_i \approx \log_2(p_i) \approx 0.5\log_2(n)$ and, similarly, $g_i \approx 0.5\log_2(p_i) \approx 0.25\log_2(n)$ for $i = 1, 2$.

Example 1. Let $n \approx 0.7 \cdot 2^{1024}$ *and* $R = 2^{512}$. *If we assume* $p_i/R \approx \beta$ *then* $E(X_{u_2} - X_{u_1}) \approx -107\,c_{ER}$ *in Case B, and* $E(X_{u_2} - X_{u_1}) \approx -214\,c_{ER}$ *in Case C.*

The basic scheme of our attack is stated below. It will be completed in Sect. 5. Note that $-c_{ER}\log_2(n)\beta/16 \approx 0.5[(E(X_{u_2} - X_{u_1} \mid \text{Case A is true}) + (E(X_{u_2} - X_{u_1} \mid \text{Case B is true})]$.

The attack — basic scheme

Phase 1: Choose an integer u with $\beta R \le u < n$ and set (e.g.) $\Delta := 2^{-6}R$

$u_2 := u, \quad u_1 := u_2 - \Delta$

while $\left(T(u_2) - T(u_1) > -c_{ER}\frac{\log_2(n)\beta}{16}\right)^{**}$ do {

$u_2 := u_1, \quad u_1 := u_1 - \Delta$ }

Phase 2: while $(u_2 - u_1 > 1000)$ do {

$u_3 := \lfloor (u_1 + u_2)/2 \rfloor$

if$\left(T(u_2) - T(u_3) > -c_{ER}\frac{\log_2(n)\beta}{16}\right)$ then $u_2 := u_3$(decision for Case A)

else $u_1 := u_3$ (decision for Case B or C)}

Phase 3: Compute $\gcd(u, n)$ for each $u \in \{u_1 + 1, \ldots, u_2\}$.

** The attacker believes that Case A is correct

5 Error Probabilities, Error Detection, and Correction

In Sect. 4 we formulated the basic scheme of our attack. Next, we approximate the probabilities that the attacker decides for Case A although Case B or C was correct and, vice versa, that Case A was correct but the attacker decides for Case B or C. Since we need not distinguish between Cases B and C we will restrict our attention to the Cases A and B. Note that Case C is rather unlikely and if it occurs the situation for the attacker obviously is even better than in Case B.

Decisions were based on time differences $T(u_2) - T(u_1)$ (or, equivalently, on $T(u_2) - T(u_3)$, resp.) which we viewed as realizations of $N(\mu(u_1) - \mu(u_2), \sigma(u_1)^2 + \sigma(u_2)^2)$-distributed random variables $X_{u_2} - X_{u_1}$. Again we assume $u_2 - u_1 \ll p_1, p_2$, and p_j denotes the prime factor of which in Phase 2 a multiple is contained in $\{u_1 + 1, \ldots, u_2\}$. Clearly, as $u_2 (\mathrm{mod}\, p_i)$ and $u_1 (\mathrm{mod}\, p_i)$ (or $u_3 (\mathrm{mod}\, p_i)$, resp.) are not known we cannot compute the exact variances. Note that

$$u_2 (\mathrm{mod}\, p_j), u_3 (\mathrm{mod}\, p_j) \approx 0 \qquad \text{Phase 2, Case A} \qquad (13)$$

$$u_2 (\mathrm{mod}\, p_j) \approx 0,\ u_3 (\mathrm{mod}\, p_j) \approx p_j \qquad \text{Case B} \qquad (14)$$

which will be put in the respective variance and covariance terms of (10). Otherwise the factor $u_k (\mathrm{mod}\, p_i)$ is not under control so that we approximate the respective variance term $\mathrm{pr}_i(u_k)(1 - \mathrm{pr}_i(u_k))$ by its average value

$$\frac{2R}{p_i}\int_0^{p_i/2R} x(1-x)\,dx = \frac{p_i}{4R} - \frac{p_i^2}{12R^2} \quad . \qquad (15)$$

Then we approximate p_1/R and p_2/R by β. Similarly, the covariance terms $\mathrm{cov}_{i;\mathrm{MQ}}(u_k)$ and $\mathrm{cov}_{i;\mathrm{QM}}(u_k)$ are approximated by their average values $\beta^4/48 - \beta^2/12$ and $\beta^3/20 - \beta^2/12$. Further, using Assumption 1 elementary computations lead to $\sigma_{\mathrm{MM}}(u_1)^2 + \sigma_{\mathrm{MM}}(u_2)^2$ (or, $\sigma_{\mathrm{MM}}(u_3)^2 + \sigma_{\mathrm{MM}}(u_2)^2$, resp.)

$$\approx \begin{cases} \log_2(n){c_{\mathrm{ER}}}^2\left(\frac{11\beta}{12} - \frac{31\beta^2}{36} + \frac{\beta^3}{10} + \frac{23\beta^4}{168}\right) & \text{Phase 1, Case A} \\ \log_2(n){c_{\mathrm{ER}}}^2\left(\frac{19\beta}{24} - \frac{47\beta^2}{72} + \frac{\beta^3}{20} + \frac{13\beta^4}{112}\right) & \text{Phase 2, Case A} \\ \log_2(n){c_{\mathrm{ER}}}^2\left(\frac{11\beta}{12} - \frac{127\beta^2}{144} + \frac{\beta^3}{10} + \frac{53\beta^4}{336}\right) & \text{Case B .} \end{cases} \tag{16}$$

As usually, let $\Phi(\cdot)$ denote the cumulative distribution function of the $N(0,1)$-distribution. From (12) we derive the approximate average error probability for a single decision

$$p_{\mathrm{err}} \approx \Phi\left(-\frac{c_{\mathrm{ER}} \log_2(n)\beta}{16\sqrt{\sigma(u_{1|3})^2 + \sigma(u_2)^2}}\right) . \tag{17}$$

Example 2. If ${\sigma_{\mathrm{CRT}}}^2$ is negligible average error probabilities (Phase 1, Case A; Phase 2, Case A; Case B) are about
(i) 0.00094, 0.00097, and 0.00087 for $n \approx 0.7 \cdot 2^{1024}$ and $R = 2^{512}$.
(ii) 0.00022, 0.00027, and 0.00021 for $n \approx 0.9 \cdot 2^{1024}$ and $R = 2^{512}$.
(iii) 0.000005, 0.000005, and 0.000005 for $n \approx 0.7 \cdot 2^{2048}$ and $R = 2^{1024}$.

The error probability for a single decision decreases if the modulus n or the parameter β increases. Although for realistic modulus size n the probability for an erroneous decision is rather small we cannot be sure that all decisions within an attack are correct. (Of course, in Phase 2 a large sequence of decisions for Case A may be an indicator for an erroneous decision in the past.) However, at any instant within Phase 2 we can verify with high probability whether our decisions have been correct so far, i.e. whether the interval $\{u_1+1, \ldots, u_2\}$ really contains a multiple of p_1 or p_2. We just have to apply our decision rule from the basic scheme in Sect. 4 to a time difference for neighbouring values of u_1 and u_2, resp., e.g. to $T(u_2 - 1) - T(u_1 + 1)$. If this leads to the same decision it is confirmed with overwhelming probability that the interval $\{u_1 + 1, \ldots, u_2\}$ truly contains a multiple of p_1 or p_2. (Thus, we call $\{u_1 + 1, \ldots, u_2\}$ a *confirmed interval.*) Otherwise, evaluate a third time difference. If the third decision confirms that $\{u_1 + 1, \ldots, u_2\}$ contains a multiple of p_1 or p_2 then we have established a confirmed interval after all. If not, we have to go back to the preceding confirmed interval or at least to the first "close" decision thereafter to restart the attack at this point with a neighbouring value $\bar{u}$ of the one previously used. Anyway, it is indispensable to complete the basic scheme by regular attempts to establish confirmed intervals, the first at the beginning if Phase 2.

Now let suggestively denote $p_{\mathrm{err;1A}}$, $p_{\mathrm{err;2A}}$ and $p_{\mathrm{err;B}}$ the error probabilities for a single decision in the various cases. If we choose the starting value u_1 randomly, for $\Delta = 2^{-6}R$ Phase 1 requires $(64\beta/3) + 1$ time measurements on average if all decisions are correct. An erroneous decision for Case B or C within

Phase 1 should (at a cost of 4 time measurements) immediately be detected when trying to establish $\{u_1+1,\ldots,u_2\}$ as a confirmed interval. If Case B or C is correct (Phase 1 could end here!) an error costs some extra time measurements but the attack will find a multiple of the other prime factor. Altogether, Phase 1 costs about

$$1+\frac{64\beta}{3}(1+4p_{\mathrm{err;1A}})(1-p_{\mathrm{err;B}})\sum_{k=1}^{\infty}kp_{\mathrm{err;B}}^{k-1}=1+\frac{64\beta}{3}\cdot\frac{1+4p_{\mathrm{err;1A}}}{1-p_{\mathrm{err;B}}} \tag{18}$$

time measurements. Now assume that the attacker wants to establish a confirmed interval each time after s decisions. If s consecutive decisions after a confirmed interval are correct it requires $s+2$ (sometimes $(s+4)$) time measurements to establish the next confirmed interval. This event occurs with average probability $\bar{q}\approx(1-\bar{p}_{\mathrm{err}})^s$ where $\bar{p}_{\mathrm{err}}=0.5(p_{\mathrm{err;2A}}+p_{\mathrm{err;B}})$. If any of the s decisions was wrong the attacker has performed $s+4$ time measurements "for nothing" as he has to restart his attack from the preceding confirmed interval. Similar as above one concludes that establishing a confirmed interval within Phase 2 costs about $(s+4)/\bar{q}-2\bar{q}$ time measurements. Consequently, on average the whole attack (Phase 1 and Phase 2) needs about

$$1+\frac{64\beta}{3}\cdot\frac{1+4p_{\mathrm{err;1A}}}{1-p_{\mathrm{err;B}}}+\frac{0.5\log_2(n)-16}{s}\left(\frac{s+4}{\bar{q}}-2\bar{q}\right) \tag{19}$$

time measurements. The attacker should, of course, choose a parameter value s for which (19) is minimal. If we assume that ${\sigma_{\mathrm{CRT}}}^2$ is negligible for $n\approx 0.7\cdot 2^{1024}$ and $R=2^{512}$, for example, the parameter $s=46$ is optimal. The whole attack then requires about $560\approx 0.55\log_2(n)$ time measurements on average. Similarly, for $n\approx 0.7\cdot 2^{512}$ ($n\approx 0.5\cdot 2^{1024}$, $n\approx 0.9\cdot 2^{1024}$, $n\approx 0.7\cdot 2^{2048}$) for $s=11$ ($s=22$, $s=91$, $s=625$) about $0.71\log_2(n)$ ($0.60\log_2(n)$, $0.53\log_2(n)$, $0.51\log_2(n)$) time measurements are necessary. We did neither take the rare event into account that it erroneously failed to establish a confirmed interval (due to two wrong verifying decisions) nor that the preceding confirmed interval was erroneously established (see Remark 5) as their influence on (19) is small.

Remark 5. (i) If it fails to establish a confirmed interval at a certain stage of the attack for the third time it seems to be likely that the preceding confirmed interval had erroneously been confirmed (rare event!). To avoid a deadlock one simply jumps back to the last one confirmed interval.
(ii) Neighbouring values of the optimal parameter s do not yield considerably worse results.

Remark 6. In Phase 2 of our attack we successively recover the binary representation of an integer multiple of one prime factor p_j. If the attacker starts with $u_1\in[\beta R,R]$ it is likely (for $\beta>\sqrt{0.50}$ it is sure) that he will find p_j rather than a multiple of it. Indeed, if the attacker knows at least $0.25\log_2(n)$ high-order bits of p_j he may refrain from further time measurements but compute the remaining bits with a lattice-based algorithm introduced in [2] (Sect. 10 and 11). Its

running time is polynomial in $\log_2(n)$. Making use of Coppersmith's algorithm obviously nearly halves the number of time measurements. As $u(\mathrm{mod}\, p_j) = u$ for $u < p_j$ due to (7) we recommend to avoid values u_1, u_2, u_3 which are multiples of large powers of 2.

Remark 7. (i) As the terms $\Psi_i(y_i)$ and $\Psi_{i*}(\mathrm{temp})$ usually are interpreted as Montgomery multiplications with factors $a' := y_i$ and $b' := R^2(\mathrm{mod}\, p_i)$ (precomputed value!) and $a' :=$ temp and $b' := 1$, resp., their cumulative variance is negligible. The variance of the reductions $y \mapsto y(\mathrm{mod}\, p_i)$ (e.g. computed with Barrett's reduction algorithm) and of the final CRT computation $b_1x_1 + b_2x_2(\mathrm{mod}\, n)$ depends on the chosen algorithms but should be small in general.
(ii) So far we have tacitly assumed that the attacker is able to measure the running times exactly. A $N(0, {\sigma_{\mathrm{Err}}}^2)$-distributed random measurement error increases the variance of $X_{u_2} - X_{u_1}$ by $2{\sigma_{\mathrm{Err}}}^2$ which in turn increases error probability. Similarly, approximately normally distributed random external influences (or, equivalently, randomly chosen dummy operations) increase the variance of $X_{u_2} - X_{u_1}$ by $2{\sigma_{\mathrm{Ext}}}^2$.
(iii) If ${\sigma_{\mathrm{Err}}}^2 + {\sigma_{\mathrm{Ext}}}^2 + {\sigma_{\mathrm{CRT}}}^2$ is not negligible this does not prevent our attack. In fact, $E(X_{u_2} - X_{u_1})$ and thus the decision rule remains unchanged and (17) remains valid. (Of course, if $\gamma := 2({\sigma_{\mathrm{Err}}}^2 + {\sigma_{\mathrm{Ext}}}^2 + {\sigma_{\mathrm{CRT}}}^2)/(\sigma_{\mathrm{MM}}(u_{1|3})^2 + \sigma_{\mathrm{MM}}(u_2)^2)$ is too large the attack may become *practically infeasible*.) Anyway, the attacker has to establish more confirmed intervals. For $n \approx 0.7 \cdot 2^{1024}$, $R = 2^{512}$ and $\gamma = 1.0$, e.g., using $s = 11$ requires about 730 time measurements. For large γ it may be necessary to apply sequential sampling methods (see Sect. 7) or at least to apply the decision rule from the basic scheme at each step separately to three time differences, e.g. to $T(u_2) - T(u_1), T(u_2 + 1) - T(u_1 - 1)$ and $T(u_2 + 2) - T(u_1 - 2)$ (reuse existing time measurements!). The attacker decides for the majority of these (pre-)decisions which doubles to triples the number of time measurements while the probability for a wrong decision decreases from p_{err} to $(3 - 2p_{\mathrm{err}}) * {p_{\mathrm{err}}}^2$.

6 Experimental Results

We implemented modular exponentiation with CRT and Montgomery multiplication in software. Output was the total number of extra reductions within the Montgomery multiplications. (Recall that we are only interested in time differences.) This scenario corresponds to a "real" timing attack where $\sigma^2_{\mathrm{Err}} + \sigma^2_{\mathrm{Ext}} + {\sigma_{\mathrm{CRT}}}^2$ is negligible since then differences in running times are proportional to differences in the respective numbers of extra reductions. Note that the error probabilities then do not depend on the constants c and c_{ER}. Hence its efficiency does not depend on specific hardware characteristics of the attacked device.

We carried the attack through for various 1024 bit moduli with randomly chosen primes $\in [0.8, 0.85] * 2^{512}$, private key $d = 3^{-1}(\mathrm{mod}\ (p_1 - 1)(p_2 - 1))$, and $R = 2^{512}$. We always started our attack with $u = 2R$. We used the basic scheme

introduced at the end of Sect. 4. Additionally, we established confirmed intervals at the beginning of Phase 2 and then always after 42 decisions. If it failed to establish a confirmed interval at the same stage of the attack for the third time we restarted the attack from the last but one confirmed interval (see Remark 5). On average, about 570 time measurements were needed to carry through an attack. All attacks were successful. We did not make use of Coppersmith's algorithm mentioned in Remark 6. Coppersmith's algorithm would have reduced the average number of time measurements to about 300.

7 Extension to Advanced Exponentiation Algorithms

Many RSA-implementations use more efficient exponentiation algorithms than square and multiply (see e.g. [Mov], Sect. 14.6.1). Therefore, usually b-bit-tables ($b > 1$) are generated which contain powers of the respective base. Combined with CRT and Montgomery's algorithm b-bit table i ($i = 1, 2$) stores $\Psi_i(v)(= u(\mathrm{mod}\, p_i)), \Psi_i(v^2), \ldots, \Psi_i(v^{2^b-1})$ with $v = uR^{-1}(\mathrm{mod}\, n)$, or at least a subset of these values. Using a b-ary exponentiation scheme ([Mov], Alg. 14.82 and 14.83) b modular squarings are accompanied by only one multiplication with a table entry. Our attack can be extended to those table methods. The underlying idea is simple but due to lack of space we can only sketch the technical details. For the sake of efficiency we recommend to make use of Coppersmith's algorithm.

For b-ary exponentiation schemes the essential part of the exponentiation $(uR^{-1})^d(\mathrm{mod}\, n)$ requires about $\log_2(n) - 2$ squarings and $\log_2(n)/b2^{b+1}$ Montgomery multiplications with both $u(\mathrm{mod} p_1)$ and $u(\mathrm{mod} p_2)$. Additionally, $\log_2(2^b - 2)/b2^b$ multiplications with $\Psi_i(v^k)$ are necessary where (i, k) range through $\{1, 2\} \times \{2, \ldots, 2^b - 1\}$. If the table entries are calculated straightforward, i.e. if $\Psi_i(v^{k+1}) = \Psi_{i*}(\Psi_i(v^k)\Psi_i(v))$ for $k = 2, \ldots, 2^b - 1$ ([Mov], Alg. 14.82), then this additionally costs $2^b - 3$ Montgomery multiplications with both $u(\mathrm{mod}\, p_1)$ and $u(\mathrm{mod}\, p_2)$. (Concerning the probability for an extra reduction the computation of $\Psi_i(v^2) = \Psi_{i*}(\Psi_i(v)^2)$ should be viewed as a squaring.)

As the standard variant (attacking the square and multiply exponentiation scheme) also the extended version exploits time difference $T(u_2) - T(u_1)$. A careful computation yields

$$E(X_{u_2} - X_{u_1}) \approx \begin{cases} 0 & \text{in Case A} \\ -\frac{c_{ER} p_i}{2R}\left(\frac{\log_2(n)}{b2^{b+1}} + 2^b - 3\right) & \text{in Case B} \\ -2\frac{c_{ER} p_i}{2R}\left(\frac{\log_2(n)}{b2^{b+1}} + 2^b - 3\right) & \text{in Case C.} \end{cases} \tag{20}$$

Assuming $p_i/R \approx \beta$ we derive the following *predecision rule*: Decide for case A iff $T(u_2) - T(u_1) < -0.25 c_{ER} \beta \left(\frac{\log_2(n)}{b2^{b+1}} + 2^b - 3\right)$.

Similarly as in (10) we first express $\sigma_{MM}(u)^2$ as a sum of variances and covariances. Especially, $\mathrm{cov}_{i;MM}(u) = 4\mathrm{pr}_i(u)^3 - \mathrm{pr}_{*i}^2$ with average value $\beta^3/24 - \beta^2/12$. Note that the particular variances and covariances equal the expressions in (10) with $\Psi_i(v^k)/2R$ instead of $\mathrm{pr}_i(u)$. As in Sect. 5 we approximate these

variance and covariance terms in $\sigma_{\mathrm{MM}}(u_{1|3})^2+\sigma_{\mathrm{MM}}(u_2)^2$ by their average values unless $\Psi_i(v^k) = u_k(\bmod p_j)$ where we use (13) and (14). For simplicity we again assume that ${\sigma_{\mathrm{CRT}}}^2$ is negligible. Then for $n \approx 0.7 \cdot 2^{1024}$, $R = 2^{512}$ and $b = 2$, for example, we obtain approximate error probabilities 0.204 (Phase 1, Case A), 0.204 (Phase 2, Case A), and 0.204 (Case B).

To perform a successful attack, however, we need more trustworthy decisions. Therefore we apply the predecision rule to several time differences $T(u_{2(1)}) - T(u_{1(1)}), T(u_{2(2)}) - T(u_{1(2)}), \ldots$ where $u_{1(1)}, u_{1(2)}, \ldots$ and $u_{2(1)}, u_{2(2)}, \ldots$ denote arbitrary bases (if possible, reuse existing time measurements) within intervals I_1 and I_2 around the lower and the upper bound of the interval $[u_1, u_2)$ (or $[u_3, u_2)$ within Phase 2 of our attack, resp.). (As we use Coppersmith's algorithm $u_2 - u_1, u_2 - u_3 \geq R^{0.5}$ the interval lengths may be chosen fairly large.) To minimize the number of time measurements we use sequential sampling. That is, we proceed applying our predecision rule until the number of predecisions for Case A and the number against Case A differ by a fixed integer $N > 1$. Our decision rule, of course, is to accept the majority of these predecisions. The probability for a wrong decision and the expected number of predecisions follow from formulas 2.4 and 3.4 in Chap. XIV of [4]. (Therefore we interpret the decision procedere as a gambler's ruin problem with initial capital N. A wrong predecision reduces, a correct predecision increases the capital by 1 unit.) For our example we choose $N = 3$ which leads to approximate error probabilities for a single decision $p_{\mathrm{err,1A}} \approx 0.017$, $p_{\mathrm{err,2A}} \approx 0.017$ and $p_{\mathrm{err,B}} \approx 0.016$. The expected number of predecisions needed per decision is about 6.0 in all cases.

The attack itself is organised as in Sect. 4 and 5. Replacing the nominator "$0.5\log_2(n)-16$" of (19) by "$0.25\log_2(n)-6$" (we use Coppersmith's algorithm), choosing a parameter s minimizing this term (in our example, $s = 10$) and multiplying the obtained result by the average numbers of predecisions yields a first estimate for the total number of time measurements needed for our attack (=1920 in our example). However, decisions which are not used to establish a confirmed interval reuse existing time measurements. Of course, if the respective earlier decision had needed fewer predecisions this obviously costs some additional time measurements. (Its mean value may be derived from the generating function for the number of predecisions needed for one decision ([4], Sect. XIV.4) or simply be estimated with a stochastic simulation.) In our example we have to augment the number of time measurements from 1920 to about 2320. For $b = 3$, $b = 4$ and $b = 5$ altogether about 11160 (with $N = 5$ and $s = 5$), 17700 (with $N = 7$ and $s = 6$), 7050 (with $N = 4$ and $s = 5$) time measurements, resp., are necessary if again $n \approx 0.7 \cdot 2^{1024}$ and $R = 2^{512}$. The last result may be surprising at first sight but for $b = 5$ computing the tables entries requires 29 multiplications with both $u(\bmod p_1)$ and $u(\bmod p_2)$.

To improve efficiency at least for $b = 5$ a modified b-ary exponentiation may be used ([Mov], Alg. 14.83) which only stores $\Psi_i(y^k)$ for odd exponents k. Building up table i costs $2^{b-1} - 1$ multiplications with $\Psi_i(v^2)$ and one squaring with $\Psi_i(v)$. For $b = 5$ (and $n \approx 0.7 \cdot 2^{1024}$) the attack requires about 28200 time measurements. For $b = 4$ the situation for the attacker is more comfortable than

in the non-modified case since on average about 32 Montgomery multiplications with both $u(\bmod p_1)$ and $u(\bmod p_2)$ are carried out. About 7250 time measurements should be sufficient. We finally remark that sliding window exponentiation ([Mov], Alg.14.85) can be attacked with similar methods.

8 Fields of Application

As our attack requires chosen input it cannot used be used to attack signature applications with fixed padding. Our attack works, however, if the attacker can choose the complete base y provided that there is no integrity check at all or if random padding bits are used (to prevent the Bellcore attack ([1])), eventually combined with mild integrity conditions (e.g. given by two information bytes). If the attacker then would like to measure $T(u)$ he first uses a PC or a laptop to determine the integer z with smallest absolute value such that $(u+z)R^{-1}(\bmod n)$ meets the integrity condition. Then he measures $T(u+z)$ instead of $T(u)$.

However, our attack can always be applied if $y^d(\bmod n)$ decrypts a secret message $y := r^e(\bmod n)$ where e denotes the public key of the recipient. (The message r might contain a symmetric session key, for example.) Of course, the integrity of r cannot be checked until the exponentiation $y^d(\bmod n)$ has been carried out. However, we are not interested in $y^d(\bmod n)$ itself but in the respective running time (eventually inclusive an integrity check). Hence we do not need to care about integrity conditions.

9 Countermeasures

The most obvious way to prevent our attack is to carry out an extra reduction within each Montgomery multiplication (if not needed by the algorithm then as a dummy operation). Alternatively, provided that R is sufficiently large (compared with the moduli p_i) the extra reduction step may be missed out entirely ([10,5]). In fact, the intermediate results of the respective exponentiation algorithm then are bounded by $2p_i$ and the reduction will be automatically carried out within the final operation temp $\mapsto \Psi_{i*}(\text{temp})$. Using any of these countermeasures, of course, it has to be taken care that eventual time differences caused by the remaining arithmetical operations do not reveal the factorization of n or the secret exponent d.

A more general approach to prevent timing attacks is to use blinding techniques ([6], Sect. 10). Instead of $y^d(\bmod n)$ the device then internally computes $(yh_a(\bmod n))^d(\bmod n)$, followed by a modular multiplication with $h_b := h_a^{-d}(\bmod n)$. To protect the blinding factors h_a and h_b themselves against timing attacks they are updated before the next exponentiation via $h_a \mapsto h_a^2(\bmod n)$ and $h_b \mapsto h_b^2(\bmod n)$.

10 Concluding Remarks

In this article we introduced and investigated a new type timing attack which works if RSA exponentiation with the secret exponent uses CRT and Montgomery's algorithm. Our attack is very efficient and (at the expense of efficiency) tolerates measurement errors and variance caused by arithmetical operations besides the Montgomery multiplications or external influences. The central idea of our attack may also be transferred to CRT implementations using other multiplication algorithms than Montgomery's provided that mean or variance of the time needed for a multiplication of $u \in Z_{p_i}$ with a randomly chosen cofactor is significantly different for $u \approx 0$ and $u \approx p_i$. As a consequence, also for RSA applications using the CRT either constant running times or blinding techniques are indispensable.

Acknowledgement

The author wants to thank the referees for valuable comments and suggestions which helped to improve the presentation of this paper.

References

1. D. Boneh, R.A. Demillo, R.J. Lipton: On the Importance of Checking Cryptographic Protocols for Faults. In: W. Fumy (ed.): Advances in Cryptology — Eurocrypt '97, Lecture Notes in Computer Science, Vol. **1233**. Springer, New York (1997) 37–51.
2. D. Coppersmith: Small solutions to polynomial equations, and low exponent RSA vulnerabilities. J. Cryptology **10** (no. 4) (1997) 233–260.
3. J.-F. Dhern, F. Koeune, P.-A. Leroux, P.-A. Mestré, J.-J. Quisquater, J.-L. Willems: A Practical Implementation of the Timing Attack. In: J.-J. Quisquater, B. Schneier (eds.): CARDIS 1998, Third Smart Card Research and Advanced Application Conference (PreProceedings). Université catholique de Louvain (1998).
4. W. Feller: An Introduction to Probability Theory and Its Applications (Vol. 1). 3rd edition, revised printing, Wiley, New York (1970).
5. G. Hachez, J.-J. Quisquater: Montgomery Exponentiation with no Final Subtractions: Improved Results. To appear in: Ç.K. Koç, C. Paar (eds.): Workshop on Cryptographic Hardware and Embedded Systems, 2000 (Proceedings).
6. P. Kocher: Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS and Other Systems. In: N. Koblitz (ed.): Advances in Cryptology – Crypto '96, Lecture Notes in Computer Science, Vol. **1109**. Springer, Heidelberg (1996), 104–113.
7. A.J. Menezes, P.C. van Oorschot, S.C. Vanstone: Handbook of Applied Cryptography. CRC Press, Boca Raton (1997).
8. P.L. Montgomery: Modular Multiplication without Trial Division. Math. Comp. **44** (no. 170) (1985) 519–521.
9. W. Schindler: Optimized Timing Attacks against Public Key Cryptosystems. Submitted.
10. C.D. Walter: Montgomery's Exponentiation Needs no Final Subtractions. Electronics letters **35** (no. 21) (1999), 1831–1832.

A Comparative Study of Performance of AES Final Candidates Using FPGAs*

Andreas Dandalis[1], Viktor K. Prasanna[1], and Jose D.P. Rolim[2]

[1] University of Southern California, Los Angeles CA 90089, USA
{dandalis,prasanna}@halcyon.usc.edu
http://maarcII.usc.edu
[2] Centre Universitaire d'Informatique, Université de Genève
24 Rue General Dufour, 1211 Genève 4, Switzerland
Jose.Rolim@cui.unige.ch

Abstract. In this paper we study and compare the performance of FPGA-based implementations of the five final AES candidates (MARS, RC6, Rijndael, Serpent, and Twofish). Our goal is to evaluate the suitability of the aforementioned algorithms for FPGA-based implementations. Among the various time-space implementation tradeoffs, we focused primarily on time performance. The time performance metrics are throughput and key-setup latency. Throughput corresponds to the amount of data processed per time unit while the key-setup latency time is the minimum time required to commence encryption after providing the input key. Time performance and area requirement results are provided for all the final AES candidates. To the best of our knowledge, we are not aware of any published results that include key-setup latency results. Our results suggest that *Rijndael* and *Serpent* favor FPGA implementations *the most* since their algorithmic characteristics match extremely well with the hardware characteristics of FPGAs.

1 Introduction

The projected key role of AES in the 21st century cryptography led us to implement the AES final candidates using Field Programmable Gate Arrays (FPGAs). The goal of this study is to evaluate the performance of the AES final candidates on FPGAs and to make performance comparisons. In addition, we evaluate the suitability of reconfigurable hardware as an alternative solution for AES implementations.

In this study, we concentrate only on performance issues. We assume that all the considered algorithms are secure. Time performance and area requirements results are provided for all the final candidates. The time performance metrics are throughput and key-setup latency. Throughput corresponds to the amount of data processed per time unit while key-setup latency is the minimum time

* This research was performed as part of the MAARCII project. This work is supported by the DARPA Adaptive Computing Systems program under contract no. DABT63-99-1-0004 monitored by Fort Huachuca.

Ç.K. Koç and C. Paar (Eds.): CHES 2000, LNCS 1965, pp. 125–140, 2000.

required to commence encryption after providing the input key. Besides the throughput metric, the latency metric is the key measure for applications where a small amount of data is processed per key and key context switching occurs repeatedly.

FPGA technology is a growing area that has the potential to provide the performance benefits of ASICs and the flexibility of processors. This technology allows application-specific hardware circuits to be created on demand to meet the computing and interconnect requirements of an application. Moreover, these hardware circuits can be dynamically modified partially or completely in time and in space based on the requirements of the operations under execution [5,13].

Private-key cryptographic algorithms seem to fit extremely well with the characteristics of the FPGAs. The fine-granularity of FPGAs matches extremely well the operations required by private-key cryptographic algorithms such as bit-permutations, bit-substitutions, look-up table reads, and boolean functions. On the other hand, the constant bit-width required alleviates accuracy-related implementation problems and facilitates efficient designs. Moreover, the inherent parallelism of the algorithms can be efficiently exploited in FPGAs. Multiple operations can be executed concurrently resulting in higher throughput compared with software-based implementations. Finally, the key-setup circuit can run concurrently with the cryptographic core circuit resulting in low key-setup latency time and agile key-context switching.

In our implementations, we focused on the time performance. Our goal was to exploit, for each candidate, the inherent parallelism of the cryptographic core (at the round level) to optimize performance. Moreover, we have exploited the low-level hardware features of FPGAs to enhance the performance of individual required operations. Our throughput results are compared with the FPGA-based results in [9,11]. In [9,11], only the cryptographic core of each algorithm was implemented using FPGAs and, thus, no key-setup latency results were provided. As a result, only throughput comparisons are made with the FPGA-based results in [9,11]. Moreover, our time performance results are compared with the best software-based results in [3,4] and the NSA's ASIC-based results [17].

An overview of FPGAs and FPGA-based cryptography is given in Section 2. In Section 3, general aspects of our implementations are discussed. The implementation results for each algorithm are described in Section 4. In Section 5, a comparative analysis among the results of all the candidates is performed. In addition, comparisons with related work are made. Comparisons with software and ASIC implementations are made in Sections 6 and 7 respectively. Finally, in Section 8, concluding remarks are made.

2 FPGA Overview

Processors and ASICs are the cores of the two major computing paradigms of our days. Processors are general purpose and can virtually execute any operation. However, their performance is limited by the restricted interconnect, datapath, and instruction set provided by the architecture. Conversely, ASICs

are application-specific and can achieve superior performance compared with processors. However, the functionality of an ASIC design is restricted by the designed parameters provided during fabrication. Any update to an ASIC-based platform incurs high cost. As a result, ASIC-based approaches lack flexibility.

FPGA technology is a growing area of research that has the potential to provide the performance benefits of ASICs and the flexibility of processors. Application specific hardware circuits can be created on demand to meet the computing and interconnect requirements of an application. Moreover, these hardware circuits can be dynamically modified partially or completely in time and in space based on the requirements of the operations under execution. As a result, superior performance can be expected compared with the performance of the equivalent software implementation executed on a processor.

FPGAs were initially an offshoot of the quest for ASIC prototyping with lower design cycle time. The evolution of the configurable system technology led to the development of configurable devices and architectures with great computational power. As a result, new application domains become suitable for FPGAs beyond the initial applications of rapid prototyping and circuit emulation. FPGA-based solutions have shown significant speedups (compared with software and DSP based approaches) for several application domains such as signal & image processing, graph algorithms, genetic algorithms, and cryptography among others.

The basic feature underlying FPGAs is the programmable logic element which is realized by either using anti-fuse technology or SRAM-controlled transistors. FPGAs [5,13] have a matrix of logic cells overlaid with a network of wires. Both the computation performed by the cells and the connections between the wires can be configured. Current devices mainly use SRAM to control the configurations of the cells and the wires. Loading a stream of bits onto the SRAM on the device can modify its configuration. Furthermore, current FPGAs can be reconfigured very quickly, allowing their functionality to be altered at runtime according to the requirements of the computation.

2.1 FPGA-Based Cryptography

FPGA devices are a highly promising alternative for implementing private-key cryptographic algorithms. Compared with software-based implementations, FPGA implementations can achieve superior performance. The fine-granularity of FPGAs matches extremely well the operations required by private-key cryptographic algorithms (e.g., bit-permutations, bit-substitutions, look-up table reads, boolean functions). As a result, such operations can be executed more efficiently in FPGAs than in a general-purpose computer.

Furthermore, the inherent parallelism of the algorithms can be efficiently exploited in FPGAs as opposed to the serial fashion of computing in an uniprocessor environment. At the cryptographic-round level, multiple operations can be executed concurrently. On the other hand, at the block-cipher level, certain operation modes allow concurrent processing of multiple blocks of data. For example, in the ECB mode of operation, multiple blocks of data can be

processed concurrently since each data block is encrypted independently. Consequently, if p rounds are implemented, a throughput speed-up of $O(p)$ can be achieved compared with a "single-round" based implementation (one round is implemented and is reused repeatedly). Moreover, by adopting deep-pipelined designs, the throughput can be increased proportionally with the clock speed. On the contrary, in feedback modes of operation (e.g., CBC, CFB), where the encryption results of each block are fed back into the encryption of the current block [14], encryption can not be parallelized among consecutive blocks of data. As a result, the maximum throughput that can be achieved depends mainly on the encryption time required by a single cryptographic round and the efficiency of the implementation of the key-setup component of an algorithm.

Besides throughput, FPGA implementations can also achieve agile key-context switching. Key-context switching includes the generation of the required key-dependent data for each cryptographic round (e.g., subkeys, key-dependent S-boxes). A cryptographic round can commence as soon as its key-dependent data is available. In software implementations, the cryptographic process can not commence before the key-setup process for all the rounds is completed. As a result, excessive latency is introduced making key-context switching inefficient. On the contrary, in FPGAs, each cryptographic round can commence as early as possible since the key-setup process can run concurrently with the cryptographic process. As a result, minimal latency can be achieved.

Security issues also make FPGA implementations more advantageous than software-based solutions. An encryption algorithm running on a generalized computer has no physical protection [14]. Hardware cryptographic devices can be securely encapsulated to prevent any modification of the implemented algorithm. In general, hardware-based solutions are the embodiment of choice for military and serious commercial applications (e.g., NSA authorizes encryption only in hardware) [14].

Finally, even if ASICs can achieve superior performance compared with FPGAs, their flexibility is restricted. Thus, the replacement of such application-specific chips becomes very costly [10] while FPGA-based implementations can be adapted to new algorithms and standards. However, if ultimate performance is essential, ASICs solutions are superior.

3 Implementation and Design Decisions

As a hardware target for the proposed implementations, we have chosen the Xilinx Virtex family of FPGAs. Virtex is a high-capacity, high-speed performance FPGA providing a superior system integration feature set [16]. For mapping onto Virtex devices, we used the Foundation Series v2.1i software development tool. The synthesis and place-and-route parameters of the tool remained the same for all the implementations. All the results were based on placed-and-routed implementations (device speed -6) that included both the key-setup component and the cryptographic core along with their control circuit.

Among the various time-space tradeoffs, our focus was primarily time performance. For each algorithm we have implemented the key-setup component, the control circuitry, and the encryption block cipher for 128-bit data blocks using 128-bit keys. A "single-round" based design was chosen for each implementation. Since one round was implemented, it was reused repeatedly. The key-setup component was processing data in parallel with the cryptographic core. While the cryptographic core was processing the data of the ith round, the key-setup component was calculating the key-dependent data for the $(i + 1)th$ round. As a result, even if an algorithm does not support on-the-fly key generation in the software domain, the key setup can be executed on the fly in FPGAs.

Our goal was to maximize throughput for each candidate algorithm. We have exploited the inherent parallelism of each cryptographic core and the low-level hardware features of FPGAs to enhance the performance. The performance metrics are throughput and key-setup latency. The throughput metric indicates the amount of data encrypted per time unit after the initialization of the algorithm. The key-setup latency denotes the minimum time required to commence encryption after providing the input key. While throughput indicates the bulk-encryption capability of the implementation, key-setup latency indicates the capability of agile key-context switching.

Since one round was implemented and was reused repeatedly, the throughput results correspond to $\frac{128}{n * t_{round}}$, where n and t_{round} are the the number of required rounds and the encryption time per round respectively. Similar performance analysis can be performed for larger sizes of data blocks and keys as well as for implementations that process multiple blocks of data concurrently.

The key-setup latency issue was of primary interest, that is, the cryptographic core had to commence as early as possible. Based on the achieved throughput, we designed the key-setup component to sustain the processing rate of the cryptographic core and to achieve minimal latency. The key-setup latency metric is the key metric for applications where a small amount of data is processed per key and key-context switching occurs repeatedly. In software implementations, the cryptographic process cannot commence before the key-setup process for all the rounds is completed. As a result, the key-setup latency time equals the key-setup time.

To implement efficient key-setup circuits, we took advantage of the embedded memory modules (Block SelectRAM) of the Virtex FPGAs [16]. The Virtex FPGA Series provides dedicated on-chip blocks of true dual-read/write port synchronous RAM, with 4096 memory cells each. Depending on the size of the device, 32-132 Kbits of data can be stored using the Block SelectRAM memory modules. The key-setup circuit utilized these memory modules to pass its results to the cryptographic core. As a result, the cryptographic core could commence as soon as the key-dependent data for the first encryption round is available in the memory modules. Then, during each encryption round, the cryptographic core reads the corresponding data from the memory modules.

For each algorithm, we have also implemented the key-setup circuit and the cryptographic core separately. For all the implementations, the maximum clock

speed of the key-setup circuit was higher than the maximum clock speed of the cryptographic core. Based on the results of these individual implementations, we also provide latency estimates for implementations that clock each circuit at its maximum speed.

Regarding the cryptographic cores, the majority of the required operations fit extremely well in Virtex FPGAs. The permutations and substitutions can be hard-wired while distributed memory can be used as look-up tables. In addition, boolean functions, data-dependent rotations, and addition can be mapped very efficiently onto Virtex FPGAs. Wherever a multiplication with a constant was required, constant coefficient multipliers were utilized to enhance the performance compared with "regular" multipliers. Regular multiplication is required only by the $MARS$ and $RC6$ block ciphers. In both cases, two 32-bit numbers are multiplied and the lower 32-bit of the output are used in the encryption process. We tried the multiplier-macros provided for Virtex FPGAs but we found that they were a performance bottleneck. Besides the excessive latency that was introduced due to the numerous pipeline stages, excessive area was also required since the full multiplier was mapped onto the FPGA. Instead of using these macros, a multiplier that computes partial results in parallel and outputs only the required 32-bits was used. As a result, the latency was reduced by more than 50% and the area requirements were also reduced significantly.

4 Implementation Results

In the following, implementation results as well as relevant performance issues specific to each algorithm are provided. The key-setup latency results are represented both as absolute time and as the fraction of the corresponding encryption time over one 128-bit block of data. In addition, the throughput results are represented both as encryption rate and as encryption rate elaborated on area. Finally, area requirements results are provided for both the key-setup and the cryptographic core circuits. In the following, the order of presenting the algorithms is alphabetic. Detailed algorithmic information for each candidate can be found in [6,12,7,2,15].

4.1 MARS

The $MARS$ block cipher is the IBM submission to AES [6]. The time performance and area requirements results for our $MARS$ implementation are shown in Table 1.

Key Setup. The $MARS$ key expansion procedure expands the input 128-bit key into a 1280-bit key. First a linear-key expansion occurs following by stirring the key-words based on an S-box. Both processes involves simple operations performed repeatedly. However, the final stage of modifying the multiplication key-words involves string-matching operations that are relatively expensive functions. String-matching is an expensive operation compared with the rest of the

Table 1. Implementation Results for MARS

Key-Setup Latency		Throughput		Area Requirements		
					# of slices	
μs	key-setup latency time / block encryption time	MBits / sec	KBits / (sec*slice)	Total	Key-Setup	Cryptographic Core
1.96	3.12	101.88	29.55	6896	2275 (33%)	4621 (67%)

operations required by *MARS*. A compact implementation of string-matching introduces high latency while a high-performance implementation increases the area requirements dramatically. In our implementation, the last stage of the key-expansion process (i.e., string-matching) was not implemented. In spite of this, the introduced key-setup latency was still relatively high (the worst among all the implementations considered in this paper).

Cryptographic Core. The cryptographic core of *MARS* consists of a 16-round cryptographic layer wrapped with two layers of 8-round "forward" and "backward mixing" [6]. The achieved throughput depended mainly on the efficiency of the multiplier (please see Section 3). In our implementation only one round of each layer was implemented that was used repeatedly. The encryption time for one block of data was 32 clock cycles. An interesting feature of our design is that by increasing the utilization factor of the processing stages (i.e. all the three processing stages execute in parallel), the average encryption time for one block of data can be reduced to 16 clock cycles for operation modes that allow concurrent processing of multiple blocks of data (e.g., non-feedback, interleaved).

4.2 RC6

The *RC*6 block cipher is the AES proposal of the RSA Laboratories and R. L. Rivest from the MIT Laboratory for Computer Science [12]. The implemented block cipher corresponds to $w = 32$-bit round keys, $r = 20$ rounds, and $b = 14$-byte input key. The time performance and area requirements results for our *RC*6 implementation are shown in Table 2.

Table 2. Implementation Results for RC6

Key-Setup Latency		Throughput		Area Requirements		
					# of slices	
μs	key-setup latency time / block encryption time	MBits / sec	KBits / (sec*slice)	Total	Key-Setup	Cryptographic Core
0.17	0.15	112.87	42.59	2650	901 (34%)	1749 (66%)

Key Setup. The *RC*6 key setup expands the input 128-bit key into 42 round keys. The key for each round corresponds to a 32-bit word. The key scheduling is fairly simple. The round-keys are initialized based on two constants. We have implemented the initialization procedure using a look-up table since it is the same for any input key. Then, the contents of the look-up table were used to generate the round-keys with respect to the input key. As a result, remarkably low key-setup latency was achieved that was equal to the 15% of the time for encrypting a block of data.

Cryptographic Core. The cryptographic core of *RC*6 consists of 20 rounds. The symmetry and regularity found in the *RC*6 block cipher resulted in a compact implementation. The entire data-block was processed at the same time by using two identical circuits. The achieved throughput depended mainly on the efficiency of the multiplier (please see Section 3).

4.3 Rijndael

The *Rijndael* block cipher is the AES proposal of J. Daemen and V. Rijmen from the Katholieke Universiteit Leuven [7]. The implemented block cipher corresponds to $N_b = 4$, $N_k = 4$, and $N_r = 10$ (i.e., 4×32-bit block data, 4×32-bit key, 10 rounds). The time performance and the area requirements results of our implementation are shown in Table 3.

Table 3. Implementation Results for Rijndael

Key-Setup Latency		Throughput		Area Requirements		
µs	key-setup latency time / block encryption time	MBits / sec	KBits / (sec*slice)	Total	# of slices Key-Setup	Cryptographic Core
0.07	0.20	353.00	62.22	5673	1361 (24%)	4312 (76%)

Key Setup. The *Rijndael* key setup expands the input 128-bit key into a 1408-bit key. Simple operations are used that resulted in extremely low key-setup latency latency. ROM-based look-up tables were utilized to perform the *SubByte* transformation. The achieved latency was the lowest among all the implementations considered in this paper.

Cryptographic Core. The cryptographic core of *Rijndael* consists of 10 rounds. The cryptographic core is ideal for implementations on FPGAs. It combines fine-grain parallelism with look-up table operations. The round transformation can be represented as a look-up table resulting in extremely high speed. We have implemented a ROM-based fully-parallel version of the look-up table. By combining common references to the look-up table, we have achieved a 25% savings

in ROM compared with the straightforward implementation suggested in the AES proposal [7]. The simplicity of the operations and the inherent fine-grain parallelism resulted in the highest throughput among all the implementations. Furthermore, the *Rijndael* implementation had the highest area utilization factor (i.e., throughput per area unit).

4.4 Serpent

The *Serpent* block cipher is the AES proposal of R. Anderson, E. Biham, and L. Knudsen from Technion, Cambridge University, and University of Bergen respectively [2]. The time performance and area requirements results for our *Serpent* implementation are shown in Table 4.

Table 4. Implementation Results for Serpent

Key-Setup Latency		Throughput		Area Requirements		
					# of slices	
µs	key-setup latency time / block encryption time	MBits / sec	KBits / (sec*slice)	Total	Key-Setup	Cryptographic Core
0.08	0.09	148.95	58.41	2550	1300 (51%)	1250 (49%)

Key Setup. The *Serpent* key setup expands the input 128-bit key into a 4224-bit key. First, the input key is padded to 256 bits and then it is expanded to an intermediate key by iterative mixing of the key data. Finally, by using look-up tables, the keys for all the rounds are calculated. The simplicity of the required operations resulted in extremely low key-setup latency (the second lowest among all the implementations considered in this paper).

Cryptographic Core. The cryptographic core of *Serpent* consists of 32 rounds. The round transformation is a linear transform consisting of rotations, shifts, and XOR operations. Neither multiplication nor addition is required. As a result, the lowest encryption time per round and the most compact implementation were achieved among all the implementations. Furthermore, the *Serpent* implementation had the second higher area utilization factor (i.e. throughput per area unit).

4.5 Twofish

The $Twofish$ block cipher is the AES proposal of the Counterpane Systems, Hi/fn, Inc., and D. Wagner from the University of California Berkeley [15]. The time performance and area requirements results of our implementation are shown in Table 5.

Table 5. Implementation Results for Twofish

Key-Setup Latency		Throughput		Area Requirements		
					# of slices	
µs	key-setup latency time / block encryption time	MBits / sec	KBits / (sec*slice)	Total	Key-Setup	Cryptographic Core
0.18	0.25	173.06	18.48	9363	6554 (70%)	2809 (30%)

Key Setup. The $Twofish$ key setup expands the input 128-bit key into a 1280-bit key. Moreover, it generates the key-dependent S-boxes used in the cryptographic core. Four 128-bit S-boxes are generated. Since our goal was to minimize latency, we have implemented a parallel version of the key setup consisting of 24 q_0/q_1 permutation boxes and 2 MDS matrices [15]. Moreover, the RS matrix was implemented for the S-box generation. The matrices are used for "constant matrix"-to-matrix multiplication over $GF(2^8)$. The best known implementation of a constant coefficient multiplier in Virtex FPGAs is by using a look-up table [16]. As a result, low latency was achieved but excessive area was required. The area requirements corresponded to the 70% of the total area. However, by implementing a more compact design (e.g., reusing processing elements), the key-setup latency would increase.

Cryptographic Core. The cryptographic core of $Twofish$ consists of 16 rounds. The structure of the round transformation is similar to the structure of the key-expansion circuit. The only major difference is the S-boxes that the cryptographic core uses.

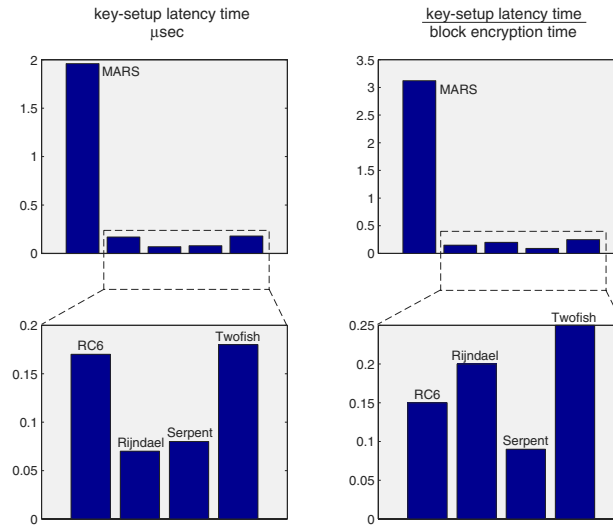
4.6 Key-Setup Latency Improvements

For each algorithm, we have also implemented the key-setup circuit and the cryptographic core separately. For each algorithm, the maximum clock speed of the key-setup circuit was higher than the maximum clock speed of the cryptographic core. Thus, by clocking each circuit at its maximum clock speed, improvement in key-setup latency can be achieved. No additional synchronization hardware is required since we can configure the read/write ports of the Block SelectRAMs having different clock speeds. Compared with implementations using one clock, the key-setup latency time can be reduced by a factor of 1.35, 2.96, 1.43, 1.00, and 1.15 for $MARS$, $RC6$, $Rijndael$, $Serpent$, and $Twofish$ respectively. Clearly, the $RC6$ block cipher can achieve the best key-setup latency improvement by clocking the key-setup and the cryptographic core circuits at their maximum clock speeds. For the $MARS$ block cipher, the result is based on an implementation that does not include the circuit for modifying the multiplication key-words.

5 Comparative Analysis of Our FPGA Implementations

In Table 6, key-setup latency comparisons are made among our FPGA implementations. The comparisons are made in terms of absolute time and the ratio of the key-setup latency time to the time required to encrypt one block of data. The latter metric represents the capability of agile key-context switching with respect to the encryption rate.

Table 6. Key-Setup Latency Comparisons of Our FPGA Implementations

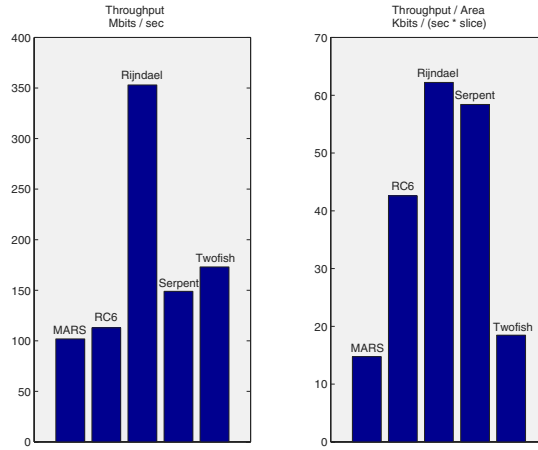


Clearly, *Rijndael* and *Serpent* achieve the lowest key-setup latency times while the latency times for *RC*6 and *Twofish* are higher by a factor of 2.5. As we have mentioned in Section 4, the key-setup latency introduced by *MARS* is the highest. All the algorithms (except *MARS*) achieve key-setup latency time that is equal to the 7-25 % of the time for encrypting one block of data.

In Table 7, throughput comparisons are made among our FPGA implementations. The comparisons are made in terms of the encryption rate and the ratio of the encryption rate to the area requirements. The latter metric reveals the hardware utilization efficiency of each implementation.

Rijndael achieves the highest encryption rate due to the ideal match of its algorithmic characteristics with the hardware characteristics of FPGAs. In addition, the encryption rate of *Rijndael* is higher than the ones achieved by the other algorithms by a factor of $1.7 - 3.12$. Moreover, *Rijndael* also achieves the best hardware utilization. The latter metric combines, for each algorithm, the computational demands in terms of an FPGA implementation with the inherent parallelism of the cryptographic round.

Table 7. Throughput Comparisons of Our FPGA Implementations



Serpent achieves the second best hardware utilization while having the lowest encryption time per round. The latter suggests that, under the same area constraints, *Serpent* can achieve throughput equivalent to *Rijndael* for operation modes that allow concurrent processing of multiple blocks of data. Similar to *Rijndael*, the algorithmic characteristics of *Serpent* matches extremely well with the hardware characteristics of FPGAs.

Finally, in Table 8, area comparisons are made among our FPGA implementations. The comparisons are made in terms of the total area as well as the area required by each of the key-setup and the cryptographic core circuits. *Serpent* and *RC*6 have the most compact implementations. *Serpent* also has the most compact cryptographic core circuit while *RC*6 has the most compact key-setup circuit. For the *MARS* block cipher, the result shown is based on an implementation that does not include the circuit for modifying the multiplication key-words [6].

5.1 Related Work

In [9,11], FPGA implementations of the AES candidate algorithms were described using Virtex devices. However, only the cryptographic core for each algorithm was implemented. No results regarding key-setup were provided. In Table 9, throughput results for [9,11] and our work are shown for encrypting 128-bit data blocks using 128-bit keys. To make a fair comparison, the results shown for [9] correspond to the performance evaluation for feedback modes. In [9], results for non-feedback modes were also provided, which corresponded to implementations that process multiple blocks of data concurrently.

The major difference in the throughput results is the *Serpent* algorithm. By implementing 8 rounds of the algorithm [9], the distribution of the subkeys among consecutive rounds becomes very efficient resulting in 3× speed-up

Table 8. Area Comparisons of Our FPGA Implementations

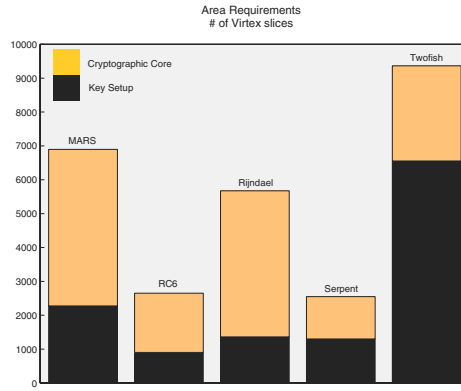


Table 9. Performance Comparisons with FPGA Implementations [9,11]

AES Algorithm	Throughput MBits/sec		
	[9]	Our	[11]
MARS	---	101.88	39.80
RC6	126.50	112.87	103.90
Rijndael	300.10	353.00	331.50
Serpent	444.20	148.95	339.40
Twofish	119.60	173.06	177.30

compared with our "single-round" implementation. For *MARS*, our implementation achieved higher throughput by a factor of 2.5 compared with [11]. The *MARS* block cipher was not implemented in [9]. For *RC*6 and *Rijndael*, all the implementations achieved similar throughput performance. For *Twofish*, the throughput achieved in [11] and in our work is higher than the one in [9] by a factor of 1.5. By combining the throughput results provided in [9,11] and the performance results provided in our work, we can verify that *Rijndael* and *Serpent* favor FPGA implementations *the most* among all the AES candidate algorithms.

6 Comparison with Software Implementations

Our performance results are compared with the best software-based results found in [3] and [4]. In [3], optimized assembly-language implementations on the Pentium II were described for *MARS*, *RC*6, *Rijndael*, and *Twofish*; only throughput results were provided. In [4], *ANSI C*-based implementations on a variety of

Table 10. Performance Comparisons with Software Implementations [3,4]

AES Algorithm	Throughput			Key-Setup Latency		
	MBits/sec		Speed-up	µs		Speed-up
	Software	Our		Software	Our	
MARS	[3] 188.00	101.88	1/1.84	[4] 8.22	1.96	4.19
RC6	[3] 258.00	112.87	1/2.28	[4] 3.79	0.17	22.29
Rijndael	[3] 243.00	353.00	1.45	[4] 2.15	0.07	30.71
Serpent	[4] 60.90	148.95	2.44	[4] 11.57	0.08	144.62
Twofish	[3] 204.00	173.06	1/1.17	[4] 15.44	0.18	85.78

platforms were described for all the AES candidate algorithms; both throughput and key-setup time results were provided.

In Table 10, throughput and key-setup latency comparisons are shown for encrypting 128-bit data blocks using 128-bit keys. Clearly, the FPGA implementations achieve significant reduction in the key-setup latency time by a factor of $4 - 144$. In software implementations, the cryptographic process can not commence before the key-setup process for all the rounds is completed. As a result, the key-setup latency time equals to the key-setup time making key-context switching inefficient. On the contrary, in FPGAs, each cryptographic round can commence as early as possible since the key-setup process can run concurrently with the cryptographic process. As a result, minimal latency can be achieved.

Regarding throughput results, the software implementations achieve higher throughput by a factor of 1.84, 2.28, and 1.17 for $MARS$, $RC6$, and $Twofish$ respectively. The latter algorithms require multiplication operations. Our intuition is that the hardware specialization and parallelism exploited in FPGAs were not enough to outperform the efficiency of the multiplication in software. On the contrary, the FPGA implementations achieved higher throughput by a factor of 1.45 and 2.44 for *Rijndael* and *Serpent* respectively. The latter reconfirms that *Rijndael* and *Serpent* favor FPGA implementations *the most* among the AES candidate algorithms. It is also worthy to mention that *Rijndael* results in one of the fastest implementations in both software and FPGAs. Finally, for operation modes that allow concurrent processing of multiple blocks of data (e.g., non-feedback, interleaved), the parallel fashion of computing in FPGAs can result in higher throughput for all the AES candidate algorithms compared with uniprocessor-based software implementations.

7 Comparison with ASIC Implementations

Our performance results are also compared with the results of ASIC-based implementations described in the NSA's "Hardware Performance Simulations of Round 2 AES Algorithms" [17]. Our time performance results are compared

with the results provided for encrypting 128-bit data blocks using 128-bit keys using iterative architectures. In Table 11, throughput and key-setup latency comparisons are shown for encrypting 128-bit data blocks using 128-bit keys. Clearly, besides our implementations, *Rijndael* achieves the highest throughput in ASICs too. Surprisingly enough, the FPGA implementations for *MARS*, *RC*6, and *Twofish* achieve higher throughput than the ASIC-based counterparts. For one reason, since ASIC technology can provide the ultimate performance, we assume that the resulted speed-ups are due to the design techniques (e.g., inherent parallelism) and the individual components (e.g., multiplier) incorporated in our implementations. For another, the Virtex FPGAs are fabricated on a leading edge 0.18μm, six-layer metal silicon process [16], while a 0.5μm MOSIS-specific technology library was utilized in [17]. Regarding the key-setup latency time, the only major difference is the *RC*6 algorithm where an improvement by a factor of 33.76 has been achieved.

Table 11. Performance Comparisons with ASIC Implementations [17]

AES Algorithm	Throughput			Key-Setup Latency		
	MBits/sec		Speed-up	µs		Speed-up
	NSA ASIC	Our		NSA ASIC	Our	
MARS	56.71	101.88	1.79	9.55	1.96	4.87
RC6	102.83	112.87	1.09	5.74	0.17	33.76
Rijndael	605.77	353.00	1/1.71	0.00	0.07	---
Serpent	202.33	148.95	1/1.35	0.02	0.08	1/4
Twofish	105.14	173.06	1.64	0.06	0.18	1/3

8 Conclusions

In this paper we have provided time performance and area requirements results for the implementations of the five final AES candidates (*MARS*, *RC*6, *Rijndael*, *Serpent*, and *Twofish*) using FPGAs. To the best of our knowledge, we are not aware of any published results that include key-setup latency results. In our implementations, the key-setup process can be performed in parallel with the encryption process regardless of the capability of the software implementation to support on-the-fly key setup. Our implementations suggest that *Rijndael* and *Serpent* favor FPGA implementations *the most* due to the ideal match of their algorithmic characteristics with the characteristics of FPGAs. The *Rijndael* implementation achieves the lowest key-setup latency time, the highest throughput, and the highest hardware utilization. Comparing our results with software [4,3] and ASIC [17] implementations, we verified that *Rijndael*

also achieves the best time performance across different platforms (i.e., ASIC, FPGA, software).

The work reported here is part of the USC MAARCII project (`http://maarcII.usc.edu`). This project is developing novel mapping techniques to exploit dynamic reconfiguration and facilitate run-time mapping using configurable computing devices and architectures. A domain-specific mapping approach is being developed to support instance-dependent mapping [8]. Moreover, computational models and algorithmic techniques are being developed to exploit self-reconfiguration using FPGAs. Finally, the idea of "active" libraries is exploited to develop a framework for automatic dynamic reconfiguration.

References

1. Advanced Encryption Standard, `http://www.nist.gov/aes/`
2. R. Anderson, E. Biham, and L. Knudsen, "Serpent: A Proposal for the Advanced Encryption Standard", AES Proposal, June 1998.
3. K. Aoki and H. Lipmaa, "Fast Implementations of AES Candidates", Third AES Candidate Conference, April 2000.
4. L. E. Bassham III, "Efficiency Testing of ANSI C Implementations of Round 2 Candidate Algorithms for the Advanced Encryption Standard", Third AES Candidate Conference, April 2000.
5. S. Brown and J. Rose, "FPGA and CPLD Architectures: A Tutorial", IEEE Design & Test of Computers, Summer 1996.
6. C. Burwick et al., "MARS - a candidate cipher for AES", AES Proposal, August 1999.
7. J. Daemen, V. Rijmen, "The Rijndael Block Cipher", AES Proposal, September 1999.
8. A. Dandalis, "Dynamic Logic Synthesis for Reconfigurable Devices", PhD Thesis, Dept. of Electrical Engineering - Systems, University of Southern California. Under Preparation.
9. A. J. Elbirt, W. Yip, B. Chetwynd, and C. Paar, "An FPGA Implementation and Performance Evaluation of the AES Block Cipher Candidate Algorithm Finalists", Third AES Candidate Conference, April 2000.
10. D. Fowler, "Virtual Private Networks: Making the Right Connection", Morgan Kaufmann Publishers, Inc., San Francisco, California, 1999.
11. K. Gaj and Pawel Chodowiec, "Comparison of the hardware performance of the AES candidates using reconfigurable hardware", Third AES Candidate Conference, April 2000.
12. R. L. Rivest, M. J. B. Robshaw, R. Sidney, and Y. L. Yin, "The RC6^{TM} Block Cipher", AES Proposal, June 1998.
13. J. Rose, A. El Gamal, and A. Sangiovanni-Vincentelli, "Architecture of Field Programmable Gate Arrays", Proceedings of the IEEE, July 1993.
14. B. Schneier, "Applied Cryptography", John Willey & Sons, Inc., 2nd edition, 1996.
15. B. Schneier, J. Kelsey, D. Whitingz, D. Wagnerx, and C. Hall, "Twofish: A 128-Bit Block Cipher", AES Proposal, June 1998.
16. Virtex Series FPGAs, `http://www.xilinx.com/products/virtex.htm`.
17. B. Weeks, M. Bean, T. Rozylowicz, and C. Ficke, "Hardware Performance Simulations of Round 2 Advanced Encryption Standard Algorithms", Third AES Candidate Conference, April 2000.

A Dynamic FPGA Implementation of the Serpent Block Cipher

Cameron Patterson

Xilinx, Inc.
2300 55th Street, Boulder Colorado 80301, USA
Cameron.Patterson@xilinx.com

Abstract. A JBits implementation of the Serpent block cipher in a Xilinx FPGA is described. JBits provides a Java-based Application Programming Interface (API) for the run-time modification of the configuration bitstream. This allows dynamic circuit specialization based on a specific key and mode (encrypt or decrypt). Subkeys are computed in software and treated as constants in the Serpent datapath. The resulting logic optimization produces a circuit that is half the size and twice the speed of a static, synthesized implementation. With a throughput of over 10 Gigabits per second, the JBits implementation has sufficient bandwidth for SONET OC-192c (optical) networks.

1 Introduction

The United States Department of Commerce defines a standard cryptographic algorithm for non-classified government use, and the Data Encryption Standard (DES) has fulfilled this role since 1977 [1]. DES was intended to be used for no more than 10 years, but the absence of a new standard means that it is still the workhorse private key encryption algorithm. There are, however, compelling reasons to replace DES:

- Exhaustive key search has been a serious threat to DES for several years. Triple DES extends the effective key size, but incurs a significant performance penalty.
- Cryptographic algorithms must provide high performance in software as well as hardware. DES is hardware-oriented. When implemented in software, an n-bit processor performs operations on small subsets of the bits in a word. The fastest publically available DES software that encrypts a single block at a time has a throughput of 15 Mbits/sec on a 200 MHz Pentium [2]. If 64 blocks are encrypted in parallel, then software throughput can be increased to 137 Mbits/sec [3].
- The DES 64-bit block size is insufficient for high bandwidth applications.

In 1997, the National Institute of Standards and Technology (NIST) solicited candidates for a successor to DES, which is to be called the Advanced Encryption Standard (AES) [4]. AES, like DES, will be a private key algorithm (i.e. it uses

Ç.K. Koç and C. Paar (Eds.): CHES 2000, LNCS 1965, pp. 141–155, 2000.

the same key for both encryption and decryption). However, AES defines key sizes of 128, 192, or 256 bits, and doubles the block size to 128 bits. Fifteen algorithms were submitted to the First AES Candidate Conference in August 1998 [5]. One year later, NIST announced the five finalists: MARS, RC6™, Rijndael, Serpent and Twofish.

Field-Programmable Gate Arrays are frequently used to implement cryptographic algorithms [6]. The author previously implemented DES in the Xilinx Virtex™ family [7]. An electronic codebook (ECB) mode throughput in excess of 10 Gbits/sec was achieved using JBits to perform key-specific dynamic circuit specialization [8]. According to the National Security Agency, high speed network encryption may require the use of non-feedback modes such as ECB [9].

Key-specific circuit specialization removes a logic level and the associated routing from the critical path in a DES round. Compared with a static design that uses the same Virtex process and degree of pipelining, speed is increased by over 50%. The reduction in circuit size also decreases power consumption and permits the use of smaller devices. Cheaper packages may be used, since key input pins are no longer required. The resulting implementation has better throughput and lower power than the fastest reported DES ASIC [10]. High volume cost in the Spartan™-II family should be competitive with ASICs.

The author is developing dynamic implementations of the AES finalists, which can provide additional data for the selection process. Serpent was chosen first for several reasons:

- It uses many of the same primitive operations as DES. This allows some of the DES building blocks to be reused.
- The simple round structure means that high clock rates can be achieved with only one pipeline stage per round.
- Decryption requires inverse permutations and linear transformations, which would normally increase the size of a static implementation. A dynamic implementation, however, simply reconfigures the same FPGA resources when either the key or mode changes.
- Speed and size comparisons can be made with a static Virtex design [11].

A dynamic Serpent implementation is twice the speed and half the size of the static design. If both encryption and decryption are required, then the dynamic circuit requires an even smaller fraction of the FPGA resources. ECB mode data throughput is over 10 Gbits/sec. This is the same bandwidth as the dynamic DES design, and is achieved with a similar degree of pipelining, twice the data path width and half the clock rate. Power consumption for Serpent should be about 4 watts, compared with 3.2 watts for DES. Hence, the JBits approach is providing the same performance and cost benefits for Serpent as it did for DES. The remainder of this paper applies dynamic circuit specialization to Serpent.

2 The Serpent Algorithm

Serpent is a substitution-permutation (SP) network that uses 32 rounds. The output of round i is the input to round $(i + 1)$. The algorithm has two different

modes of implementation: standard and bitslice. The standard mode operates on individual bits or groups of four bits, while the bitslice mode improves software efficiency by operating on entire 32-bit words. Serpent software using the bitslice optimization encrypts at roughly 32 Mbits/sec on a 200 MHz Pentium [2].

The bitslice mode was chosen for both software and hardware implementation, since:

- It does not increase the number of lookup tables (LUTs) in the critical path.
- Layout considerations encourage the partitioning of 128-bit data paths into four 32-bit words.
- Like DES, the standard Serpent mode requires permuting the input bits to the first round, and permuting the output bits from the final round. Subkey permutations are also required. Permutations on 128-bit buses consume significant routing resources in an FPGA device, and can increase the routing delay.

Serpent's bitslice mode is to be assumed in the following exposition.

2.1 The Round Function

The rounds are numbered from 0 to 31. Round i has input B_i and output B_{i+1}. The round function is defined as:

$$\begin{aligned} B_{i+1} &= L(S_i(B_i \oplus K_i)) \qquad \text{for } i = 0, \ldots, 30 \\ B_{32} &= (S_{31}(B_{31} \oplus K_{31})) \oplus K_{32} \end{aligned}$$

S_i applies 32 copies of a 4-bit permutation called an S-box. Eight different S-boxes are used, which are derived from the DES S-boxes. Round i applies S-box (i mod 8), so that each S-box is used in four different rounds. The generation of the 128-bit round subkeys K_i is described in Section 2.2. L is a linear transformation. In bitslice mode, it applies left rotations ($<<<$), left shifts ($<<$) and XORs ($\oplus$) to the 32-bit operands X_0, X_1, X_2, and X_3:

$$\begin{aligned} X_0, X_1, X_2, X_3 &:= S_i(B_i \oplus K_i) \\ X_0 &:= X_0 <<< 13 \\ X_2 &:= X_2 <<< 3 \\ X_1 &:= X_1 \oplus X_0 \oplus X_2 \\ X_3 &:= X_3 \oplus X_2 \oplus (X_0 << 3) \\ X_1 &:= X_1 <<< 1 \\ X_3 &:= X_3 <<< 7 \\ X_0 &:= X_0 \oplus X_1 \oplus X_3 \\ X_2 &:= X_2 \oplus X_3 \oplus (X_1 << 7) \\ X_0 &:= X_0 <<< 5 \\ X_2 &:= X_2 <<< 22 \\ B_{i+1} &:= X_0, X_1, X_2, X_3 \end{aligned}$$

2.2 The Key Schedule

If required, the user-supplied key is first padded to 256 bits. This is done by assigning a 1 to the most significant bit, and a 0 to the remaining bits. The key is stored as eight 32-bit words $w_{-8}, \ldots, w_{-1}$. This is used to generate the prekeys $w_0, w_1, \ldots, w_{131}$ with the recurrence:

$$w_i := (w_{i-8} \oplus w_{i-5} \oplus w_{i-3} \oplus w_{i-1} \oplus \phi \oplus i) <<< 11$$

where ϕ is the hexadecimal constant 9E3779B9.[1] Finally, round subkeys are computed from the prekeys by applying the S-boxes as follows:

$$\begin{aligned} K_0 &:= S_3(w_0, w_1, w_2, w_3) \\ K_1 &:= S_2(w_4, w_5, w_6, w_7) \\ K_2 &:= S_1(w_8, w_9, w_{10}, w_{11}) \\ &\vdots \\ K_{31} &:= S_4(w_{124}, w_{125}, w_{126}, w_{127}) \\ K_{32} &:= S_3(w_{128}, w_{129}, w_{130}, w_{131}) \end{aligned}$$

The structure of the rounds and the application of the subkeys are shown in Figures 1 and 2.

2.3 Decryption

Feistel networks such as DES decrypt by simply applying the round subkeys in reverse order. Serpent, however, is not a Feistel network. It also requires applying the inverse of the S-boxes in reverse order, and inverting the linear transformation. Although this represents an area overhead for ASICs and static FPGAs, the same logic and routing resources can simply be reconfigured with a dynamic implementation.

3 Run-Time Reconfiguration

The run-time optimization of FPGAs to the problem instance at hand can have considerable speed and area advantages. For example, a dynamic implementation of the DES algorithm exceeds the performance of the fastest known DES ASIC. In this case, the reduction in circuit complexity more than compensates for the routing overheads associated with FPGAs.

However, most systems do not exploit the run-time reconfiguration (RTR) of SRAM-based FPGAs. This is primarily because there is no support for RTR in the standard design capture, verification and implementation tools. The conventional netlist-based FPGA design flow is the same as for ASICs, and has far too much time and memory overhead to be used in real-time or embedded environments.

[1] This is the fractional part of the golden ratio $(1 + \sqrt{5})/2$.

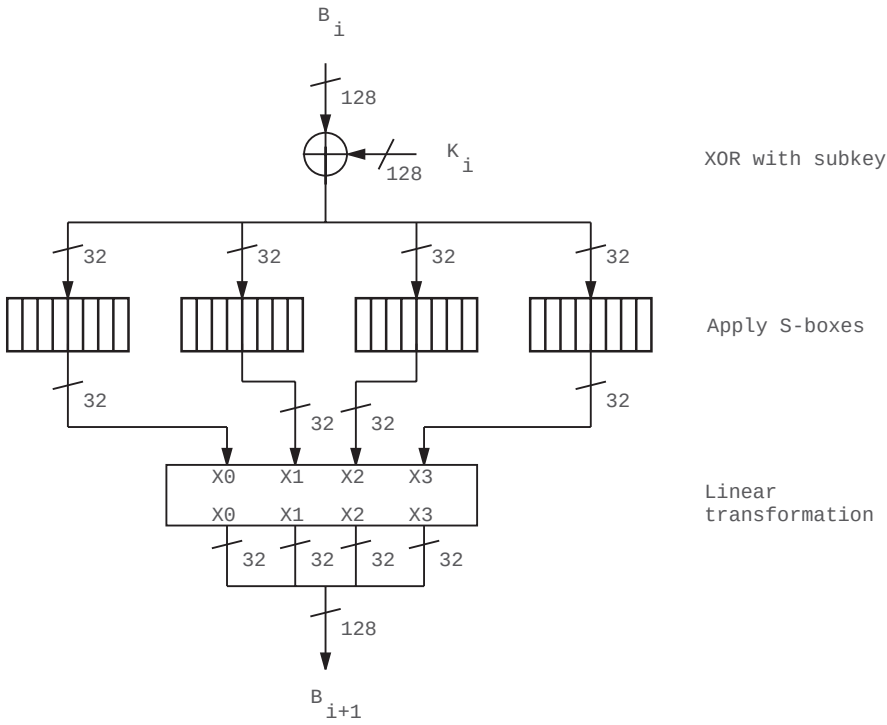


Fig. 1. Structure of Serpent Rounds 0 to 30

RTR is most easily controlled with a microprocessor. Many systems already make use of one or more microprocessors for those operations that do not require hardware speeds. Software can directly create or modify the FPGA's configuration with a suitable Application Programming Interface (API). This model readily supports hardware/software co-design, since the integration of hardware and software occurs early in the development effort.

3.1 JBits

Xilinx provides a Java-based configuration API for the XC4000 and Virtex architectures called JBits [12]. The tiles used in an architecture are defined as JBits classes. For example, Virtex has a Configurable Logic Block (CLB) tile that includes the associated General Routing Matrix (GRM). A CLB instance is treated as an object that is referenced by a row and column. The state of the configurable structures in the CLB can be queried or modified. Figure 3 illustrates the JBits calls required to configure a CLB at (`row`,`col`) as a 4-input 4-output

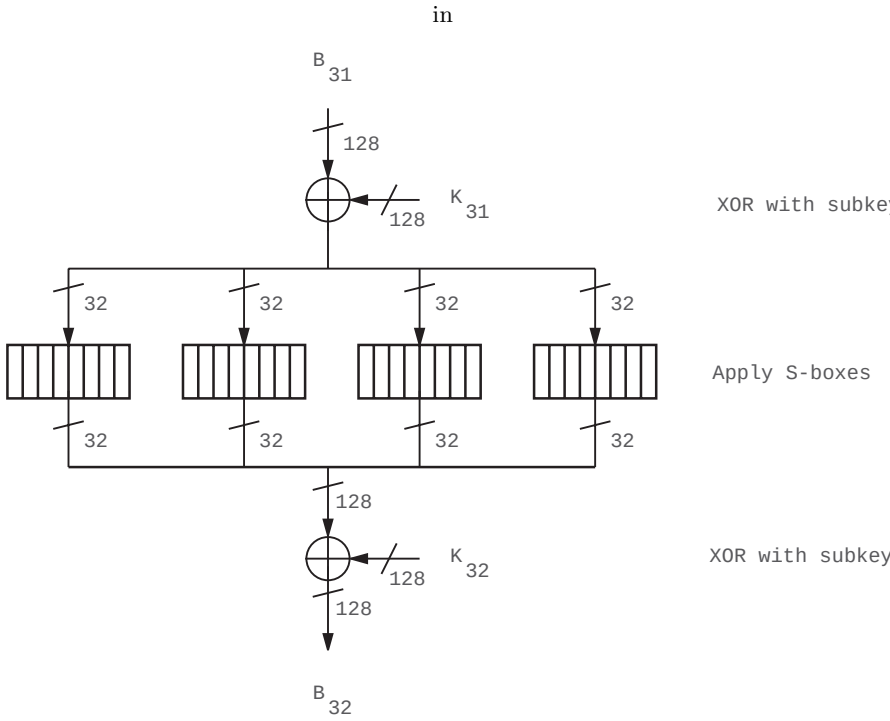


Fig. 2. Structure of Serpent Round 31

registered S-box. There is less code than the equivalent constrained structural netlist of Virtex primitives defined in an HDL. Many of the configuration settings used are the power-up defaults.

Generating a configuration bitstream with JBits generally takes on the order of seconds, compared with minutes or even hours for the Xilinx M2.1 implementation tools. JBits is a physical design tool, and avoids the optimization problems that arise during the logical to physical transformation in conventional CAD flows. All mapping, placement and routing is fully specified in a JBits design.

4 Serpent Implementation

In 1998, Xilinx introduced the Virtex architecture as the successor to the XC4000 family [13]. It uses a 2.5 volt, 0.22 micron, 5 metal layer process. Like the XC4000, it can be characterized as a symmetric array of CLBs surrounded by IOBs. Each CLB contains two slices, where each slice is roughly equivalent to an XC4000 CLB (i.e. it contains two 4-input LUTs, two flip flops, and a carry path). System-level resources such as block RAM and delay-locked loops have also been added. Unlike the XC4000, Virtex supports packet-based partial reconfiguration [14].

```
public void configureSBox(int row, int col) {
  try {
    jbits.set(row, col, S0RAM.DUAL_MODE,     S0RAM.ON);
    jbits.set(row, col, S0RAM.LUT_MODE,      S0RAM.ON);
    jbits.set(row, col, S0Control.X.X,       S0Control.Cin.FOUT);
    jbits.set(row, col, S0Control.Y.Y,       S0Control.Y.GOUT);
    jbits.set(row, col, S0Control.XDin.XDin, S0Control.XDin.X);
    jbits.set(row, col, S0Control.YDin.YDin, S0Control.YDin.Y);
    jbits.set(row, col, S0Control.LatchMode, S0Control.ON);
    jbits.set(row, col, S0Control.Sync,      S0Control.ON);
    jbits.set(row, col, S1RAM.DUAL_MODE,     S1RAM.ON);
    jbits.set(row, col, S1RAM.LUT_MODE,      S1RAM.ON);
    jbits.set(row, col, S1Control.X.X,       S1Control.Cin.FOUT);
    jbits.set(row, col, S1Control.Y.Y,       S1Control.Y.GOUT);
    jbits.set(row, col, S1Control.XDin.XDin, S1Control.XDin.X);
    jbits.set(row, col, S1Control.YDin.YDin, S1Control.YDin.Y);
    jbits.set(row, col, S1Control.LatchMode, S1Control.ON);
    jbits.set(row, col, S1Control.Sync,      S1Control.ON);
  } catch (ConfigurationException ce) {
    System.out.println("S-box configuration error at R" +
                       row + "C" + col);
    System.out.println(ce);
  }
}
```

Fig. 3. Implementing a Registered S-box with JBits

The Virtex CLB is well-suited to the Serpent algorithm. An S-box is specified as a table with a 4-bit input and a 4-bit output. Logic minimization algorithms find little structure in the S-boxes, so it is reasonable to implement an S-box as a set of four single-output LUTs that are driven by the same four inputs. A single Virtex CLB can implement all four LUTs, while the XC4000 architecture would require two CLBs.

In addition, the Virtex segmented routing structure efficiently implements the shift and rotation operations in Serpent's linear transformation stages. Each Virtex horizontal and vertical channel has 96 hex-length and 24 single-length segments, which provides high speed and high density wire permutations.

The same approach is used for Serpent as for DES, namely to investigate the speed, size and power optimizations achievable with dynamic circuit specialization. Two additional considerations also influenced the design:

- An effort was made to make a fair comparison with a static Virtex implementation of Serpent. Both designs use only one pipeline stage per round. The static design includes support for cipher block chaining mode [15], and the dynamic design has resources available for the required XOR gates.
- Wherever practical, the Serpent core was designed to be compatible with the JBits DES core. This was largely achieved for throughput, degree of

pipelining and power consumption, despite the fact that Serpent has twice the datapath width and number of rounds compared with DES. Note that Serpent's larger key size does not increase the pin requirement, since the JBits approach does not require any key-handling circuitry or IOBs.

4.1 Dynamic Circuit Specialization Using JBits

Serpent's key scheduling logic is more complicated than DES, but the datapath is simpler. This results in even greater benefits when the datapath is implemented in hardware and the key schedule is precomputed in software. The logical operations performed in the datapath are permutations on groups of 4 bits, and XORs. As shown in Figure 1, the first 31 rounds require 128 XORs between the data and subkey bits. Let b be a data bit from B_i and k be the corresponding subkey bit from K_i. Since k is a constant for a given encryption key, $\mathrm{XOR}(b, k)$ is either b or $\overline{b}$. An inversion on an S-box input is equivalent to reordering the LUT contents. For example, inverting the least significant bit is the same as swapping adjacent entries in the LUT, and inverting the most significant bit can be achieved by swapping the two halves of the LUT. The result of the optimization is shown in Figure 4.

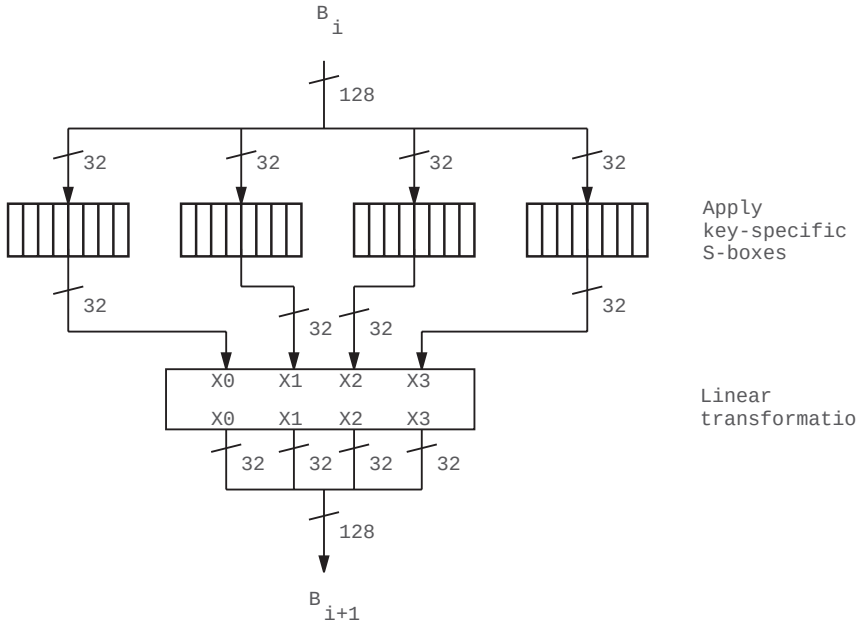


Fig. 4. Optimization of Serpent Rounds 0 to 30

The final round, shown in Figure 2, replaces the linear transformation with an additional XOR between the datapath and subkey K_{32}. These 128 XOR gates can also be folded into the round's S-boxes. Again let s be an output bit from the S-boxes, and k be the corresponding subkey bit form K_{32}. Each XOR results in either s or $\overline{s}$. An inversion on an S-box output is effected by inverting the contents of the LUT. As shown in Figure 5, the optimized final round contains only S-boxes.

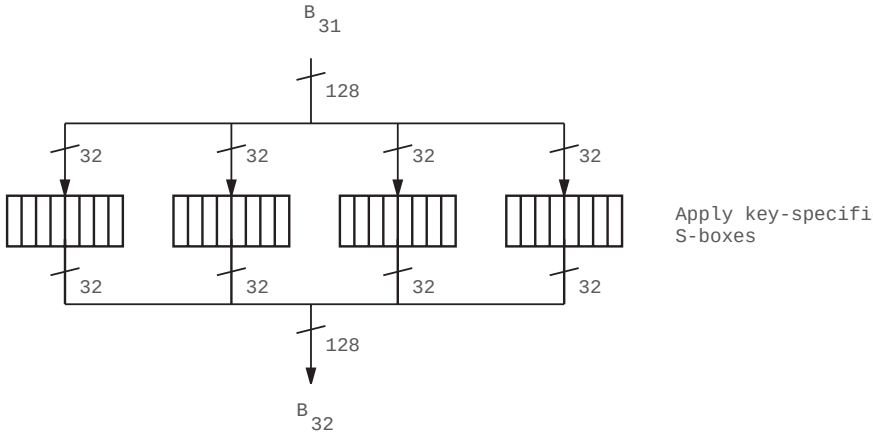


Fig. 5. Optimization of Serpent Round 31

Dynamic specialization has removed logic, register and routing resources for 4224 two-input XOR gates and 4224 subkey bits. This accounts for the two-fold reduction in circuit area compared with the static design. IOBs are no longer required for subkey loading. One of the four logic levels and the associated routing has also been removed from the critical path between pipeline registers, which is largely responsible for the speed improvement.

The static design uses an XCV1000BG560 Virtex part, although the authors indicate that it should fit in an XCV800. By contrast, the dynamic design targets the significantly less expensive XCV400BG432. Routing delays are not reduced by using a larger part, because the mapping and placement is fixed. Subkey precomputation is performed outside the FPGA in both designs, and could be performed by similar software environments.

4.2 High-Level JBits Code

The Serpent design is defined as a JBits run-time parameterized core (RTPCore). An RTPCore can generate a bitstream directly. It can also generate EDIF, for integration with logic simulators and the NGD/NCD-based Xilinx implementa-

tion tools. The EDIF flow was used for validation and timing analysis of the Serpent design.

An RTPCore supports hierarchy with ports and subcores. The subcore's ports are connected with nets and buses. Depending on whether a bitstream or EDIF is being generated, a primitive RTPCore either makes JBits calls to configure logic and routing resources, or creates a netlist of components from the Virtex SIMPRIM library.

The constructor for the Serpent module is shown in Figure 6. The arguments are the hexadecimal value of the encryption key, and the external nets and buses to be connected to the clock, B_0 and B_{32} ports. For simplicity, encryption is assumed. An initial key is required to support the static EDIF flow. Note that run-time modification of the encryption key does not require the core to be recreated, since the key only affects the contents of LUTs.

```
public class Serpent extends RTPCore {
  public Serpent(String hexKey, Net clk, Bus din, Bus dout) {
    byte[]   key    = Key.fromHexString(hexKey);
    int[]    preKey = Key.computePreKey(key);
    byte[][] subKey = Key.computeSubKey(preKey);

    Port clkPort  = newInputPort("clk", clk);
    Port dinPort  = newInputPort("din", din);
    Port doutPort = newOutputPort("dout", dout);
    Net clkInt = newNet("clk");
    Bus[] b = new Bus[NUM_ROUNDS+1];
    for (int i = 0;  i < b.length;  i++)
      b[i] = newBus("b" + i, DATAPATH_WIDTH);
    clkPort.setIntSig(clkInt);
    dinPort.setIntSig(b[0]);
    doutPort.setIntSig(b[NUM_ROUNDS]);

    SerpentRound[] round = new SerpentRound[NUM_ROUNDS];
    for (int i = 0;  i < LAST_ROUND;  i++)
      round[i] = new SerpentRound(i, subKey[i], clkInt, b[i], b[i+1]);
    round[LAST_ROUND] = new SerpentRound(subKey[LAST_ROUND],
                                         subKey[LAST_ROUND+1], clkInt,
                                         b[LAST_ROUND], b[LAST_ROUND+1]);
  }
}
```

Fig. 6. Serpent Constructor

The key is first converted to binary, then to the prekey array of 132 32-bit words, and finally to a subkey array of 33 128-bit words. Ports are defined and connected to the external and internal signals. An array is created to reference the 32 `SerpentRound` submodules, which are instantiated with

the computed subkey(s) and connected to the clock net and the data buses. The `configureSBox` method (shown in Figure 3) is called 32 times by each `SerpentRound` instance.

The ports, nets and buses are used by both JRoute [16] and the EDIF generator. JRoute is a routing API built upon JBits, and selects and configures the routing resources necessary to make connections between CLBs. The core need only specify the end points of the connections as logical ports, which are translated to physical CLB pins once the module is placed. JRoute provides a significant abstraction layer for physical routing, since the core need not specify the detailed sequence of routing resources needed to complete a route. This can still be done, however, if JRoute does not select the required path.

4.3 Layout

Rounds 0 to 30 require 32 CLBs for the S-boxes, and 32 CLBs for the linear transformation. The final round does not perform a linear transformation, and is implemented entirely with S-boxes. As shown in Figure 7, an 8×8 array of CLBs is used for the non-final rounds. All 32 rounds fit within the 40×60 CLB array of an XCV400 device. Every LUT is used in the 2016 CLBs that implement the 32 rounds.

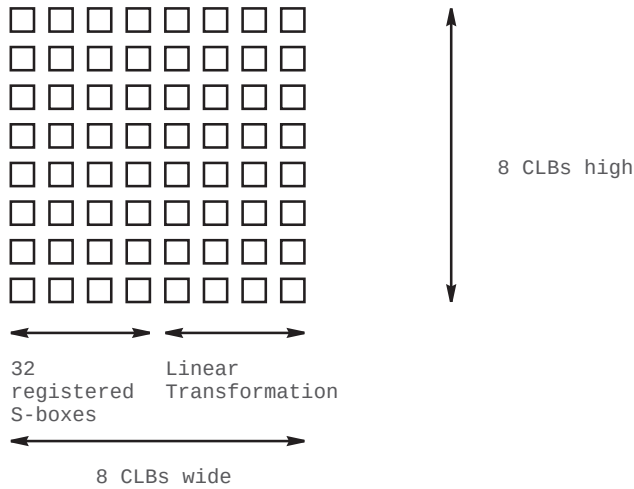


Fig. 7. Floorplan for Rounds 0 to 30

RTPCores have a mechanism similar to placement directives [17] for assigning coordinates to relocatable modules. This is used to create the floorplan in Figure 7, and to define the folded datapath for the 32 rounds. Beginning in the lower left corner, the rounds are placed horizontally in a zigzag pattern with

alternating left-to-right and right-to-left dataflows. This snake-like layout can be seen in Figure 8.

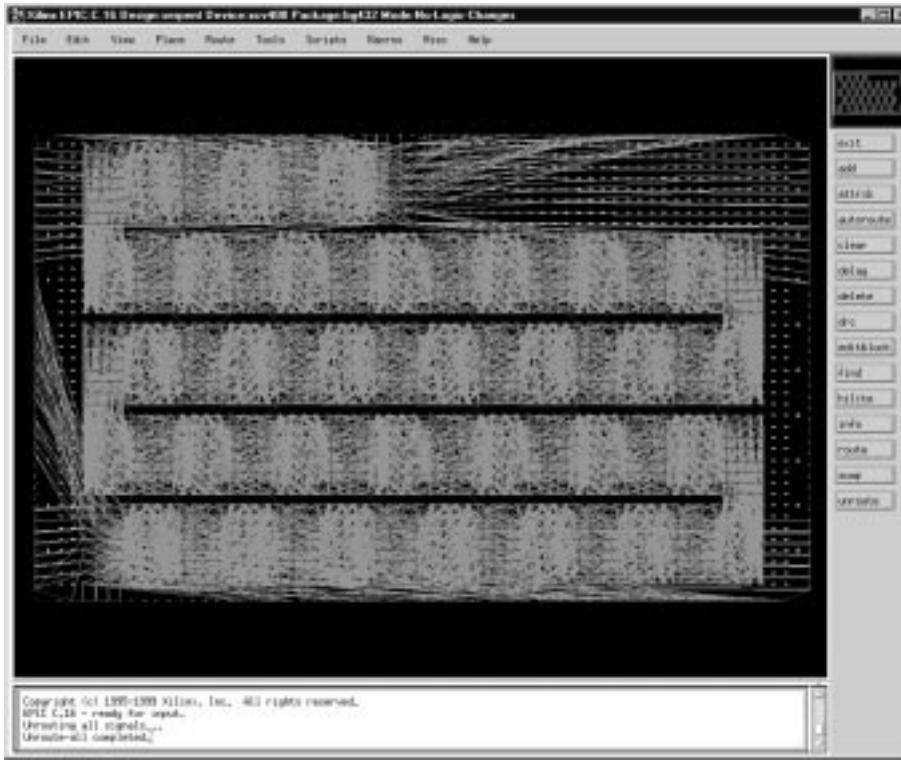


Fig. 8. EPIC Ratsnest View of the 32 Rounds

The design uses 8064 LUTs, 4352 flip flops, and 257 IOBs. All of the S-box CLBs and data IOBs are registered. Hence the total number of pipeline stages from input to output pins is 34. Slice utilization is 84% (4032 slices used from the 4800 available in an XCV400 device). A total of 384 CLBs and 60 IOBs are available for additional interface circuitry.

4.4 Validation and Performance

Functional verification was performed with the Model Technology VHDL simulator. An EDIF file with the S-box LUTs configured for a particular key is first generated. This is converted to an NGD file using `ngdbuild`. The `ngd2vhdl` program creates a structural VHDL netlist and testbench. Encryption and decryption results were identical to the output from a reference software implementation [2].

Table 1 reports the speed achieved for fully pipelined Serpent implementations. Both the static and dynamic FPGA designs use the same implementation technology (Virtex -4 speed grade) and tools (M2.1). The ASIC performance is estimated by the National Security Agency for a MOSIS 0.5 micron process [9]. High volumes are required before an ASIC can justify using a process technology similar to Virtex.

The bulk of the performance improvement is achieved with dynamic circuit specialization, although floorplanning also contributes. Regular floorplans make reconfiguration more efficient. Power consumption for the XCV400-4 dynamic design is estimated at 4 watts, and is not reported for the static design. Note that the highest speed grade XCV400 in a BG432 package may be less expensive than the lowest speed grade XCV1000 or XCV800 in a BG560 package.

Table 1. Performance Results

Technology Used	Package	Design Methodology	Floorplanned	Clock Rate (MHz)	Throughput (Gbits/sec)
Xilinx V1000-4	BG560	static	no	37.97	4.86
ASIC		static		62.73	8.03
Xilinx V400-4	BG432	dynamic	no	75.48	9.66
Xilinx V400-4	BG432	dynamic	yes	80.27	10.27
Xilinx V400-6	BG432	dynamic	yes	101.44	12.98
Xilinx V400E-8	BG432	dynamic	yes	137.15	17.55

In order to change the key or switch between encryption and decryption, 56 of the CLB columns have to be reconfigured (i.e. about 85% of the 2,546,080 configuration bits in an XCV400). Using the 8-bit wide Virtex SelectMap™ port running at 50 MHz, this reconfiguration is accomplished in under 10 milliseconds. Assuming the use of a high-end microprocessor, computing new subkeys and updating the LUTs with JBits calls also requires about 10 milliseconds. This is roughly the same as the time required for other system operations such as disk I/O. If there is no switching between encryption and decryption, an offset folding of the Serpent datapath can reduce the number of reconfigured CLB columns by 50%.

5 Conclusions

A dynamic implementation of the Serpent block cipher in a Virtex FPGA has been presented. It achieves a throughput of over 10 Gbits/sec, which is about 100 times faster than software implementations on high-end microprocessors. When compared with a static Virtex implementation, the dynamic circuit is over twice the speed and half the size. Power consumption and the number of package pins required is also reduced. This combination of factors result in significant cost savings.

Creating the key schedule is much more complicated for the AES finalists than for DES, and suits software implementation. The size of the key, and the time required to compute the subkeys, makes a high degree of key agility—such as changing the key on every clock cycle—difficult to achieve. Given this characteristic of the AES algorithms, FPGA reconfiguration overhead is less significant compared with DES.

As shown by code fragments in this paper, JBits provides several levels of abstraction for defining circuits. The lowest level JBits class provides complete configuration control, while the RTPCore class allows connectivity to be specified in terms of ports, nets and buses. Placement directives and JRoute help to bridge these levels. For systems that are partitioned between hardware and software, this single-language approach greatly reduces the integration effort.

Acknowledgements

Steve Guccione's development of JBits has made this work possible. Advice and assistance from the other members of the JBits project at Xilinx and Virginia Tech is greatly appreciated. This work was supported by the U.S. Defense Advanced Research Projects Agency, under contract DABT63-99-3-0004.

References

1. National Bureau of Standards. *FIPS PUB 46, The Data Encryption Standard*. U.S. Department of Commerce, Jan 1977.
2. Ross Anderson, Eli Biham, and Lars Knudsen. Serpent: A proposal for the Advanced Encryption Standard. `http://www.cl.cam.ac.uk/~rja14/serpent.html`.
3. Eli Biham. A fast new DES implementation in software. In *Fourth International Workshop on Fast Software Encryption (FSE'97)*, pages 260–271. Springer-Verlag Lecture Notes in Computer Science, Volume 1267, 1997.
4. National Institute of Standards and Technology. Announcing request for candidate algorithm nominations for the Advanced Encryption Standard (AES). *Federal Register*, 62(117):48051–48058, Sep 1997.
5. `http://www.nist.gov/aes`.
6. Stephen Charlwood and Philip James-Roxby. Evaluation of the XC6200-series architecture for cryptographic applications. In Reiner W. Hartenstein and Andres Keevallik, editors, *Eighth International Workshop on Field-Programmable Logic and Applications (FPL'98)*, pages 218–227. Springer-Verlag Lecture Notes in Computer Science, Volume 1482, Aug 1998.
7. Cameron Patterson. High performance DES encryption in Virtex FPGAs using JBits. In Kenneth L. Pocek and Jeffrey M. Arnold, editors, *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2000)*, Apr 2000.
8. Jason Leonard and William H. Mangione-Smith. A case study of partially evaluated hardware circuits: Key-specific DES. In Wayne Luk, Peter Y.K. Cheung, and Manfred Glesner, editors, *Seventh International Workshop on Field-Programmable Logic and Applications (FPL'97)*, pages 151–160. Springer-Verlag Lecture Notes in Computer Science, Volume 1304, Sep 1997.

9. Bryan Weeks, Mark Bean, Tom Rozylowicz, and Chris Ficke. Hardware performance simulations of round 2 Advanced Encryption Standard algorithms. In *Third AES Candidate Conference (AES3)*, Apr 2000.
10. D. Craig Wilcox, Lyndon G. Pierson, Perry J. Robertson, Edward L. Witzke, and Karl Gass. A DES ASIC suitable for network encryption at 10 Gbps and beyond. In Çetin K. Koç and Christof Paar, editors, *First International Workshop on Cryptographic Hardware and Embedded Systems (CHES'99)*, pages 37–48. Springer-Verlag Lecture Notes in Computer Science, Volume 1717, Aug 1999.
11. A.J. Elbirt and C Paar. An FPGA implementation and performance evaluation of the Serpent block cipher. In *ACM Eighth International Symposium on Field Programmable Gate Arrays (FPGA 2000)*, pages 33–40, Feb 2000.
12. Steve Guccione, Delon Levi, and Prasanna Sundararajan. JBits: Java based interface for reconfigurable computing. In *Second Annual Military and Aerospace Applications of Programmable Devices and Technologies (MAPLD'99)*, The Johns Hopkins University, Laurel, Maryland, Sep 1999.
13. Xilinx, Inc., 2100 Logic Drive, San Jose, California. *Virtex 2.5V FPGA Series Data Sheet*, Oct 1999.
14. Steve Kelem. *Virtex Configuration Architecture Advanced User's Guide*. Xilinx, Inc., 2100 Logic Drive, San Jose, California, Jun 1999. Application Note 151.
15. Bruce Schneier. *Applied Cryptography*. John Wiley & Sons, Inc., second edition, 1996.
16. Eric Keller. JRoute: a run-time routing API for FPGA hardware. In *Seventh Reconfigurable Architectures Workshop (RAW 2000)*, pages 874–881. Springer-Verlag Lecture Notes in Computer Science, Volume 1800, May 2000.
17. James Hwang, Cameron Patterson, S. Mohan, Eric Dellinger, Sujoy Mitra, and Ralph Wittig. Generating layouts for self-implementing modules. In Reiner W. Hartenstein and Andres Keevallik, editors, *Eighth International Workshop on Field-Programmable Logic and Applications (FPL'98)*, pages 525–529. Springer-Verlag Lecture Notes in Computer Science, Volume 1482, Aug 1998.

A 12 Gbps DES Encryptor/Decryptor Core in an FPGA

Steve Trimberger[1], Raymond Pang[1], and Amit Singh[2]

[1] Xilinx, Inc, 2100 Logic Dr, San Jose, CA 95124, USA
Steve.trimberger@xilinx.com
[2] University of California, Santa Barbara, CA, USA

Abstract. This paper describes two implementations of a Data Encryption Standard (DES) encryptor/decryptor that operate at data rates up to 12 Gbps. The 12 Gpbs number is faster than any previously published design. In these DES implementations, the key can be changed and the core switched from encryption to decryption mode on a cycle-by-cycle basis with no dead cycles. The designs were synthesized from Verilog HDL and implemented in Xilinx XCV300 and XCV300E devices. This paper describes the optimizations used and the coding conventions required to direct the synthesis tools to map the design to achieve a high-speed implementation. No physical constraints were given to the tools.

1 Introduction

The rapid growth of virtual private networks has heightened demand for encryption hardware that can handle high data rates. The hardware-friendly DES algorithm is well-suited to this application. Concerns about the vulnerability of DES are driving further standardization efforts, so any encryption hardware that is deployed today may become obsolete in a few months. In contrast, if the encryption engine resides in an FPGA, it could be updated in the field with a new encryption algorithm when that algorithm is available.

The DES algorithm has a regular structure that lends itself to pipelining, and simple data manipulations that permit fast operations. Several high-speed DES hardware implementations have been reported in the literature. These implementations unroll the 16 rounds of encryption and pipeline them. Wilcox [8] describes an ASIC implementation that operates above 10Gbps. Patterson [5] compiles a key-dependent data path for encryption in an FPGA that runs over 12Gbps [6], but the latency to change keys is tens of milliseconds. The most directly comparable prior art design implemented in an FPGA has complete loop unrolling and encrypts at 3.05Gbps [3].

This paper describes the implementation and optimization of an FPGA core for DES encryption and decryption. The core achieves a data rate of 8.4 Gbps with 16 cycles of latency, and 12 Gpbs with 48 cycles of latency. The core takes a key an encrypt/decrypt signal, both of which may change on a cycle-by-cycle basis.

Ç.K. Koç and C. Paar (Eds.): CHES 2000, LNCS 1965, pp. 156 - 163, 2000.

Since the core is compiled Verilog, it is simple to concatenate multiple copies of the core in a larger FPGA to provide triple DES [1] at the same data rate. It is also straightforward to interface the core to data concentrators and different communication interfaces supported by the FPGA, such as LVDS or double data rate (DDR) RAM.

2 The DES Algorithm

DES [2][7] takes as input one 56-bit key and one 64-bit block of data, and produces one 64-bit block of encrypted data. The same basic design is used for both encryption and decryption. As shown in figure 1, the DES algorithm begins with an initial permutation (IP), encrypts in sixteen "rounds", followed by the inverse of the initial permutation (IP^{-1}). In each round, the right-side 32 bits of the block are transformed with the function labeled "*f* " and the key, then exclusive-ored with the left side 32 bits. The key for each round is a subset of the original 56-bit key with bits permuted. After each round, the two sides of the data block are swapped and the algorithm continues.

The *f* function expands the right side to 48 bits, exclusive-ors those bits with the key, and divides the resulting 48 bits into eight 6-bit fields. Those fields are used as addresses into 8 64-word by 4-bit memories called S-boxes. The eight 4-bit S-box outputs are re-assembled into the 32-bit word that is XORed with the left side of the block.

Decryption differs from encryption in the way the bits of the sub-keys (K_1-K_{16}) are selected from the encryption key. This selection leads to the key bits multiplexer in figure 2.

In summary, the DES algorithm consists of 16 identical encryption rounds. Each round contains a significant amount of bit movement, which is simple wire in a hardware implementation, 80 2-bit XORs, and 8 lookups in 64-word by 4-bit S-boxes. Each round uses a subset of the key bits with a particular permutation. The permutation depends on the round and on whether the operation is encrypt or decrypt. Consisting primarily of wiring, table lookups and bitwise operations, the algorithm fits nicely into an FPGA.

3 Implementation

We coded the design in Verilog and simulated with Cadence Verilog-XL 2.2.31. We synthesized with Synopsys Design Analyzer 1998.08, targeting Xilinx Virtex-6 speed grade. Physical design was done with Xilinx Design Manager 2.1I, C.22.

The original Verilog design was intended to be space efficient, and was implemented as a single instantiation of the encryption round that operated iteratively. A single block of data passed through the round 16 times to produce one block of output. We modified this original version in several ways to gain significant throughput and clock rate improvements. The following sections describe those improvements.

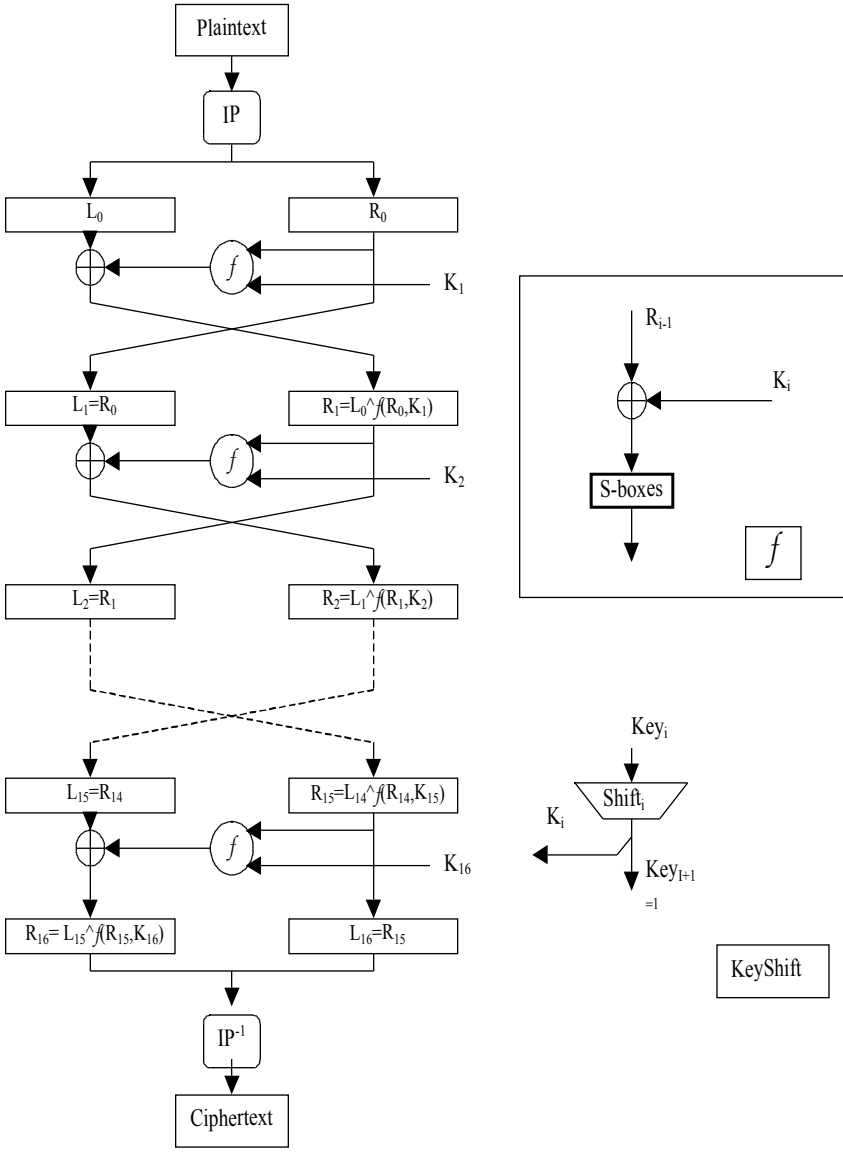


Fig. 1. DES Algorithm Overview

3.1 Loop Unrolling and Pipelining

To gain speed, we built 16 copies of the round and unrolled the loop, pipelining the data through the 16 stages. This increased the data rate by a factor of sixteen, but at the cost of approximately sixteen times as much logic. This design simulated fine, but logic synthesis, which we ran at medium effort, predicted a clock rate less than 25MHz. The critical path through the round is shown in figure 2. A multiplexer selects key bits depending on the round and on whether we are encrypting or decrypting. The resulting selected key bits are XORed with the right side of the data block (R_i) and 6-bit fields are used to address in to the S-boxes. One bit of one S-box is shown as 4LUTs, F5 and F6 MUXes. The resulting bits from the S-box are XORed with the left side bits from of the block (L_i) and stored in the pipeline register.

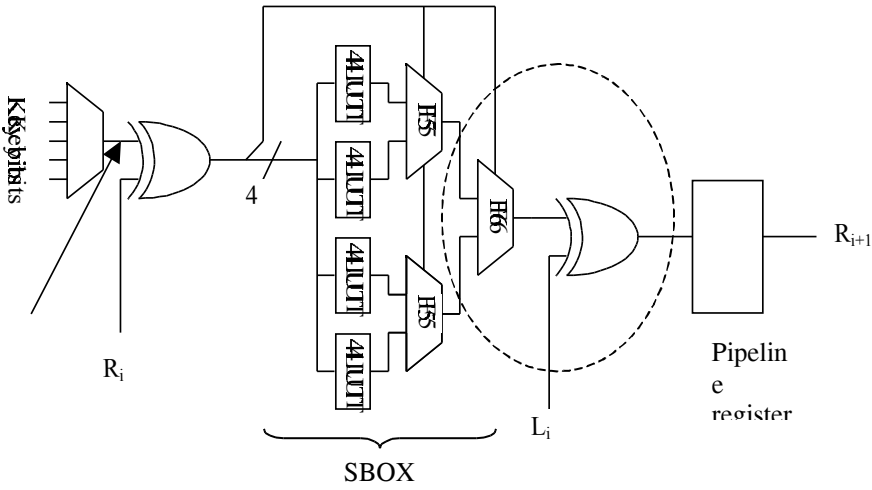


Fig. 2. Single Round Data Path.

3.2 Mapping to LUTs

Clearly, the critical path is through the logic of a single round. In our initial implementation, we used an SBOX expressed as 2-input functions[4] which produced an appallingly slow result, so we re-coded the SBOXes as 64x4 lookup tables. This design resulted in a critical path of eleven levels of LUTs which still performed rather poorly. We recognized that the Virtex CLB can implement one bit of an S-Box completely as a 64-word lookup table, which would reduce the logic to only three levels in the FPGA, but the synthesis tool did not implement the logic that way despite the Verilog description that looked like a lookup table. The stylized form of Verilog shown in figure 3 directs the synthesis tool to generate a 4LUT, but the form does not extend to 5LUTs or 6LUTs, which is why the original design was mashed into gates.

We changed the Verilog to build 4LUTs, and directly instantiated the Virtex F5 MUX in Verilog. We decided not to instantiate the F6 MUX because the F6 function could be merged with the following XOR gate in a single 4LUT (circled in figure 2), and the Virtex F5 function is slightly faster than the F6 function. The LUT following the SBOX is required in either case to implement the XOR with the left side of the data block that follows the SBOX lookup.

After these modifications, synthesis implemented the critical path in five levels of logic: two for the key MUX, one for the key XOR, one for the S-box through the 5LUT, and one for the F6 MUX plus the final XOR. Synopsys estimated the resulting speed at 50MHz.

```
always @(addr) begin
case (addr[2:5])
    0:  d2 = 4;
    1:  d2 = 1;
    2:  d2 = 14;
    3:  d2 = 8;
    4:  d2 = 13;
    5:  d2 = 6;
    6:  d2 = 2;
    7:  d2 = 11;
    8:  d2 = 15;
    9:  d2 = 12;
    10: d2 = 9;
    11: d2 = 7;
    12: d2 = 3;
    13: d2 = 10;
    14: d2 = 5;
    15: d2 = 0;
endcase
end
```

Fig 3. Verilog Case Statement that Generates a 4LUT.

3.3 Verilog Parameters

Focus turned now to the key multiplexer (on the left in figure 2). That multiplexer was rather wide because the Verilog code, which had been written for an iterative implementation, included a module for the key shift block. The module took as input signals that identified the round because the shift is different for different rounds. In the iterative version, those signals changed every cycle, but in the pipelined version,

those bits were tied to constants. However, since they were passed into a Verilog module, the multiplexers were not simplified by synthesis, so each of those MUXes had five or six inputs instead of two (one for encryption, one for decryption).

The solution was to implement these signals as Verilog parameters. To do this, the module must be declared a *template* in the synthesis tool, so the tool can create a separate module in the data base for each instance.

3.4 Decoupling the Key from the Data Path

In order to take the next step in performance improvement, we decided to take the key MUX off the critical path by pre-computing the key shift selection. This was rather straightforward. Since the key calculation must be pipelined along with the block of data, we moved the pipeline registers in the key calculation data path to the location marked with an arrow in figure 2. We added an additional pipeline stage in the first round at that location. The key must now be presented one cycle before the data it operates on. This modification is similar in concept to pre-computing the key schedule, which is common in software implementations, and which Patterson [5] did in software in his reconfigurable implementation. Since we compute the key in hardware anyway, we require no more logic to continually re-compute the key schedule one cycle ahead of when it is needed. This way, the decryptor is still able to switch keys on a cycle-by-cycle basis. After this modification, synthesis estimated a clock rate of about 70MHz. It was time for physical design.

3.5 Physical Design

We generated EDIF from the synthesized circuit, read the EDIF into the Xilinx Design Manager and set the target to XCV300-6. Without any constraints, placement and routing ran for about half an hour and produced a design that Design Manager reported used 4216 LUTs (about 69% capacity) and would run about 80MHz. We set a single timing constraint: the clock period should be 10ns, and set high placer and router effort (5). Placement and routing met the constraint after about four hours, yielding a circuit that encrypts or decrypts a 64-bit block every 10ns, a 6.4Gbps data rate. Further tightening of the clock period did not improve the resulting performance.

Next, we set the target to XCV300E-8 and tightened the clock rate constraint to 7.5ns. The resulting circuit ran at 132MHz, encrypting at 8.4Gbps.

We gave no placement constraints or hints. All performance numbers reported after physical design are post-layout, worst-case timing reports from the Xilinx Design Manager.

3.6 Deeper Pipeline

Finally, to wring a higher data rate, we inserted pipeline stages after the key XOR and after the F5 step in the S-Box lookup (and added two more pipeline stages in the key shift to maintain data alignment). This resulted in a 48-stage pipeline. Placement and routing with high effort, multiple-pass place-and-route and a tightening clock period

constraint yielded a designs that would operate at a data rate of 10.1Gbps in an XCV300-6 (6.3ns clock period), and 12Gbps in an XCV300E-8 device (5.3ns clock period).

4 Statistics

Both the 16-cycle and 48-cycle latency designs have these IO connections:

`Data_bus[1:64]`	Input data
`Data_out[1:64]`	Output data
`Key[1:56]`	Key data to be applied to the following data block
`Decrypt/encrypt`	The mode of operation on the current block
`E_data_rdy`	Input data is valid this cycle
`D_data_rdy`	Output data is valid this cycle
`Clk`	Clock

`Decrypt`, `E_data_rdy` and `D_data_rdy` are presented simultaneous with the data and are pipelined along with the data. The design can encrypt one cycle, and decrypt the next cycle with no dead cycles. `Key` must be presented one cycle before the data it operates on. Keys can also change every cycle with no dead cycles.

Here are implementation results for the two designs. Notice that the number of LUTs does not increase with increased pipeline depth. Pipeline registers in the data path require no additional logic, since the flip-flops in following the logic in the 16-stage pipeline were unused. The additional pipeline registers on the key bits required to maintain data alignment do add additional logic.

	16-stage pipeline	48-stage pipeline
Verilog Code Size (lines)	1106	1156
I/O	187	187
Design Size (LUTs)	4216	4216
Design Size (FFs)	1943	5573
Design Size estimated by Xilinx mapper (gates)	52936	72952
Data rate in XCV300-6 device (Gbps)	6.4	10.1
Data rate in XCV300E-8 device (Gbps)	8.4	12.0

5 Conclusion and Future Work

We designed a DES encryptor/decryptor core in Verilog and targeted it to an FPGA. The resulting design with 16 cycles of latency runs at 8.4Gbps. The design with 48

cycles of latency runs at 12Gbps. The speed of this design is approximately three times faster than the previous fastest comparable FPGA implementation. It is faster than an ASIC design reported only a year ago, and is comparable to a custom-key encryptor that requires tens of microseconds to change keys.

Part of the reason for executing this design was to determine the performance gained from various forms of optimizations to the design. We intend to use this information to drive design automation software development. In this design we were able to improve performance by more than a factor of two by applying an understanding of the algorithm to force a preferred mapping of the logic, and by changing the pipelining of the key. Although the former may be someday incorporated into logic synthesis software, many designers may not appreciate software that unilaterally changes the data alignment.

An observation of the delays in the final design shows that most of the delay in both implementations is due to interconnect routing. The next step in this investigation is to apply manual floorplanning and placement to reduce interconnect delay.

We are also interested in implementing additional encryption algorithms, with the intent that a system would load the algorithm of choice into the FPGA as needed. This strategy would permit, for example, fielding a system today that could be updated when the advanced encryption standard becomes available.

References

1. ANSI, "Triple Data Encryption Algorithm Modes of Operation", American National Standards Institute X9.52-1998, American Bankers Association, Washington DC, July 29, 1998
2. FIPS, "Data Encryption Standard", Federal Information Processing Standards Publication 46-2, 1993 December 30.
3. FreeIP, http://www.free-ip.com/DES/index.html
4. Kwan, M., "Bitslice DES", http://www.darkside.com.au/bitslice/ nonstd.c
5. Patterson, C., "High Performance DES Encryption in Virtex FPGAs using Jbits", *FCCM 2000*, IEEE Computer Society, 2000.
6. Patterson, C., private communication, 2000.
7. Schneier, B., *Applied Cryptography*, John Wiley and Sons, 1996.
8. Wilcox, D.C., et al., "A DES ASIC suitable for network encryption at 10Gbps and beyond", *First International Workshop on Cryptographic Hardware and Embedded Systems*, 1999.

A 155 Mbps Triple-DES Network Encryptor*

Herbert Leitold, Wolfgang Mayerwieser, Udo Payer, Karl Christian Posch, Reinhard Posch, and Johannes Wolkerstorfer

Institute for Applied Information Processing and Communications,
Graz University of Technology, Inffeldgasse 16a, A-8010 Graz, Austria
Johannes.Wolkerstorfer@iaik.at
http://www.iaik.at

Abstract. The presented Triple-DES encryptor is a single-chip solution to encrypt network communication. It is optimized for throughput and fast switching between virtual connections like found in ATM networks. A broad range of optimization techniques were applied to reach encryption rates above 155 Mbps even for Triple-DES encryption in outer CBC mode. A high-speed logic style and full-custom design methodology made first-time working silicon on a standard 0.6 μm CMOS process possible. Correct functionality of the prototype was verified up to a clock rate of 275 MHz.

Keywords. Network security, encryption, DES algorithm, Triple-DES, cipher block chaining, pipelining, true single-phase logic, full-custom design.

1 Introduction

Modern network technology offers transmission rates in the multi megabit range. In addition, Quality of Services (QoS) parameters like throughput and latency are guaranteed. These parameters make the transmission of voice and video in addition to normal LAN data possible. Whenever public accessible infrastructure is involved, mechanisms to secure confidential information are required. The Triple-DES algorithm in the Cipher Block Chaining (CBC) mode of operation meets these requirements [4], but demands considerable design effort to reach the desired throughput. Sustaining QoS-parameters even for short data packets requires an architecture where keys and encryption modes can be changed quickly. This task is far from trivial. Only dedicated hardware solutions can provide these properties by applying strong encryption.

This paper describes a single-chip context-agile encryption unit which is capable of encrypting (or decrypting) at rates of 155 Mbps using various DES-related algorithms. With respect to speed, the most demanding choice is Triple-DES in outer CBC mode. However, a network employing statistical multiplexing, such as the Asynchronous Transfer Mode (ATM), imposes another bottleneck: each

* The work described origins from the European Comission funded Project *Secure Communications in ATM Networks (SCAN)* established under contract AC0330 in the Advanced Communications Technologies and Services (ACTS) Program.

Ç.K. Koç and C. Paar (Eds.): CHES 2000, LNCS 1965, pp. 164–174, 2000.

user connection asks for its own encryption context consisting of keys and mode of operation. The time span between getting to know the identifier of a new user connection and the actual use of the related keys is too short. Moreover, two directions of data flow need to be served by a common key repository, and both directions may ask for accessing different keys at the same time.

The bottleneck of replacing keys rapidly arises due to the nature of ATM and is referred to as key-agile encryption [7]. ATM is a relaying technique operating on data units of a fixed size - called cells. ATM cells are relatively small units and consist of five byte header information and 48 byte payload. ATM is a connection oriented technology employing virtual connections (VCs) that are identified by a 24 bit value in the cell header. As ATM cells are statistically multiplexed between VCs, replacing the session key may be required for each ATM cell. Further requirements may even strive for assigning different encryption algorithms to each VC, such as DES and Triple-DES, which is referred to as algorithm-agile encryption [8]. Actually, the encryption unit presented in this paper allows to uniquely assign the encryption context including the operational mode to each connection which is called context-agile encryption [6].

The remainder of this paper sketches in section 2 general constraints arising from the application and their architectural impacts. Section 3 presents implementation details with emphasis on high-speed optimization techniques. Measuring results of the produced silicon and the prototype Network Interface Card (NIC) are presented in section 4. Finally, conclusions are drawn and future work is discussed.

2 Architecture

High-speed digital hardware can take advantage of exploiting parallelism. The more parts of a circuit work in parallel, the more data can be processed. For a network encryptor this is especially true with respect to the number of encryption modules [9]. Unfortunately, encryption in the CBC mode of operation requires the result of the previous encryption in order to process the current block. Thus, a parallelized architecture with more encyption modules would only speed up the electronic code book (ECB) mode and does not improve performance in general.

It might be assumed that other operational modes, such as the counter mode (CM) do not have the drawback of the CBC mode and can use multiple encryption modules in parallel. Nevertheless, CBC has excellent properties regarding synchronization. When ATM networks drop cells in periods of congestion, cryptographic resynchronization is required. The CBC mode re-establishes synchronization within two blocks when multiples of the block size get lost - as in the case of lost ATM cells. In contrary, the CM requires an explicit mechanisim to re-establish synchronization. This turns out to be major advantage of CBC, even if it forms tougher constraints on the crypto hardware.

In a network application, only two encryption modules can work independently as depicted in Figure 1. The first module encrypts the Down-Stream,

where data is sent to the network. The second one decrypts the Up-Stream, which receives data from the network.

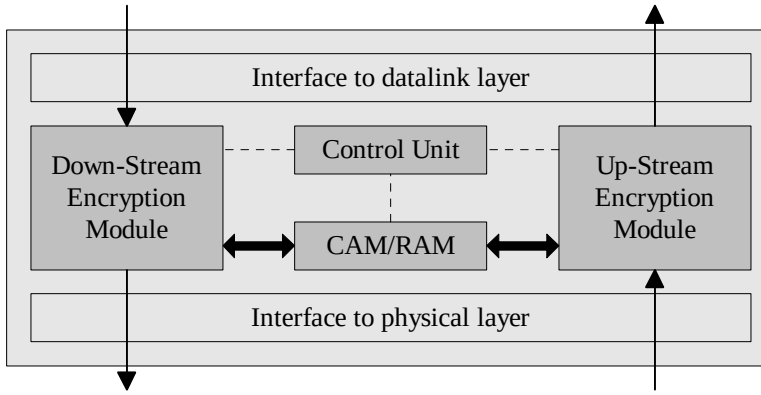


Fig. 1. Architecture of the network encryptor

2.1 Virtual Connections

When choosing an architecture with two encryption modules, connection parameters like session keys are loaded through the Down-Stream into the encryptor to avoid an extra interface. For every virtual connection, these data are stored in a CAM/RAM module for later retrieval. The CAM/RAM module is addressed with the 24-bit value identifying the virtual connection. Eight bits identify the virtual path, the remaining 16 bits identify the virtual channel. We do not distinguish between the virtual path identifier and the virtual channel identifier and denote all 24 bits as VCI. The VCI is part of the header information of a cell. It precedes the user data which is called payload. Encryption applies only to the payload. The header information is unaffected to preserve routing mechanisms. Each time a cell is processed, encryption parameters like type of algorithm, mode, session keys and initial vectors are retrieved from CAM/RAM. The worst case access rate for CAM/RAM is derived in Equation 1. The calculation is based on a 155 Mbps STM-1 signal used in ATM networks which offers a net bandwidth of 149.8 Mbps. It has a fixed cell size of 48 bytes payload and five byte header information.

$$f_{CAM} = \frac{bandwidth}{cellsize} = \frac{149.76\ Mbps}{(5+48)\,8\ Bit} = 353.2\ kHz = \frac{1}{2.83\ \mu s} \tag{1}$$

During these 2.83 μs the cell has to be identified by its VCI and the according connection parameters have to be copied from CAM/RAM into the encryption

module. In order to raise parallelism, this is done during encryption of the previous cell. Data is retrieved sequentially from CAM/RAM and stored in intermediate buffers for keys A and B and the initial vector (KEYA, KEYB, IV) as depicted in Figure 2. When encryption starts, data and encryption parameters are loaded concurrently into the DES module.

2.2 Encryption Module

The encryption module (Figure 2) is instantiated twice in the network encryptor: once for Down-Stream encryption and a second time for Up-Stream decryption. Each can perform both DES encryption and decryption, because the Triple-DES algorithm with two keys in the encryption-decryption-encryption (EDE) scheme demands both. The 16 rounds of a DES operation are executed sequentially. In our approach each round consumes two clock cycles which yields small logic functions that are convenient for a high-speed circuit. A version consuming one clock cycle per round would spend relatively more time for loading than for encryption, would have bigger logical functions and would demand a more complicated sub-key generator. Speed optimization would still be necessary and would not be simpler as for the two-cycle-variant. The overall-effort would be even higher. A pipelined version of the DES-round hardware makes no sense, because in the CBC mode no DES-blocks can be processed concurrently. Loading a 64-bit data block takes 10 clock cycles and occurs concurrently to unloading the previously processed block.

Encryption data is loaded from a First In First Out (FIFO) buffer which collects data bytewise from an asynchronous interface. The output FIFO buffers an encrypted block for asynchronous output. A complete DES encryption - loading included - takes 42 clock cycles, a Triple-DES encryption 108 cycles and plaintext loading 12 cycles. The CBC mode requires no additional clock cycles. Its XOR operation is done during loading for encryption and during unloading for decryption. For the sake of simplicity, a detailed descripton of the CBC dataflow is omitted, but it should be mentioned that the need for updating initial vectors (IV) in CAM/RAM and the need to treat the first block of a cell differently from subsequent ones raises the complexity of a hardware solution significantly.

2.3 Troughput

Triple-DES encryption with or without block chaining is the worst case scenario for throughput considerations. The required clock speed can be derived from encrypting a complete ATM cell in this mode as shown in Equation 2.

$$f_{clk} = f_{CAM} \times cycles = 353.2\ kHz \times (6 \times 108 + 12) = 233.1\ MHz \quad (2)$$

In practice, a clock speed close to 250 MHz is needed because retrieving data from CAM/RAM takes longer than encrypting the previous block. The resulting idle time of the DES module is compensated by a higher clock speed.

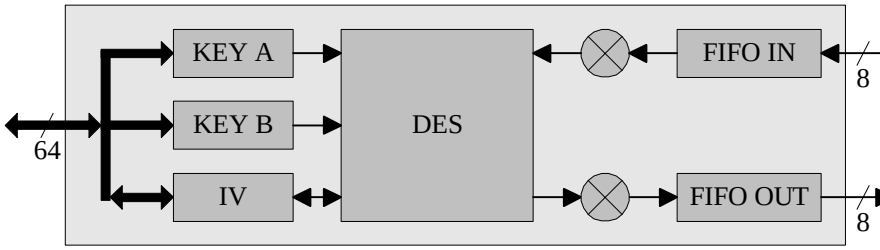


Fig. 2. Architecture of the encryption module

3 Circuit Implementation

The throughput calculation given above makes clear that a conventional chip design methodology like a standard-cell approach cannot cope with a clock speed of 250 MHz. This is especially true when a widely available CMOS process is to be used. We selected a standard 0.6 μm process from AMS International that offers a single polysilicon layer, two metal layers, and an option for a third metal layer. The nominal supply voltage is 5.0 Volts for the core and IO.

Those parts of the circuit which have to run at 250 MHz were designed using a full-custom design methodology. They exploit the true single-phase logic style that exhibits appropriate parameters for high speed applications. Due to the high design effort for a full-custom approach, we partitioned the circuit into a high-speed block (250 MHz) and a low-speed block. The latter operates with one quarter of the clock frequency (62.5 MHz). The low-speed block is assembled by auto-routed standard-cells synthesized from a VHDL description.

3.1 Standard-Cell Circuit

The standard-cell circuit handles the cell-level and block-level control of the network encryptor. It monitors input data of the Up-Stream and Down-Stream that are collected in the input FIFOs of the encryption modules. There it detects cell borders on basis of a start of cell (SOC) bit that is stored in addition to data. From the header information the cell type is identified. Roughly spoken, three types of cells are distinguished: key-download cells, signalling cells, and user cells. Key-download cells are used to update connection parameters of a virtual connection in CAM/RAM. They are only accepted in the Down-Stream to prevent malicious manipulation via the network. Signalling cells are passed to the output without any alteration to maintain the routing mechanism. When a user cell is identified, the connection parameters according to its VCI are retrieved from CAM/RAM. The cell's header information is passed unaltered to the output, and the payload is encrypted with the encryption algorithm, the encryption mode, the keys, and the initial vectors as stated in the CAM/RAM entry. If no entry was found in CAM/RAM, the cell will be output unaltered if

the static external signal "feedthough" is set to high. Otherwise the complete cell will be dropped. The cell-level control of Up-Stream and Down-Stream is completly independent, but their CAM/RAM access is shared. An arbitration unit schedules the access and grants higher priority to the Up-Stream to avoid data congestion and cell loss.

The standard-cell circuit also controls the block level. It starts the encryption module and cares for pipelined dataflow from the input FIFO to the DES module and from there to the output FIFO. Header blocks have to be treated differently because they have only five bytes instead of eight bytes. Another pecularity is the encryption of the first payload block in the CBC mode. It requires an initial vector from CAM/RAM, whereas subsequent blocks take the previous result as IV. After encrypting a cell's last block, the IV entry in CAM/RAM has to be updated. When no further data is available in the input FIFO, the block has to be moved into the output FIFO without pipelined loading of new data.

3.2 Full-Custom Encryption Module

The encryption module is a full-custom circuit for processing 64-bit data blocks. Besides plaintext operation, it supports two different algorithms: DES and two-key Triple-DES in EDE scheme. As modes of operation, ECB and CBC are supported for all algorithms without affecting throughput. Pipelined loading and unloading of the DES module is possible, because input FIFO and output FIFO act as buffers. Each FIFO is able to hold one and a half data blocks and has an asynchronous interface for off-chip communication. This interface conforms to the defacto standard UTOPIA [2], which connects the ATM layer with the physical layer in ATM components.

Datapath Optimization. The highest clock frequency at which a digital circuit has correct functionality depends on its crictical path. The crictical path is the part of circuit that needs most time for evaluating its logic function. In case that the evaluation time exceeds the cycle time of the clock, erroneous output will result. High-speed optimization basically tracks down critical paths repeatedly until the desired clock rate including a safety margin is reached.

In case of a hardware DES implementation, the critical path surely lies in the S-boxes which are used to substitute 6-bit values by 4-bit values. By granting a S-Box operation two clock cycles, the attainable clock frequency was nearly doubled. As an architectural consequence, every round of a DES encryption takes two cock cycles which in turn allows a simplified subkey generator design. The subkey generator has to rotate two 28-bit values up to two positions per DES round. Having to rotate only one position per clock cycle reduced the subkey generators functionality and improved its regular structure.

Control Logic. The control logic of the encryption module has to be clocked with the same frequency as the datapath. As in the datapath, a standard cell approach cannot be applied. Hence, a full-custom methodology was used to meet

the performance requirements. In contrary to the datapath, control logic lacks of regular structures. The design effort concentrates therefore on (hierarchical) decomposition. This strategy produces small subcircuits which can generate control signals adjacent to their controlled elements. Further, it is easier to generate layout for small subcircuits and to interconnect them to a complete circuit. Hierarchical decomposition also helps to cope with the functional complexity of controlling sequences. The control logic of the encryption module is split into two control machines. The first machine generates the control sequences for the 16 rounds of DES encryption and decryption. The second machine is able to perform Triple-DES encryption by starting the DES-controller three times. In addition, it controls the pipelined loading mechanism.

True Single-Phase Logic. As stated before, a semi-custom design methodology like a standard-cell approach does not reach the desired clock frequency of 250 MHz. Hence, a full-custom design methodolgy was applied which offers higher flexibility at the cost of additional design effort. Using this approach, a logic style was selected that has several benefits for speed optimized circuits: true single-phase logic (TSPL) [10].

TSPL is a dynamic logic style that combines combinational functionality with storage behaviour and thus offers low transistor count. It requires just a single clock signal which simplifies clock generation and clock distribution. Besides a complementary version of TSPL, like depicted in Figure 3, precharged N-latches can be built, where the P-logic block is replaced by a P-clock-transistor. Precharged N-latches just occur in ROM circuitry, which executes the S-box substitution of the DES algorithm. They speed up NOR structures significantly, but dissipate more power than a complementary version. This technique decreased the delay of the 256-bit S-box-ROMs to an acceptable level and defused the critical path. The logical function of complementary TSPL gates are kept simple to preserve their high speed. Especially, the number of P-transistors connected in series is kept low. Only 40 gate structures fulfilled these requirements, which made electrical and layout optimization a rewarding task due to their high reusability.

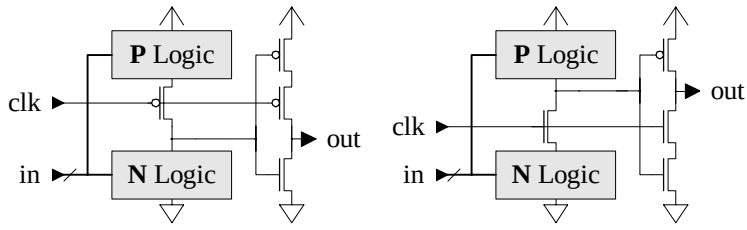


Fig. 3. Complementary true single-phase logic: P- and N-Latch

Clock Generation and Clock Distribution. When a clock signal for a synchronous high-speed circuit is distributed across a chip, delays caused by the distributed RC-effect of wires have to be considered. Especially long wires delay the signals by their parasitic sheet-resistance and sheet-capacitance that form a low-pass filter. The effective delay will vary across the chip area depending on the distance to the clock generation module (wire length) and the capacitive load. This delay variation is called clock skew. It can cause pipelining errors and data loss. Intersecting the clock net by inserting clock buffers solves this problem. A disadvantage of distributed clock drivers is the costly layout generation. For our circuit we chose a hybrid solution: two 'central' clocktrees for each encryption module shorten the average length of clock wires and their induced skew to an acceptable level.

The clock signal itself can either be fed via a pad into the circuit, when operating at clock frequencies up to 50 MHz, or it can be generated on-chip by a voltage controlled oscillator (VCO) module that is controlled by an analog pad. The VCO's frequency ranges from 25 MHz beyond 400 MHz. The clock signal is divided by four to clock standard-cell based modules. Their clock tree was automatically generated with place-and-route tools.

A problem similar to clock-distribution arises in the distribution of control signals. A control signal may have to drive many gates. Cascaded inverters are used to amplify the signal to an appropriate strength. In a high speed circuit, these inverters may cause a transport delay in the magnitude of a clock cycle. This delay can lead to erroneous behaviour. Our approach to avoid this situation was a limitation of the number of driven gates to 64. For higher gate counts, the control logic was doubled. Keeping the strength of the cascaded inverters moderate led to a reduction of the transport delay by slightly decreasing the steepness of the signal.

Layout. Layout and schematics of the encryption module were generated with Mentor GDT software. Regular structures - like the matrixces of the S-box-ROMs - were programmed by writing generators. Generators are used to instantiate interactively captured layout fractions and assemble them by wiring. The number of interactively captured layout cells was tried to be kept at a minimum which resultet in a library of highly reused leaf cells.

Special attention was paid to the floorplan. As mentioned above, the non-ideal behaviour of wires makes it necessary to keep routing distances as small as possible. The floorplan was optimized to avoid very long wires at cost of medium length wires. In the full-custom circuit, wires do not exceed the length of one millimeter. Wire length is also of interest when using TSPL cells. Their output signal should only be used near the cell, because it is not driven in all situations. In such a situation, the logic value is only held by the parasitc capacitance of the output node. If non-local interconnections are required, the insertion of static inverters overcomes this disadvantage.

Another floorplanning issue are the permutations of the DES algorithm. A straight forward implementation would require considerable routing area for per-

mutations. By exploiting regular structures of these permutations, routing area can be fairly reduced [5]. Figure 4 shows the complete layout of the chip. On the left side, two instances of the full-custom encryption module can be identified. Each has an area of 1.8 mm^2 and contains 32,000 transistors. The standard-cell circuit is located in the right half. The circuit's total die size is 23.7 mm^2 and counts nearly 120,000 transistors.

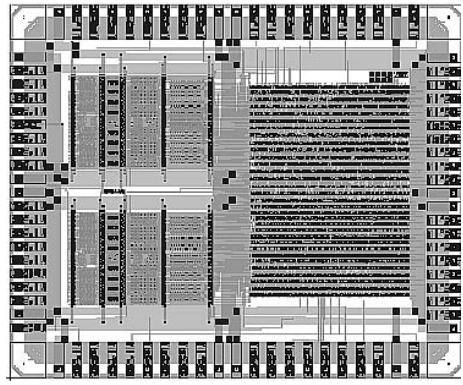


Fig. 4. Layout of the network encryptor

4 Measured Results

The functionality of the prototype chip samples (Figure 5) was verified on a chip tester. At a supply voltage of 5.0 Volts, correct functionality was verified up to a clock frequency of 275 MHz. Some samples even reached 290 MHz. These measurements match exactly circuit level simulations with extracted parasitics. A safety margin of more than 25 MHz ensures error-free behaviour at the target frequency of 250 MHz. When both encryption modules are held busy at this frequency, the circuit consumes 230 mA. At a supply voltage of 3.3 Volts, correct functionality is given up to 160 MHz.

The network encryptor chip was also verified in a real application. A conventional ATM network interface card was modified, that the encryptor chip could intercept communication between ATM layer and physical layer at the UTOPIA interface. The card has a PCI interface and software drivers for Windows 2000. It passed several basic tests.

5 Conclusion

Expertise in various domains like networking, security, hardware design, and high-speed optimization was necessary in order to implement a full functioning

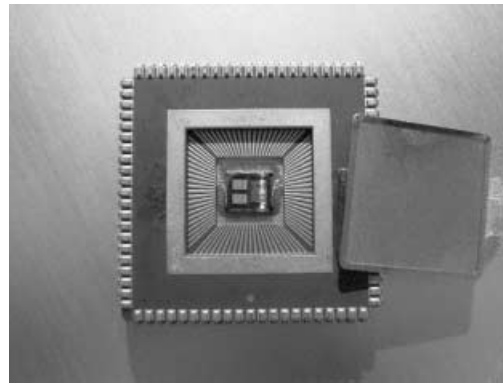


Fig. 5. Chip sample of the network encryptor

single chip for network encryption. The encryptor can concurrently encrypt and decrypt two 155 Mbps data streams with the Triple-DES algorithm in outer CBC mode. It operates at a clock frequency of 250 MHz. This combination of throughput and cryptographic strength was unattainable up to now. In addition, a sophisticated architecture allows encryption of several multiplexed virtual connections with different encryption algorithms, modes of operation, and keys without deteriorating Quality of Servce parameters.

Future work will include an expansion of the on-chip CAM/RAM memory size in order to support more virtual connections. This will improve the applicability in network interface cards. For use in network switches, an interface to external CAM/RAM will raise the number of virtual connections to a level that will satisfy even the needs of large networks. The prototype's interface for off-chip communication obeys the UTOPIA standard. Future versions will additionally support a microcontroller interface. This will open the scope of this chip for a broad range of applications, where high-speed encryption is asked for.

References

1. ATM Forum, ATM Security Specification Version 1.0, ATM-SEC-01.0100, The ATM Forum, Security Working Group, 1999.
2. ATM Forum, Utopia Level 2 - Version 1, af-phy-039.000, The ATM Forum, Technical Comittee, 1995.
3. ANSI, American Standard for Data Encryption Algorithm (DEA), ANSI 3.92, American National Standards Institute, 1983.
4. ANSI, American Standard for Information Systems Data Encryption Algorithm - Modes of Operation, ANSI 3.106, American National Standards Institute, 1983.
5. Hoornaert, F., Goubert, J. and Desmedt, Y., Efficient Hardware Implementation of the DES, Advances in Cryptology: Proceedings of CRYPTO '84, p. 147, Springer-Verlag, Berlin, 1985.

6. Lyndon G. Pierson, Edward L. Witzke, Mark O. Bean, Gerry J. Trombley, Context-Agile Encryption for High Speed Communication Networks, ACM SIGCOMM Computer Communication Review, vol. 29, no. 1., p. 35, 1999.
7. Daniel Stevenson, Nathan Hillery, Greg Byrd, Fengmin Gong, Dan Winkelstein, Design of a Key Agile Cryptographic System for OC12c Rate ATM, Internet Society Symposium on Network and Distributed Systems Security, San Diego, CA, 1995.
8. Thomas D. Tarman, Robert L. Hutchinson, Lyndon G. Pierson, Peter E. Sholander, Edward L. Witzke, Algorithm-Agile Encryption in ATM Networks, IEEE Computer, vol. 31, no. 9., p. 57, 1998.
9. D. Craig Wilcox, Lyndon G. Pierson, Perry J. Robertson, Edward L. Witzke, Karl Gass, A DES ASIC Suitable for Network Encryption at 10 Gbps and Beyond, Cryptographic Hardware and Embedded Systems: Proceedings of CHES '99, p. 37, Springer-Verlag, Berlin, 1999.
10. Jiren Yuan, Christer Svensson, High Speed Circuit Technique, IEEE J. Solid-State Circuits, vol. 24, p. 62, 1989.

An Energy Efficient Reconfigurable Public-Key Cryptography Processor Architecture*

James Goodman[1] and Anantha Chandrakasan[2]

[1] Chrysalis-ITS, Ottawa ON K2C-3R7, Canada
jgoodman@chrysalis-its.com
[2] Massachusetts Institute of Technology, Cambridge MA 02139, USA
anantha@mtl.mit.edu

Abstract. The ever increasing demand for security in portable, energy-constrained environments that lack a coherent security architecture has resulted in the need to provide energy efficient hardware that is algorithm agile. We demonstrate the feasibility of utilizing domain-specific reconfigurable processing for asymmetric cryptographic applications in order to satisfy these constraints. An architecture is proposed that is capable of implementing a full suite of finite field arithmetic over the integers modulo-N, binary Galois Fields, and non-supersingular elliptic curves over $GF(2^n)$, with operands ranging in size from 8 to 1024 bits. The performance and energy efficiency of the architecture are estimated via simulation and compared to existing solutions (e.g., software and FPGA's), yielding approximately two orders of magnitude reduction in energy consumption at comparable levels of performance and flexibility.

1 Introduction

In the past, several standards for implementing various asymmetric techniques have been proposed such as the ISO, ANSI (X9.*), and PKCS standards. The variety of standards[1] has resulted in a multitude of incompatible systems that are based upon different underlying mathematical problems and algorithms. For example, the IEEE P1363 public key cryptography standard [1] recognizes three distinct families of problems upon which to implement asymmetric techniques: integer factorization, discrete logarithms, and elliptic curves.

As a result, system developers have had to either utilize software-based techniques in order to achieve the algorithm agility required to maintain compatibility, or have utilized special purpose hardware and restricted themselves to only providing secure communications with compatible systems. Unfortunately,

* The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), the Air Force Research Laboratory, or the U.S. Government.

[1] A wise man once said that the best thing about standards is that there are so many to choose from!

Ç.K. Koç and C. Paar (Eds.): CHES 2000, LNCS 1965, pp. 175–190, 2000.

software-based approaches lead to slow implementations that are very energy inefficient. In the past these inefficiencies could be ignored as the typical user operated from a fixed-location system such as a desktop computer which had a great deal of memory, processing power, and an effectively limitless energy budget. However, with the migration to portable, battery-operated nomadic computing terminals these assumptions break down, requiring us to re-evaluate the use of a software-based implementation. Hardware-based implementations on the other hand, while being very energy and computationally efficient, are very inflexible and capable of supporting only a single type of asymmetric cryptography. A compromise between these two extremes is achieved by taking advantage of the fact that the range of operations is small enough that domain specific reconfigurable hardware can be developed that is capable of implementing the various asymmetric algorithms without incurring the overhead associated with generic reconfigurable logic devices. Furthermore, this is done in an energy-efficient manner that enables operation in the portable, energy-constrained environments where this algorithm agility is required most of all. The resulting implementation is known as the Domain Specific Reconfigurable Cryptographic Processor (DSRCP).

2 Domain Specific Reconfigurability

In conventional reconfigurable applications such as Field Programmable Gate Arrays (FPGAs), the architectural goals of the device are to provide a large number of small, yet powerful programmable logic cells, embedded within a flexible programmable interconnect (e.g., [2], [3]). Unfortunately, the overhead associated with making such a general purpose computing device ultimately limits its energy efficiency, and hence its utility in energy-constrained environments. To illustrate this fact consider the space of all possible functions. A conventional reconfigurable logic device attempts to cover as much of this space as possible given its architectural constraints in terms of technology and logic/routing resources. This results in a considerable amount of overhead that isn't necessary given a specific subset of functions. Kusse [4] quantified this overhead by breaking down the energy consumption of a conventional FPGA (Xilinx XC4003A [5]) into its architectural components (Table 1). The analysis revealed that only 1/20th of the total energy is used to perform useful computation.

Table 1. Energy consumption breakdown of XILINX XC4003A [4].

Component	% Total Energy Consumption
Interconnect	65%
Global Clock	21%
I/O	9%
Logic	**5%**

The DSRCP differs from conventional reconfigurable implementations in that its reconfigurability is limited to the subset of functions, called a domain, required for asymmetric cryptography. This domain requires only a small set of configurations for performing all of the required operations over all possible problem families as defined by IEEE P1363. As a result, the reconfiguration overhead is much smaller in terms of performance, energy efficiency, and reconfiguration time, making the DSRCP feasible for algorithm-agile asymmetric cryptography in energy constrained environments.

3 Instruction Set Architecture (ISA)

The instruction set definition of the DSRCP is dictated by the IEEE P1363 description document. For the various primitives described in the standard, a list of the required arithmetic functions is tabulated in order to determine the required ISA of the processor. Note that certain primitives also require such operations as the ability to set specific bits within a given operand and the ability to generate random bits, neither of which are implemented in this version of the processor. The resulting functional matrix is shown in Table 2.

The functional matrix is used to define the required ISA of the processor, along with additional auxiliary functions for controlling the processor configuration, as well as moving data into, out of, and within the processor. The resulting instruction format for the DSRCP is a 30-bit word partitioned as shown in Figure 1. The DSRCP executes 24 instructions in all, a brief summary of which are given in Table 3.

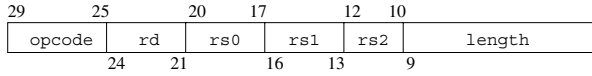


Fig. 1. DSRCP instruction word.

4 Architecture

Figure 2 shows the overall system architecture of the DSRCP. The processor consists of four main architectural blocks: the global controller and microcode ROMs, the I/O interface, the reconfigurable datapath, and an embedded SHA-1 [6] hash function engine. The inclusion of a hash engine was desirable as the key derivation primitives contained within IEEE P1363 call for this functionality.

4.1 Global Controller and Microcode ROMs

The DSRCP features a three-tiered control approach that utilizes both hardwired and microcode ROM-based control functions. This multi-tiered approach

Table 2. Functional matrix of the IEEE P1363 for the DSRCP instruction set.

	ADD	SUB	MULT	MOD	MOD_ADD	MOD_SUB	MOD_MULT	MOD_INV	MOD_EXP	GF_ADD	GF_MULT	GF_SQUARE	GF_INV	GF_EXP	EC_ADD	EC_DOUBLE	EC_MULT
PKO #1									X								
PKO #2			X		X	X	X		X								
IFEP-RSA									X								
IFDP-RSA			X		X	X	X		X								
IFSP-RSA1			X		X	X	X		X								
IFVP-RSA1									X								
DLSVDP-DH														X			
DLSVDP-MQV	X				X		X				X			X			
DLSP-DSA				X	X		X	X									
DLVP-DSA				X			X	X						X			
ECSVDP-DH																	X
ECSVDP-MQV					X		X								X		X
ECSP-DSA				X	X		X	X									
ECVP-DSA				X			X	X							X		X
MOD_EXP					X		X										
GF_EXP											X	X					
EC_ADD										X	X	X	X				
EC_DOUBLE										X	X		X				
EC_MULT															X	X	

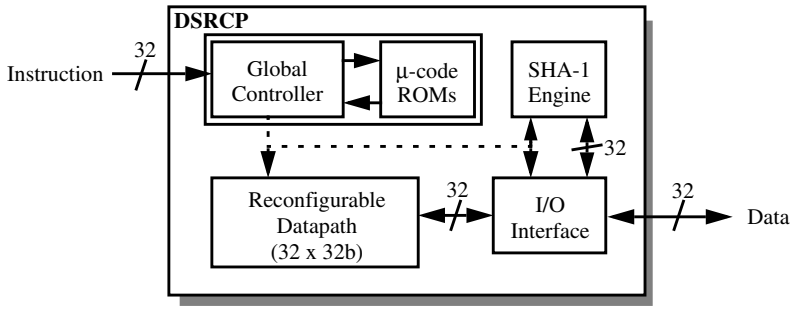


Fig. 2. Overall system architecture of the DSRCP.

Table 3. DSRCP instruction set.

Mnemonic	Description
SET_LENGTH length	sets width of processor to be length + 1
REG_CLEAR rd,rs0	clears registers specified in mask formed by (rd,rs0) = $R_{7:0}$
REG_MOVE rd,rs0	rd = rs0
REG_LOAD rd	rd is loaded from I/O interface
REG_UNLOAD rs1	rs0 is unloaded to I/O interface
COMP rs0,rs1	sets gt = (rs0 > rs1) and eq = (rs0 == rs1) flags
ADD/SUB rd,rs0,rs1,rs2	$rs2_{2:1}$ = 00: rd = rs0 + rs1 + $rs2_0$ $rs2_{2:1}$ = 01: rd = (rs0 + rs1 + $rs2_0$)/2 $rs2_{2:1}$ = 10: rd = rs0 - rs1 $rs2_{2:1}$ = 11: rd = (rs0 - rs1)/2
MOD_ADD rd,rs0,rs1,rs2	rd = (rs0 + rs1 + $rs2_0$) mod N
MOD_SUB rd,rs0,rs1	rd = (rs0 - rs1) mod N
MONTRED_A	$(Pc, Ps) = A \cdot 2^{-n} \bmod N$
MONTMULT	$(Pc, Ps) = A \cdot B \cdot 2^{-n} \bmod N$
MONTRED	$(Pc, Ps) = (Pc, Ps) \cdot 2^{-n} \bmod N$
MOD rd,rs0,rs1,rs2	rd = (rs1·2^n + rs0) mod N, 2^{2n} mod N initially stored in rs2
MOD_MULT rd,rs0,rs1,rs2	rd = (rs0·rs1) mod N, 2^{2n} mod N initially stored in rs2
MOD_INV rd,rs0	rd = (1/rs0) mod N
MOD_EXP rd,rs0,rs2,length	rd = $rs0^{Exp}$ mod N, Exp has length+1 bits, 2^{2n} mod N stored in rs2
GF_ADD rd,rs0,rs1	rd = rs0⊕rs1
GF_MULT	$Pc = A \cdot B$
GF_INV	$A = 1/Pc$
GF_INVMULT	$A = B/Pc$
GF_EXP rd,rs0,length	rd = $rs0^{Exp}$ mod N Exp has length+1 bits, 2^{2n} mod N stored in rs2
EC_ADD rd,rs0,rs1,rs2,wb	(rd,rd+1) = (rs0,rs0+1) + (rs1,rs1+1), curve defined by (R_6, N) NOTE: if (wb = 0) addition is performed but result is discarded
EC_DOUBLE rd,rs0,rs2	(rd,rd+1) = 2·(rs0,rs0+1), curve defined by (R_6, N)
EC_MULT length	$(R_4, R_5) = Exp \cdot (R_2, R_3)$, Exp has length+1 bits, curve defined by (R_6, N)

is required as various instructions within the DSRCP's ISA are implemented using other instructions within the ISA, as illustrated for the MOD_MULT instruction in Figure 3. The first tier of control corresponds to those instructions that are implemented directly in hardware. The second tier of control represents the first level of microcode encoded instructions, which are composed of sequences of first tier instructions. Similarly, the third tier of control represents the second level of microcode encoded instructions which consist of sequences of both first and second tier instructions.

The microcode approach is chosen due to its simplicity and extensibility as modifications and enhancements of the ISA can be accomplished with a minimal amount of design effort by modifying the microcode ROM contents. The drawback of using this approach is the additional latency that is incurred by accessing the ROMs, which can end up consuming a significant portion of the processor's cycle time. This performance issue is addressed by pipelining the instruction decoding/sequencing at the output of the first-level microcode ROM.

The global controller is also responsible for disabling unused portions of the circuitry in order to eliminate any unnecessary switched capacitance. The shutdown strategy is dictated by the current width of the processor and enables the datapath to be shutdown in 32 32-bit increments. SET_LENGTH(length), $7 \leq$ length ≤ 1023, is used to set the current width of the processor. All operands

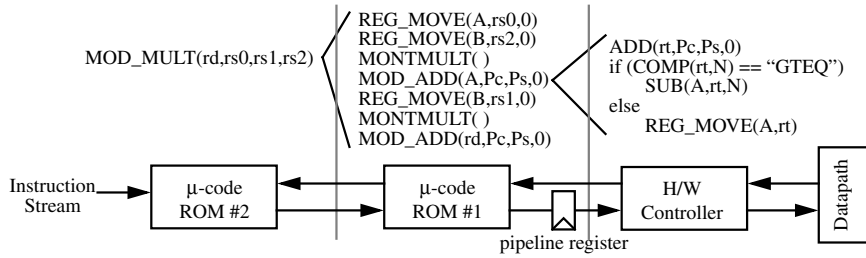


Fig. 3. Hierarchical instruction structure of the DSRCP.

accessed and operated upon by the datapath are assumed to be the size of the current width of the processor, as set by the last invocation of `SET_LENGTH`. This length is then used by the control logic to determine the number of iterations that need to be performed by the various operations.

4.2 I/O Interface

Operands used within the processor can vary in size from 8-1024 bits, requiring the use of a flexible I/O interface that allows the user to transfer data to/from the processor in a very efficient manner. A 32-bit interface is used which is very well suited to existing processors and systems which are predominantly built upon 32-bit interfaces. The choice enables fast operand transfer onto and off of the processor, requiring at most 32 cycles to transfer the largest possible operand.

4.3 Reconfigurable Datapath

The primary component of the DSRCP is the reconfigurable datapath, whose architecture is shown in Figure 4. The datapath consists of four major functional blocks: an eight word register file, a fast adder unit, a comparator unit, and the main reconfigurable computation unit. The datapath is implemented using a very area-efficient bitsliced implementation in order to minimize its size, and the corresponding wiring capacitance of its control signal generation/distribution.

The register file size is chosen to be eight words as it is the minimum number required to implement all of the functions of the datapath. The limiting case for this architecture is that of elliptic curve point multiplication in which registers (R_2,R_3) are used to store the point that is going to be multiplied by the value stored in Exp register, (R_4,R_5) are used to store the result, (R_0,R_1) are used to store an intermediate point used during the computation, R_6 is used to store the curve parameter a, and R_7 is used as a dummy register in order to provide resilience to timing attacks.

The number of read and write ports within the register file is dictated by the requirement to be able to perform single cycle, two operand instructions which generate a writeback value. In certain cases two write ports could have proved

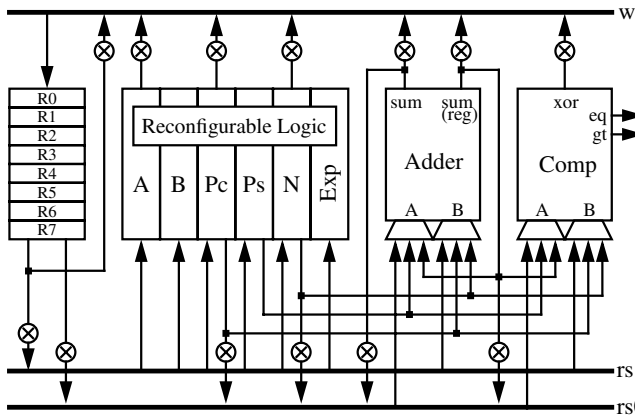


Fig. 4. Reconfigurable datapath architecture block diagram.

useful (e.g., elliptic curve point transfers), but the infrequency of the operation didn't merit the additional overhead that it would have introduced. The register file provides access to the LSB's of R_0, R_1, R_2, and R_3, as required by the modular inversion operation.

The fast adder unit is capable of adding/subtracting two n-bit ($8 \leq n \leq 1024$) operands in four cycles using the hybrid carry-bypass and carry-select technique described in [7] (Figure 5), and optimized for a bitsliced implementation. The unit features a local register to store the previous sum result, a feature that is used in modular addition/subtraction and inversion routines. The adder unit can also right shift its result, as required by the modular inversion algorithm used within the DSRCP.

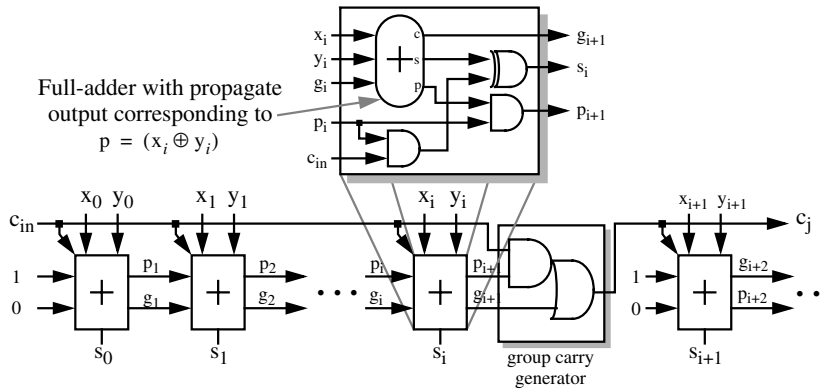


Fig. 5. Modified bitsliced carry-bypass/select adder [7].

The comparator unit performs single-cycle magnitude comparisons between two n-bit operands, as well as computing the XOR of the two operands (i.e., $GF(2^n)$ addition). The comparator generates two flags, `gt` and `eq`, that can be decoded into all possible magnitude relations using a fast $\mathcal{O}(\log_2 n)$ depth tree-based topology that enables single-cycle magnitude comparisons.

The reconfigurable computation unit consists of six local registers (Pc, Ps, A, B, Exp, and N) and a reconfigurable logic block that is capable of implementing all of the required datapath operations. Using local memory within the datapath eliminates the need to continually access the register file every cycle, eliminating the associated overhead of repeated register file accesses. The Pc and Ps registers are used primarily in modular operations to store the carry-save format partial product, and in Galois Field operations as two separate temporary values. A and B store the input operands used in all modular and Galois Field operations. The Exp register is used for storing either the exponent value, in the case of exponentiation operations, or the multiplier value, in the case of elliptic curve point multiplication. The N register also serves a dual purpose; for modular operations it's used as the modulus value, and in Galois Field operations it stores the field polynomial in a binary vector form (e.g., $x^3 + x^2 + 1$ is stored as [1,1,0,1]). In all relevant operations, it's assumed that both the Exp and N registers are pre-loaded with their required values.

5 Reconfigurable Logic Cell Design

The DSRCP is capable of performing a variety of algorithms using both conventional and modular integer fields, as well as binary Galois Fields. These operations are implemented using a single computation unit that can be reconfigured on the fly to perform the required operation. The possible configurations are Montgomery multiplication/reduction, $GF(2^n)$ multiplication, and $GF(2^n)$ inversion. All other operations are either handled by other units such as the fast-adder and comparator, or implemented in microcode.

5.1 Montgomery Multiplication

Montgomery multiplication [8] utilizes the simple iterated radix-2 implementation:

$$(Pc, Ps)_{j+1} = \frac{(Pc, Ps)_j + b_j A + q_j N}{2}, j = 0, \ldots, n-1 \qquad (1)$$

where $q_j = Ps_0 \oplus b_j A_0$, and b_j is the jth bit of operand B. A redundant carry-save representation of the partial product accumulator (Pc,Ps) is exploited in order to minimize the cycle time. This operation can be implemented using the basic computational resources of Figure 6(a): two full-adders and two `AND` gates. Montgomery reduction of A can be performed by setting $B = 1$ (i.e., $b_0 = 1, b_i = 0, i = 1, ..., n-1$). Similarly, reduction of (Pc,Ps) can be performed by setting $B = 0$.

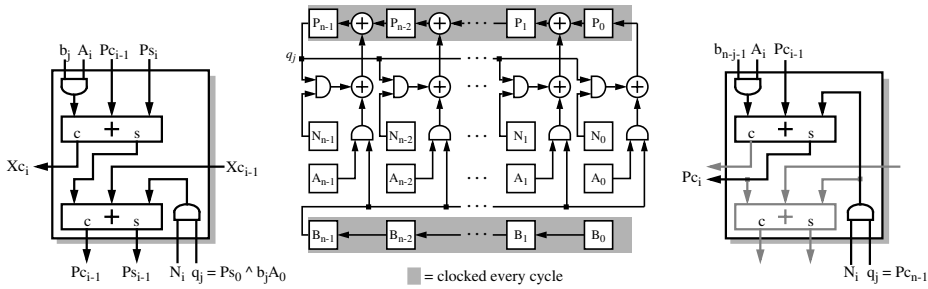


Fig. 6. Multiplication architectures and datapath configurations.

5.2 $GF(2^n)$ Multiplication

Mastrovito's thesis [9] serves as an extensive reference of hardware architectures for performing $GF(2^n)$ multiplication. Given our choice of a polynomial basis, the most efficient multiplier architecture is an MSB-first approach (Figure 6(b) [10]) as it minimizes the number of registers that are clocked in any given cycle. In addition, the MSB-first approach can be mapped to the existing hardware of the Montgomery multiplier (Figure 6(c)) by exploiting the fact that a full-adder's sum output computes a 3-input $GF(2)$ addition. Hence, $GF(2^n)$ multiplication can be performed using the iteration:

$$Pc_j = 2Pc_{j-1} \oplus b_{n-j-1}A \oplus q_jN, j = 0, \ldots, n-1 \tag{2}$$

where q_j is bit $n-1$ of Pc_{j-1}, which is used to modularly reduce the partial product Pc_j. The field polynomial, $f(x)$, is stored as a binary vector in N, and the resulting approach is universal in the sense that it can operate with any valid field polynomial over $GF(2^n)$ for $8 \leq n \leq 1024$.

5.3 $GF(2^n)$ Inversion

The limiting operation in affine co-ordinate Elliptic Curve point operations is typically the inversion operation. In hardware using a polynomial basis, the Extended Binary Euclidean Algorithm [11] can be used to compute inverses in a very efficient manner. This algorithm can also be modified to perform a multiplication in concurrency with the inversion by initializing the Y variable to be the multiplier value (if no multiplication is required, the Y register can simply be initialized with the value 1). This optimization provides significant savings during elliptic curve point operations as it eliminates one multiplication, reducing the total cycle count by approximately 18%. The resulting algorithm (Algorithm 1) can be further optimized by parallelizing the two embedded `while` loops, which effectively halves the number of cycles required as the dominant portion of time is spent in this part of the algorithm. The net result of these

optimizations is a universal invert-and-multiply operation that takes at most four multiplication times ($T_{mult} = n$ cycles), and on average $3.3 \cdot T_{mult}$ in order to invert (and multiply) an element of $GF(2^n)$.

Input: W: a, the element of $GF(2^n)$ that is to be inverted
X: N, the binary representation of the field polynomial $f(x)$
Y: b, the element of $GF(2^n)$ that is to be multiplied by the computed inverse
Z: 0, just plain old zero!

Output: $Z = b/a$

Algorithm:

```
while (W != 0)
   while (W0 == 0)
      W = W/2
      Y = (Y + Y0·N)/2
   endwhile
   while (X0 == 0)
      X = X/2
      Z = (Z + Z0·N)/2
   endwhile
   if (W >= X)
      W = W + X
      Y = Y + Z
   else
      X = W + X
      Z = Y + Z
   endif
endwhile
```

Algorithm 1. Extended Binary Euclidean Algorithm over $GF(2^n)$ used in DSRCP.

Inversion can be implemented with the datapath cell used in both Montgomery and $GF(2^n)$ multiplication by providing a small degree of reconfigurability such that computational resources can be re-used to perform different parts of Algorithm 1. The basic requirements are two 2-input adders over $GF(2)$ to perform each of the parallel `while` loops, and the two summations in each branch of the if clause. Each iteration of the parallel `while` loops requires one cycle for performing the actual operations as all operations are performed in parallel. An additional cycle is incurred when the exit condition of the parallel `while` loops is satisfied (i.e., $W_0 = X_0 = 1$) as it must be detected via an additional iteration of the loop. The second part of the algorithm requires a single cycle as well. The two datapath adders can be used as two-input $GF(2)$ adders by zeroing one of their inputs, and then utilizing multiplexors to allow the adder inputs to be changed on the fly to accommodate Algorithm 1. The corresponding architecture and its resulting mapping to the datapath cell is shown in Figure 7.

The final datapath cell is shown in Figure 8. In all it contains two full-adders, two `AND` gates, 6 two input multiplexors, and 6 register cells. The reconfiguration muxes are controlled through the use of 8 control lines (three for the adder muxes and five for the register muxes).

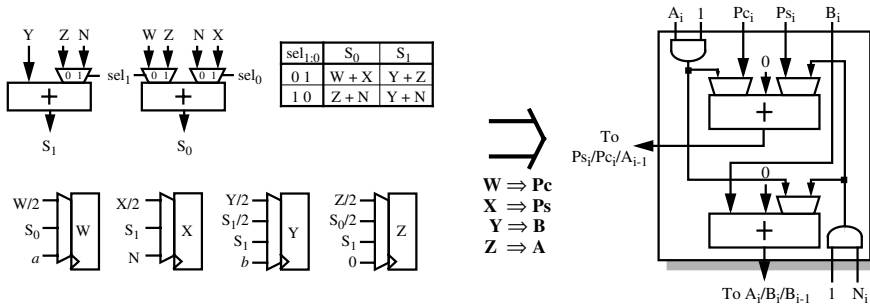


Fig. 7. Basic $GF(2^n)$ inversion architecture and resulting datapath cell.

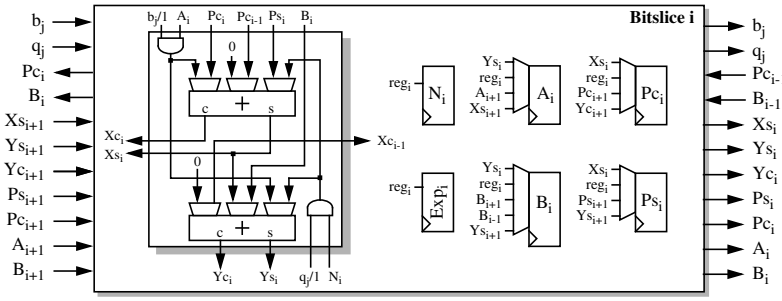


Fig. 8. Final reconfigurable datapath cell.

6 Algorithm Implementation

The DSRCP performs a variety of algorithms ranging from modular integer arithmetic to Elliptic Curve arithmetic over $GF(2^n)$. All operations are universal in that they can be performed using any valid n-bit modulus or $GF(2^n)$ field polynomial, with $8 \leq n \leq 1024$.

6.1 Modular Arithmetic

The various complex modular arithmetic operations (multiplication, reduction, inversion, and exponentiation) are implemented using microcode. Multiplication is performed using Montgomery multiplication, which requires a correction factor of $2^{2n} \bmod N$ be provided with the modulus N in order to undo the division by 2^n inherent in Montgomery's method.

Modular reduction is performed using Montgomery reduction and exploits the fact that the value being reduced can be decomposed into two n-bit values and then reduced via Algorithm 2.

Modular inverses are computed using the Extended Binary Euclidean Algorithm. This technique requires special architectural considerations such as the

Input: rs0, rs1: n-bit registers containing the value to be reduced stored in the format $(\text{rs1}\cdot 2^n + \text{rs0})$
rs2: n-bit register containing the Montgomery correction value $2^{2n} \bmod N$

Output: rd = $(\text{rs1}\cdot 2^n + \text{rs0}) \bmod N$

```
Algorithm: REG_MOVE(Ps,rs0), CLEAR_PC  // Ps = rs0, Pc = 0
           MONTRED( )                  // (Pc,Ps) = rs0·2^-n mod N
           MOD_ADD(rd, Pc,Ps,0)        // rd = rs0·2^-n mod N
           MOD_ADD(rd,rd,rs1,0)        // rd = (rs1·2^n + rs0)2^-n mod N
           REG_MOVE(A,rd)              // A = rd
           REG_MOVE(B,rs2)             // B = 2^2n mod N
           MONTMULT( )                 // (Pc,Ps) = (rs1·2^n + rs0) mod N
           MOD_ADD(rd,Pc,Ps,0)         // rd = (rs1·2^n + rs0) mod N
```

Algorithm 2. Modular reduction implementation on the DSRCP.

ability to right shift the output of the adder unit, and explicit access to the LSB of R_0, R_1, R_2, and R_3 in order to check the looping conditions of the Euclidean algorithm.

Modular exponentiation is performed using a standard square-and-multiply algorithm [12] with an exponent scanning window of size two. The algorithm (Algorithm 3) pre-computes and stores the values $\{2^n, rs0 \cdot 2^n, rs0^2 \cdot 2^n, rs0^3 \cdot 2^n\}$ in $\{R_0, R_1, R_2, R_3\}$ respectively. During each iteration the current value is squared twice, and then the exponent is scanned two bits at a time (this scanning is done non-destructively so exponent values don't need to be reloaded prior to each operation). The value scanned corresponds to the register that is used during the subsequent multiplication (e.g., if `01` is read then R_1 is used). Note that multiplying by the value stored in R_0 is essentially a NOP as Montgomery multiplication is being used which implicitly divides the product by 2^n.

The use of NOPs provides protection from timing attacks and simple power analysis as a multiplication is always performed, thereby eliminating any variation in execution based on the exponent's value. The expense of this immunity is that strings of zeros in the exponent cannot be exploited to speed up the operation. The loss in efficiency due to this fixed performance, assuming that the exponent is uniformly distributed, is only 9%.

The use of the length operand in the `MOD_EXP` instruction enables the length of the exponent and the operands to be decoupled, leading to much more efficient exponentiation when the exponent value is significantly shorter than the operands.

6.2 $GF(2^n)$ Arithmetic

$GF(2^n)$ addition is performed using the XOR function of the comparator unit, and both $GF(2^n)$ multiplication and inversion are implemented directly in hardware using the reconfigurable datapath. $GF(2^n)$ exponentiation is implemented in the same manner as modular exponentiation, with $\{1, rs0, rs0^2, rs0^3\}$ being pre-computed and stored in $\{R_0, R_1, R_2, R_3\}$. NOPs are once again exploited to provide immunity to timing attacks and simple power analysis.

Input: rs0: n-bit register containing value to be exponentiated
rs2: n-bit register containing Montgomery correction value $2^{2n} \bmod N$
length: 10-bit value representing length of the exponent stored in Exp

Output: rd = rs0$^{Exp} \bmod N$

Algorithm:

```
REG_MOVE(Ps,rs2)                    // Ps = 2^2n mod N
MONTRED( )                          // (Pc,Ps) = 2^n mod N
MOD_ADD(R0,Pc,Ps,0)                 // R0 = 2^n mod N
REG_MOVE(A,rs0)                     // A = rs0
REG_MOVE(B,rs2)                     // B = 2^2n mod N
MONTMULT( )                         // (Pc,Ps) = rs0·2^n mod N
MOD_ADD(R1,Pc,Ps,0)                 // R1 = rs0·2^n mod N
REG_MOVE(A/B,R1)                    // A,B = rs0·2^n mod N
MONTMULT( )                         // (Pc,Ps) = rs0^2·2^n mod N
MOD_ADD(R2,Pc,Ps,0)                 // R2 = rs0^2·2^n mod N
REG_MOVE(B,R2)                      // B = rs0^2·2^n mod N
MONTMULT( )                         // (Pc,Ps) = rs0^3·2^n mod N
MOD_ADD(R3,Pc,Ps,0)                 // R3 = rs0^3·2^n mod N
REG_MOVE(A/B,R0)                    // A,B = 2^n mod N
for (i = length-1; i >= 0;i = i-2)
   MONTMULT( )                      // (Pc,Ps) = P^2·2^n mod N
   MOD_ADD(A/B,Pc,Ps,0)             // A,B = P^2·2^n mod N
   MONTMULT( )                      // (Pc,Ps) = P^4·2^n mod N
   MOD_ADD(A,Pc,Ps,0)               // A = P^4·2^n mod N
   REG_MOVE(B,R<Exp2i:2i−1>)        // B = R<Exp2i:2i−1> = Rj
   MONTMULT( )                      // (Pc,Ps) = P^4+j·2^n mod N
   MOD_ADD(A/B,Pc,Ps,0)             // A,B = P^4+j·2^n mod N
endfor
MONTRED_A( )                      // (Pc,Ps) = P^Exp mod N
MOD_ADD(rd,Pc,Ps,0)                 // rd = P^Exp mod N
```

Algorithm 3. Modular exponentiation on the DSRCP.

6.3 Elliptic Curve

The DSRCP performs affine coordinate elliptic curve operations on non-supersingular curves over $GF(2^n)$ of the form:

$$E : y^2 + xy = x^3 + ax^2 + b \tag{3}$$

where $a, b \in GF(2^n)$. The corresponding point addition and doubling formulae, assuming that P_1 and P_2 are distinct points on E, are given by:

$$\begin{aligned} P_1 + P_2 = (x_3, y_3), x_3 &= \lambda^2 + \lambda + x_1 + x_2 + a \\ y_3 &= (x_2 + x_3)\lambda + x_3 + y_2 \\ \lambda &= \frac{y_1 + y_2}{x_1 + x_2} \end{aligned} \tag{4}$$

$$\begin{aligned} 2P_1 = (x_3, y_3), x_3 &= \lambda^2 + \lambda + a \\ y_3 &= (x_1 + x_3)\lambda + x_3 + y_1 \\ \lambda &= x_1 + \frac{y_1}{x_1} \end{aligned} \tag{5}$$

Note that the ISA of the DSRCP enables it to also perform elliptic curve operations over fields of prime characteristic using an external sequencer and the appropriate formulae (e.g., [13]).

Point addition and doubling are implemented in microcode using the above formulae, with curve points stored as register pairs $(R_i, R_{i+1}) = (x, y)$. Point addition features an additional input in the form of a writeback enable bit which must be set for the result to be written back to the destination register pair. If the enable bit is not set, then the computation is performed and the result is discarded, leaving the destination register pair unaffected. This feature is used to provide immunity to timing attacks and simple power analysis during elliptic curve point multiplication.

Point multiplication is performed using a repeated double-and-add algorithm, with a window size of one. Larger window sizes are not possible on the current DSRCP architecture due to memory limitations of the register file (e.g., four pre-computed values would require 8 additional registers). The issue of timing attacks is once again addressed by utilizing NOPs via the writeback enable bit of the point addition operation. The overhead associated with using NOPs is 33% relative to a conventional implementation where NOPs are skipped, and 50% if a signed radix-2 representation is used for the multiplier [12].

7 Performance Estimates

Cycle counts for the proposed architecture are determined via simulation using Verilog and the Synopsys Timemill^{TM} simulator [14]. The results are shown in Figure 9 for the various operations in terms of the cycles per bit of the operand (e.g., 1024-bit modular multiplication takes approximately 2048 cycles). The execution times for the modular/$GF(2^n)$ exponentiation and elliptic curve multiplication operations are derived using a nominal operating frequency of 50 MHz, at a supply voltage of 1 V. The power consumption of the datapath has also been simulated using Synopsys' Powermill^{TM} simulator [15], and is estimated to be at most 15 mW using the aforementioned operating conditions.

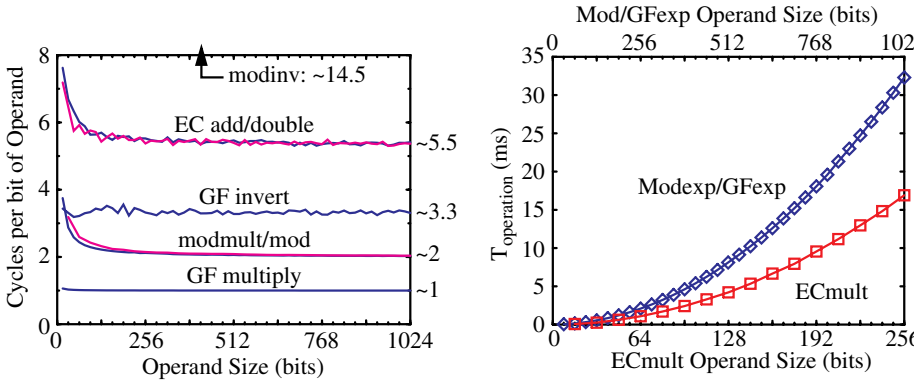


Fig. 9. Simulated performance of the DSRCP.

The SHA-1 hash function engine's performance under these conditions is $266 Mbps$ at a peak power consumption of 800 μW, or 3 pJ/bit. In comparison, an optimized assembly language software-based solution executing on Intel's StrongARM^{TM} SA-1100 [16] has a rate of 33.9 $Mbps$ at a power consumption of 352.5 mW, for 10.4 nJ/bit. Hence, the hardware-based solution described here is over three orders of magnitude more energy efficient.

8 Results and Conclusions

The resulting estimated energy consumption of the DSRCP for a variety of operations and operand sizes is presented in Figure 10. Energy estimates for conventional FPGA and software based solutions are also depicted for comparison. The energy estimates for the FPGA's come from estimates based on the work described in [18] and [17]. The software-based energy consumptions were measured using a StrongARM^{TM} SA-1100 evaluation platform that was executing hand-optimized assembly language implementations of the various operations.

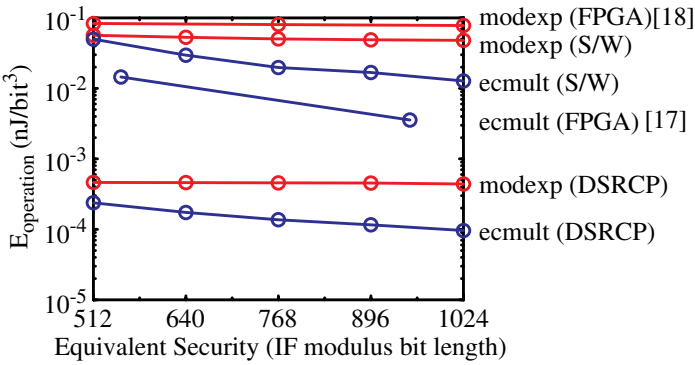


Fig. 10. Estimated energy efficiency of the DSRCP vs. conventional solutions (FPGA & S/W).

The comparison illustrates that the DSRCP is estimated to be on the order of $30-180\times$ more energy efficient than generic FPGA-based solutions, and over two orders of magnitude more energy efficient than conventional software-based solutions. In addition, the proposed architecture enables the user the same flexibility as both the software and FPGA-based solutions in terms of implementing asymmetric cryptographic algorithms.

Acknowledgements

Effort sponsored by National Semiconductor, as well as the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air

Force Materiel Command, USAF, under agreement number F30602-00-2-0551. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

References

1. IEEE P1363, "Standard specifications for public-key cryptography - Draft 13," IEEE, November 12, 1999.
2. Xilinx Corporation, *Virtex-E Field Programmable Gate Arrays (XCV00) Databook*, 1999.
3. Altera Corporation, *APEX 20K Programmable Logic Device Family Data Sheet*, 2000.
4. E. Kusse and J. Rabaey, "Low-energy embedded FPGA structures," *ISLPED'98 - Proceedings of the 1998 International Symposium on Low Power Electronic Design*, 1998, 155–160.
5. Xilinx Corporation, *XC4000 Field Programmable Gate Arrays: Programmable Logic Databook*, 1996.
6. FIPS 180-1, "Secure hash standard," Federal Information Processing Standards Publication 180-1, U.S. Department of Commerce/N.I.S.T., National Technical Information Service, Springfield, Virginia, April 17, 1995.
7. A. Satoh, *et. al.*, "A high-speed small RSA encryption LSI with low power dissipation," *ISW'97 - Proceedings of First International Information Security Workshop*, 1998, 174–187.
8. P. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, 48 (1987), 243–264
9. E.D. Mastrovito, *VLSI Architectures for Computation in Galois Fields*, Ph.D. Thesis, Linköping University, Linköping, Sweden, 1991.
10. P.A. Scott, S.E. Tavares, and L.E. Peppard, "A fast VLSI multiplier for $GF(2^m)$," *IEEE Journal on Selected Areas of Communications*, vol. SAC-4, no.1, January 1986, 62–66.
11. D.E. Knuth, *The Art of Computer Programming - Volume 2: Seminumerical Algorithms*, Addison-Wesley, Reading MA, 2nd Edition, 1981.
12. D.M. Gordon, "A survey of fast exponentiation methods," *Journal of Algorithms*, vol. 27, no. 1, April 1998, 129–146.
13. G. Seroussi, N.P. Smart, and I.F. Blake, *Elliptic Curve Cryptography*, Cambridge University Press, February 2000.
14. Synopys Corporation, *TimeMill User's Manual*, 1999.
15. Synopsys Corporation, *PowerMill User's Manual*, 1999.
16. Intel Corporation, *StrongARM SA-1100 Microprocessor for Portable Applications Brief Datasheet*, September 1999.
17. M. Rosner, *Elliptic Curve Cryptosystems on Reconfigurable Hardware*, Master's Thesis, Worcester Polytechnic Institute, Worcester MA, 1998.
18. T. Blum, *Modular Exponentiation on Reconfigurable Hardware*, Master's Thesis, Worcester Polytechnic Institute, Worcester MA, 1999.

High-Speed RSA Hardware Based on Barret's Modular Reduction Method*

Johann Großschädl

Graz University of Technology
Institute for Applied Information Processing and Communications
Inffeldgasse 16a, A–8010 Graz, Austria
Johann.Groszschaedl@iaik.at

Abstract. The performance of public-key cryptosystems like the RSA encryption scheme or the Diffie-Hellman key agreement scheme is primarily determined by an efficient implementation of the modular arithmetic. This paper presents the basic concepts and design considerations of the RSAγ crypto chip, a high-speed hardware accelerator for long integer modular exponentiation. The major design goal with the RSAγ was the maximization of performance on several levels, including the implemented hardware algorithms, the multiplier architecture, and the VLSI circuit technique.
RSAγ uses a hardware-optimized variant of Barret's modular reduction method to avoid the division in the modular multiplication. From an architectural viewpoint, a high degree of parallelism in the multiplier core is the most significant characteristic of the RSAγ crypto chip. The actual prototype contains a 1056$*$16 bit partial parallel multiplier which executes a 1024-bit modular multiplication in 227 clock cycles. Due to massive pipelining in the long integer unit, the RSAγ crypto chip reaches a decryption rate of 560 kbit/s for a 1024-bit exponent. The decryption rate increases to 2 Mbit/s if the Chinese Remainder Theorem is exploited.
Keywords: Public-key cryptography, RSA algorithm, modular arithmetic, partial parallel multiplier, pipelining, full-custom VLSI design.

1 Introduction

Security is an important aspect in many applications of modern information technology, including electronic commerce, virtual private networks, secure internet access, and digital signatures. All these services apply public-key cryptography. Practical and secure examples for public-key algorithms are the Rivest, Shamir and Adleman (RSA) encryption scheme [RSA78], the Diffie-Hellman (DH) key agreement scheme [DH76], and the Digital Signature Algorithm (DSA) for generation/verification of digital signatures [Nat94]. From a mathematical viewpoint,

* The work described in this paper was funded by the Austrian Science Foundation (FWF) under grant number P12596–INF "Hochgeschwindigkeits-Langzahlen-Multiplizierer-Chip".

Ç.K. Koç and C. Paar (Eds.): CHES 2000, LNCS 1965, pp. 191–203, 2000.

the algorithms mentioned have a common characteristic: they perform long integer modular exponentiation.

In order to meet modern security demands, the modulus should have a length of at least 1024 bits. The calculation of a 1024-bit RSA decryption in software causes a very high computational cost since the complexity of the modular exponentiation is n^3 for n-bit numbers. Special hardware accelerators like the RSAγ crypto chip contain an optimized long integer multiplier and for this reason they are more efficient for RSA decryption than a general-purpose 32-bit CPU.

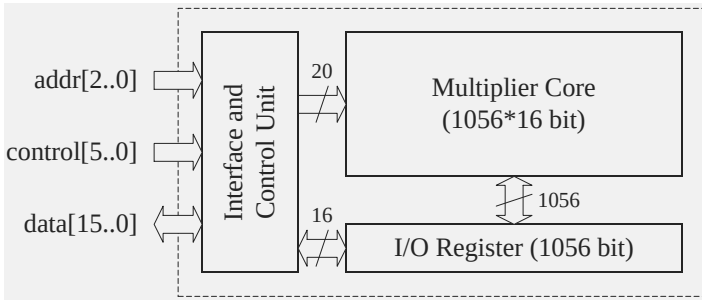


Fig. 1. Main components of the RSAγ crypto chip.

Figure 1 shows the most important components of the RSAγ crypto chip on the system-level. For communication with the off-chip world, the RSAγ provides a 16-bit standard microcontroller *interface*. Data interchange and command invocation are performed via this interface. The *control unit* is responsible for feeding the multiplier core with control signals. Since the multiplier is clocked with 200 MHz, and the control signals have to be provided with the same frequency, they can not be delivered from outside the chip via the pads. Therefore, the control sequences for the square and the multiply operation are stored in a FIFO within the control unit. Other parts of the control unit like the register for the exponent and the non-speed critical portions of the control logic operate at 25 MHz. The frequency ratio between the core and the interface is 8:1.

The *I/O register* supports 16-bit serial data transfer with the interface unit, and 1056-bit parallel data exchange with the multiplier core. Note that the data transfer from and to the multiplier core does not affect the throughput since the fetching of the exponentiation result and the loading of the next data block takes place independently of the multiplier core. The performance of the RSAγ crypto chip mainly relies on the efficiency of the modular arithmetic, therefore this paper focusses on the *multiplier core*. We present the basic algorithmic and architectural concepts of the multiplier, and describe how they were combined and optimized for each other in order to reach maximum performance.

Since the publication of the RSA public-key cryptosystem in 1978, many algorithms for modular multiplication have been proposed; the most important are

summarized in [NM96]. Nevertheless, it turns out that two algorithms are preferred for hardware implementation: the Barret modular reduction method [Bar87] and the Montgomery algorithm [Mon85]. In recent years, most proposed approaches are based on Montgomery's algorithm, either in conjunction with a redundant number representation or in an systolic array architecture. Blum et al. reported an implementation of Montgomery modular exponentiation on FPGAs [BP99]. They reached a decryption time of approximately 10 ms for a 1024-bit modulus when the Chinese Remainder Theorem is applied. Compared to the architectures based on Montgomery's algorithm, the multiplier core of the RSAγ crypto chip differs in the following characteristics:

- *Implemented algorithms* – RSAγ uses an optimized variant of Barret's modular reduction method, termed *FastMM algorithm* [MPPS95], instead of the more frequently used Montgomery algorithm. The FastMM algorithm is very well suited for hardware implementation as it avoids the division in the modular reduction operation and calculates a modular multiplication by three simple n-bit multiplications and one addition. Additionally, the RSAγ crypto chip can exploit the Chinese Remainder Theorem (CRT) to speed up the decryption process [SV93].
- *Multiplier architecture* – From an architectural viewpoint, the multiplier in the RSAγ crypto chip is a *partial parallel multiplier* (PPM). The actual prototype contains a $1056*16$ bit PPM, which schedules the multiplicand fully parallel and the multiplier sequentially in 16-bit words. Compared to $r*r$ bit multipliers with $r=32$ or 64, the partial parallel multiplier is much faster because it is able to process the long integers directly. For this reason, the complexity of an n-bit RSA exponentiation is reduced from n^3 to n^2. Due to its high degree of parallelism, the multiplier core computes a 1024-bit modular multiplication in 227 clock cycles. Additionally, pipelining significantly increases the throughput in RSA encryption.
- *Circuit technique and design methodology* – Although the architecture (theoretically) may accept an arbitrary degree of parallelism, it must be noticed that area and power resources are limited on a single chip. Therefore, the goal of achieving optimum performance involves low-power as well as low area design. The *true single-phase circuit technique* (TSPC) turns out to be useful for applications that can take use of pipelining and massive parallelism [YS89]. The RSAγ datapath is implemented in non-precharged TSPC logic to simplify the clock generation and clock distribution. Since the whole multiplier core consists of very few basic cells and is highly regular, it can be realized rather simple in a *full-custom design methodology.*

The rest of the paper is structured as follows: Section 2 describes the implemented algorithms for exponentiation and modular multiplication in detail. Section 3 covers the architecture of the RSAγ multiplier core and explains how it executes a simple multiplication and a modular multiplication, respectively. In section 4, VLSI design related topics like floorplanning and clock distribution are sketched. The paper finishes with conclusions in section 5.

2 Implemented Algorithms

In order to develop high-speed RSA hardware, we not only need good and efficient algorithms for modular arithmetic, but also a multiplier architecture which has to be optimized for those algorithms. This section presents hardware algorithms for exponentiation, modular reduction and modular multiplication.

2.1 Binary Exponentiation Method

When applying the binary exponentiation method (also known as *square and multiply algorithm*), a modular exponentiation $C^E \bmod N$ is performed by successive modular multiplications [Knu69]. The MSB to LSB version of the algorithm is frequently preferred against the LSB to MSB one, because the latter requires storage of an additional intermediate result. Modular reduction after each multiplication step avoids the exponential growth in size of the intermediate results. The square and multiply algorithm needs $3n/2$ modular multiplications for an n-bit exponent E, assuming the exponent contains roughly 50% ones. Therefore, the efficient implementation of the modular multiplication is the key to high performance.

2.2 Barret's Modular Reduction Method

In 1987, Paul Barret introduced an algorithm for the modulo reduction operation which he used to implement RSA encryption on a digital signal processor [Bar87]. At a first glance, a modular reduction is simply the computation of the remainder of an integer division:

$$Z \bmod N \;=\; Z - \left\lfloor \frac{Z}{N} \right\rfloor N \;=\; Z - qN \quad \text{with} \quad q = \left\lfloor \frac{Z}{N} \right\rfloor \tag{1}$$

But, compared to other operations, even to the multiplication, a division is very costly to implement in hardware. Barret's basic idea was to replace the division with a multiplication by a precomputed constant which approximates the inverse of the modulus. Thus the calculation of the exact quotient $q = \lfloor \frac{Z}{N} \rfloor$ is avoided by computing the quotient $\widetilde{q}$ instead:

$$\widetilde{q} \;=\; \left\lfloor \frac{\left\lfloor \frac{Z}{2^{n-1}} \right\rfloor \left\lfloor \frac{2^{2n}}{N} \right\rfloor}{2^{n+1}} \right\rfloor \tag{2}$$

Although equation (2) may look complicated, it can be calculated very efficiently, because the divisions by 2^{n-1} or 2^{n+1}, respectively, are simply performed by truncating the least significant $n-1$ or $n+1$ bits of the operands. The expression $\lfloor 2^{2n}/N \rfloor$ only depends on the modulus N and is constant as long as the modulus does not change. This constant can be precomputed, whereby the modular reduction operation is reduced to two simple multiplications and some operand truncations.

When performing a modular reduction according to Barret's method, the result may not be fully reduced, but it is always in the range 0 to $3N-1$. Therefore, one or two subtractions of N could be required to get the exact result.

2.3 FastMM Algorithm

A closer look at Barret's algorithm shows that truncation of operands at $n-1$ or $n+1$ bit borders are necessary. For reasons of regularity, it would be advantageous to apply truncations only at multiples of the wordsize w of the multiplier hardware (usually 16 or 32 bits) rather than at the original bit positions. Therefore, we modified Barret's algorithm in order to apply the truncations only at multiples of w, whereby these truncations can be performed by successive w-bit right-shift operations. In the modified Barret reduction, the quotient $\widetilde{q}$ is calculated as follows [MPPS95]:

$$\widetilde{q} = \left\lfloor \frac{\left\lfloor \frac{Z}{2^{n-w}} \right\rfloor \left\lfloor \frac{2^{2n+w}}{N} \right\rfloor}{2^{n+2w}} \right\rfloor \tag{3}$$

The *FastMM algorithm* combines multiplication and modified Barret reduction to implement the modular multiplication by three multiplications and one addition according to the following formulas:

$$\left.\begin{array}{rl} Z & = A[n+2w-1\,..\,0] \cdot B[n+2w-1\,..\,0] \\ Q & = Z[2n+w-1\,..\,n-w] \cdot N1[n+2w-1\,..\,0] \\ NegR & = Q[2n+4w-1\,..\,n+2w] \cdot NegN[n+2w-1\,..\,0] \\ X & = Z[n+2w-1\,..\,0] + NegR[n+2w-1\,..\,0] \end{array}\right\} X = A \cdot B \bmod N + eN$$

The values in the squared brackets indicate the bit positions of the operands. All three multiplications and the addition are performed with $n+2w$ significant bits. The FastMM algorithm uses two constants, $N1$ and $NegN$, to calculate the (possibly not fully reduced) result of the modular multiplication. Since these two constants only depend on the modulus N, they can be precomputed:

$$N1 = \left\lfloor \frac{2^{2n+w}}{N} \right\rfloor \tag{4}$$

$$NegN = 2^{n+2w} - N \quad (NegN \text{ is the two's complement of } N) \tag{5}$$

Precomputation of $N1$ and $NegN$ has no significant influence on the overall performance if the modulus N changes rarely compared to the data.

The constant $N1$ approximates the exact value of $\frac{2^{2n+w}}{N}$ with limited accuracy, therefore some error eN is introduced when calculating the result X. An exact analysis of the FastMM algorithm according to [Dhe98] shows that the result of the modular multiplication is given as $AB \bmod N + eN$ with $e \in \{0, 1\}$. This means that X might not be fully reduced, but is in the range 0 to $2N-1$, thus the error is at most once the modulus.

When applying the square and multiply algorithm together with FastMM algorithm to calculate a modular exponentiation, no correction of the intermediate results is necessary. Although the intermediate results are not always fully reduced, continuing the exponentiation with an incomplete reduced intermediate result does not cause an error bigger than once the modulus. An exact proof with detailed error estimation can be found in [Dhe98] and [PP89]. If a final modular reduction is necessary after the modular exponentiation has finished, it can be performed by adding $NegN$ to the result. Thus the final reduction requires no additional hardware effort or precomputed constants.

2.4 Chinese Remainder Theorem

Taking advantage of the Chinese Remainder Theorem (CRT), the computational effort of the RSA decryption can be reduced significantly. If the two prime numbers P and Q of the modulus N are known, the modular exponentiation can be performed separately $\bmod P$ and $\bmod Q$ with shorter exponents, as described in [QC82] and [SV93]. Since the length of the operands is about $n/2$, there are only about $3n/4$ modular multiplications needed for a single exponentiation. The RSAγ crypto chip is able to compute both exponentiations in parallel, as the n-bit multiplier core can be divided into two $n/2$-bit multipliers. Running the two $n/2$-bit multipliers in parallel allows both CRT related exponentiations to be computed simultaneously. Compared to the non-CRT based RSA decryption performed on an n-bit hardware, utilizing the CRT results in a speed-up factor of almost 4.

3 Multiplier Architecture

In the previous section we explained how a modular exponentiation can be calculated by continued modular multiplications, and how three simple multiplications and one addition result in a modular multiplication. The multiplier hardware introduced in this section is optimized for the execution of long integer multiplications according to the FastMM algorithm.

3.1 Partial Parallel Multiplier

Figure 2 illustrates the architecture of the high-speed partial parallel multiplier (PPM) of the RSAγ crypto chip. In order to reach a high degree of parallelism, a wordsize of $w = 16$ was chosen for the PPM. The actual RSAγ prototype is optimized for a modulus length of $n = 1024$, thus the multiplier core has a dimension of $(n+2w)*w = 1056*16$ bit. The PPM could be implemented in an *array-type architecture* [Rab96] or a *Wallace tree architecture* [Wal64]. It turns out that the array architecture is the better choice since it offers a more regular layout and less routing effort, especially when Booth recoding is applied.

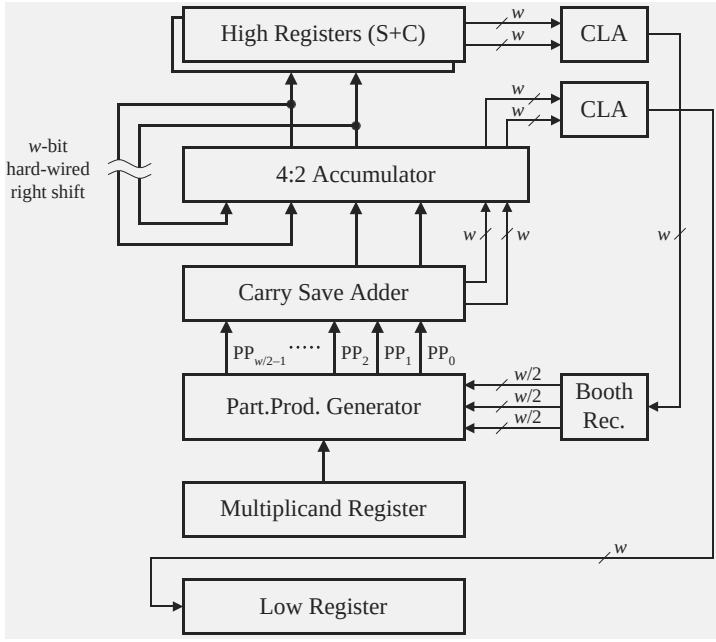


Fig. 2. Basic architecture of the partial parallel multiplier.

Booth recoding [Mac61] is implemented to halve the number of partial products, which almost doubles the multiplication speed with low additional hardware effort. According to the wordsize w, the PPM processes $w/2$ partial products of n bits at once. Since Booth recoding incorporates a radix 4 encoding of the multiplier, it requires a more complex *partial product generator* (PPG) and a *Booth recoder circuit* (BR). The Booth recoder circuit is needed to generate the appropriate control signals for the PPG. Assuming $w = 16$ and $n = 1024$, eight PPGs are required to calculate the partial products, whereby each PPG consists of 1024 Booth multiplexers.

In order to reduce these $w/2$ partial products to a single redundant number, the array multiplier needs $w/2-2$ *carry save adders* (CSA), assuming that the first three partial products are processed by one CSA. Each CSA in the multiplier core consists of n half-cycle full adders presented in [Sch96], which introduce only low latching overhead and allow the maximum clock frequency to be kept high.

The output of the CSA is accumulated to the current intermediate sum in a carry save manner, too. This implies a carry save *accumulator* circuit performing a 4:2 reduction. The accumulator circuit also consists of half cycle full adders, thus the 4:2 reduction is finished after one clock cycle. Aligning the intermediate sum to the next CSA output is done by a w-bit hard-wired right shift operation. Beside the carry save adders, also two *carry lookahead adders* (CLA) are required to perform a redundant to binary conversion of 16-bit words. Redundant to

binary conversion is a 2:1 reduction, therefore a pipelined version of the CLA proposed in [BK82] is used to overcome the carry delay.

Additionally, the PPM also consists of seven *registers,* each of them is n bits wide: *HighSum* and *HighCarry, Low, Multiplicand, Data, N1* and *NegN.* Note that the registers *Data, N1* and *NegN* are not shown in figure 2. The registers *HighSum* and *HighCarry* store the upper part of the product. Two registers are necessary since the upper part is only available in redundant representation. Register *Low* receives the lower part of the result and register *Multiplicand* contains the actual multiplicand. The register *Data* is commonly used for storing the n-bit block of ciphertext/plaintext to become de/encrypted. The registers *N1* and *NegN* contain the two precomputed constants for the FastMM algorithm. All seven registers and the accumulator are connected by an n-bit bus to enable parallel register transfers.

3.2 Execution of a Simple Multiplication

In order to explain how the PPM executes a single multiplication, let us assume a modulus length of $n = 1024$ bits and a wordsize of $w = 16$. This means that each shift operation of the registers results in a 16-bit right-shift of the stored value. At the beginning of a multiplication, the multiplicand resides within the register *Multiplicand,* and the multiplier (which is assumed to be available in redundant representation) resides within the registers *HighSum* and *HighCarry.* A single multiplication takes place in the following way:

1. Within each step, a 16-bit word of the (redundant) multiplier is shifted out of the registers *HighSum* and *HighCarry,* starting with the least significant word. These 16-bit words are converted from redundant into binary representation by a CLA, which requires three clock cycles.
2. When the 16-bit binary multiplier word reaches the Booth recoder circuit, it generates the control signals for the PPG. The PPG calculates a set of eight partial products, which is propagated to the CSA. Booth recoding and the distribution of the control signals requires two clock cycles.
3. Within three clock cycles, the CSA reduces the set of eight partial products to a single redundant number. However, as the CSA is a pipelined circuit, one set of eight partial products can be processed each clock cycle.
4. The output of the CSA is then accumulated to the current intermediate sum within one cycle. Now, the least significant 16 bits of the intermediate sum already represent a word of the lower part of the product.
5. A CLA is used again to convert the redundant 16-bit words of the lower part into binary representation. Subsequently, the binary words are shifted into the register *Low,* where they finally represent the complete lower part of the product.
6. After the last set of eight partial products has been processed, the upper part of the result resides within the accumulator after three cycles and can be loaded into the registers *HighSum* and *HighCarry.*

Whenever the upper part of the product is needed as operand for the next multiplication, it can be used as the multiplier which is allowed to be redundant. The lower part of the product always appears in binary representation; therefore it might be used as multiplicand or as multiplier. The following table summaries the operand requirements and appearance of the product.

operand	representation	schedule
multiplier	bin. or red.	sequentially, w-bit words
multiplicand	binary	parallel, begin of multiplication
lower part of product	binary	sequentially, w-bit words
upper part of product	redundant	parallel, end of multiplication

The steps needed for a single multiplication depend on the length of the modulus n and the wordsize w on which the multiplier operates. The actual RSAγ prototype has a wordsize of $w = 16$ and needs exactly 80 clock cycles for a single 1024-bit multiplication.

3.3 Execution of a Modular Multiplication

When applying the square and multiply algorithm, a modular exponentiation is performed by successive square and multiply steps. For a square step, the result of the previous modular multiplication, which resides within the register *Low,* acts as both, multiplicand as well as multiplier. For a multiply step, the n-bit block of data to become de/encrypted is the multiplicand, and the result of the previous modular multiplication is the multiplier.

According to the FastMM algorithm presented in section 2.3, a modular multiplication takes place in the following way:

1. For multiplication 1 of the FastMM algorithm, the (redundant) registers *High* are loaded from register *Low* and register *Multiplicand* is either loaded from register *Low* (square step) or from register *Data* (multiply step). After the multiplication has been performed as described in section 3.2, the lower part of the result resides within the register *Low* and the upper part resides within the accumulator.
2. For multiplication 2 of the FastMM algorithm, the (redundant) registers *High* are loaded from the accumulator and register *Multiplicand* is loaded from register *N1.* The multiplication is performed as described in section 3.2, but without shifting the words of the lower part result into register *Low.* Note that only the upper part of the result from multiplication 2 is needed for the next multiplication. Register *Low* still contains the lower part result of multiplication 1.
3. For multiplication 3 of the FastMM algorithm, the (redundant) registers *High* are loaded from the accumulator and register *Multiplicand* is loaded from register *NegN.* The multiplication is performed as described in section 3.2, but the accumulator is initialized with the lower part result of multiplication 1. Thus, multiplication 3 is performed together with the addition of the

FastMM algorithm. The result of the modular multiplication resides within register *Low* after multiplication 3.

A modular multiplication for 1024-bit operands requires 227 clock cycles when it is performed on a PPM with a wordsize of $w = 16$.

4 Floorplanning and Clocked Folding

From the viewpoint of floorplanning, the RSAγ datapath shown in figure 2 can be divided into two parts: a *regular part* which includes all n-bit wide circuits (registers, carry save adders, partial product generators, and the accumulator), and a *peripheral part* which consists of all the other circuits (CLA, booth recoder, and control logic). Since the regular part is about 80% of the total chip area, its layout is subject of detailed optimization. In order to exploit the regularity of the datapath structure, the regular part is built of $n+2w$ identical copies of a one bit *slice,* as illustrated in figure 3. This slice consists of seven 1-bit register cells, eight 1-bit adder cells and eight 1-bit partial product generator cells, including a uniform inter-cell routing.

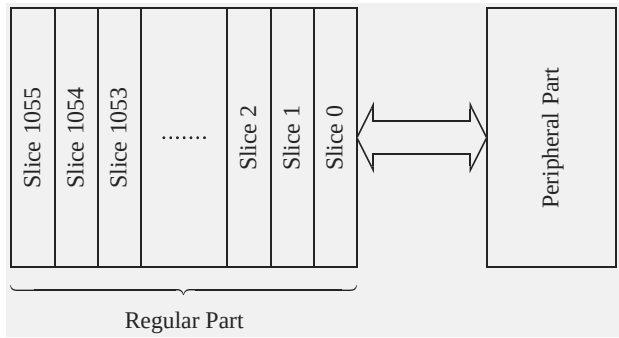


Fig. 3. Slice orientation of the regular part layout.

A slice-based layout for the regular part of the RSAγ crypto chip has two significant advantages:

1. The place-and-route procedure needs to be solved only for a single slice.
2. As all bit positions have a uniform layout, the verification process (parasitics extraction, timing simulation, back annotation) is simplified. If a particular timing is verified within a single slice, it is also verified in all other slices.

The slices are supposed to connect by abutment, as also the routing between the slices is uniform. Since the wire length of control signals and inter-slice routing grows with the width of a single slice, narrow slices reduce the total area demand.

Based on a 0.6μ CMOS process, it turns out that the slice width is limited to approximately 35 μm with a corresponding slice length of approximately 1 mm, which results in a datapath layout size of approximately $35*1$ mm. Such an aspect ratio would be unacceptable, because chip packages are most frequently considered for square shapes. Not only packaging requires a fairly square shaped chip layout, also the distribution of global signals (e.g. control signals, output signals of the booth recoder) is much easier when the layout has an aspect ratio close to 1, since the corresponding wires are significantly shorter. Also minimizing the clock skew is very important. Again, delays due to interconnection must be minimized.

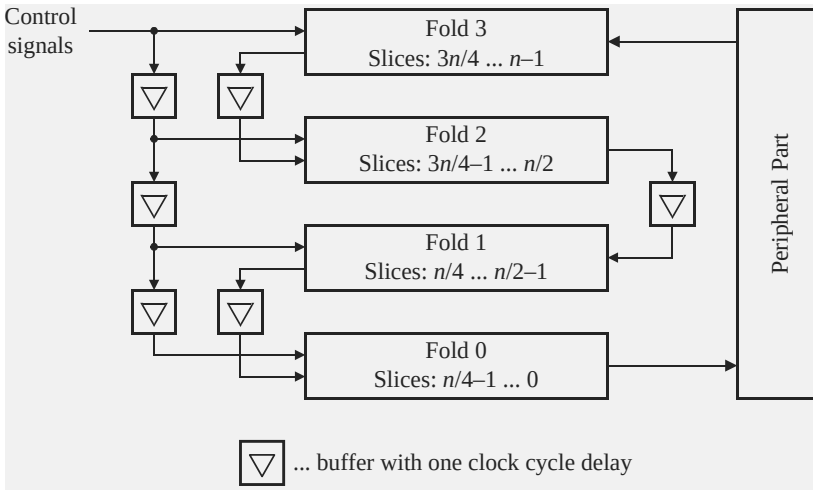


Fig. 4. The clocked folding principle.

In order to fulfill the requirement for a square shaped layout, a special floorplanning technique termed *folding* is applied to the regular part of the RSAγ datapath. The 1056 bits wide regular part is divided into four folds, whereby each fold consists of 264 slices, as illustrated in figure 4. Note that folding can be applied if and only if the direction of data signal flow is restricted from more to less significant bit positions. It is not difficult to see that the components shown in figure 2 can be arranged in a way to meet this restriction.

But folding has also a serious disadvantage: transmitting data signals between folds requires long interconnection wires. Therefore, buffers have to be inserted to drive the increased capacitive load. The additional delay caused by the buffers and the interconnection wires would compromise the overall performance. But this pipeline bottleneck can be removed by inserting buffers with one cycle delay between the folds. Since the data signal flow is limited from more to less significant bit positions, the architecture allows one cycle delay between

subsequent folds. When all data input signals for fold 2 are provided by fold 3, the control signals can also be delayed by one cycle, causing calculations taking place in fold 2 to be delayed by one cycle. Likewise, calculations in fold 1 and fold 0 are delayed by two and three cycles, respectively. The additional delay of three cycles caused by this *clocked folding* does not compromise overall latency very much. But on the other hand, buffers with one cycle delay allow much higher clock rates than ordinary buffers.

5 Summary of Results and Monclusions

The subject of this paper was to present efficient algorithms for modular arithmetic and a multiplier architecture which is optimized for these algorithms. The prototype of the RSAγ crypto chip is designed for a modulus length of $n = 1024$ bits and a multiplier wordsize of $w = 16$. Based on a $0{,}6\mu$ standard CMOS process with one poly layer and two metal layers, the silicon area of the multiplier core is about 70 mm^2 and contains approximately 10^6 transistors. The execution of a 1024-bit modular multiplication requires 227 clock cycles. When the multiplier core is clocked with 200 MHz, this results in a decryption rate of 560 kbit/s. In CRT mode, the decryption rate increases to 2 Mbit/s. The high performance confirms the efficiency of the implemented hardware algorithms and the multiplier architecture. Furthermore, the proposed design is highly scalable with respect to the multiplier wordsize as well as the modulus length. The most limiting factor is the available silicon area. A modern $0{,}25\mu$ CMOS process would allow to increase the multiplier wordsize to $w = 32$, which doubles the performance.

References

Bar87. P. Barrett. Implementing the Rivest, Shamir and Adleman public-key encryption algorithm on a standard digital signal processor. In A. M. Odlyzko, *Advances in Cryptology – CRYPTO '86 Proceedings*, vol. 263 of *Lecture Notes in Computer Science*, pp 311–323, Springer-Verlag, 1987.

BK82. R. P. Brent and H. T. Kung. A regular layout for parallel adders. *IEEE Transactions on Computers*, C-31(3), pp. 260-264, 1982.

BP99. T. Blum and C. Paar. Montgomery Modular Exponentiation on Reconfigurable Hardware. *Proceedings of the 14th IEEE Symposium on Computer Arithmetic*, pp. 70–77, 1999.

DH76. W. Diffie and M. E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, IT-22(6) pp. 644–654, November 1976.

Dhe98. J. F. Dhem. *Design of an efficient public-key cryptographic library for RISC-based smart cards.* Thesis (Ph.D.), Université catholique de Louvain, Louvain-la-Neuve, Belgium, 1998.

Knu69. D. E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming.* Addison-Wesley, Reading, MA, USA, 1969.

Mac61. O. L. MacSorley. High-Speed Arithmetic in Binary Computers. *Proceedings of the Institute of Radio Engineers*, 49:67–91, 1961.

Mon85. P. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170) pp. 519–521, 1985.

MPPS95. W. Mayerwieser, K. C. Posch, R. Posch, and V. Schindler. Testing a High-Speed Data Path: The Design of the RSAβ Crypto Chip. *J.UCS: Journal of Universal Computer Science*, 1(11) pp. 728–744, November 1995.

NM96. D. Naccache and D. M'Raïhi. Arithmetic co-processors for public-key cryptography: The State of the Art. *IEEE Micro*, pp. 14–24, June 1996.

Nat94. National Institute of Standards and Technology (NIST). *FIPS Publication 186: Digital Signature Standard*. National Institute for Standards and Technology, Gaithersburg, MD, USA, May 1994.

PP89. K. C. Posch and R. Posch. Approaching encryption at ISDN speed using partial parallel modulus multiplication. IIG report 276, Institutes for Information Processing Graz, Graz, Austria, November 1989.

QC82. J.-J. Quisquater and C. Couvreur. Fast decipherment algorithm for the RSA public-key cryptosystem. *IEE Electronics Letters*, 18(21), pp. 905–907, October 1982.

Rab96. J. M. Rabaey. Digital Integrated Circuits - A Design Perspective. *Prentice Hall Electronics and VLSI Series*, Prentice Hall, Upper Saddle River, NJ, USA, 1996.

RSA78. R. L. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public Key Cryptosystems. *Communications of the Association for Computing Machinery*, 21(2) pp. 120–126, February 1978.

Sch96. V. Schindler. A low-power true single phase clocked (TSPC) full-adder. *Proceedings of the 22nd ESSCIRC*, Neuchâtel, Switzerland, pp. 72–75, 1996.

SV93. M. Shand and J. Vuillemin. Fast Implementation of RSA Cryptography. *Proceedings of 11th Symposion on Computer Arithmetic*, 1993.

Wal64. C. S. Wallace. A suggestion for a fast multiplier. *IEEE Transactions on Electronic Computation*, EC-13(1), pp. 14–17, 1964.

YS89. J. Yuan and C. Svensson. High-speed CMOS circuit technique. *IEEE Journal of Solid-State Circuits*, 24(1), pp. 62–70, 1989.

Data Integrity in Hardware for Modular Arithmetic

Colin D. Walter

Computation Department, UMIST
PO Box 88, Sackville Street, Manchester M60 1QD, UK
http://www.co.umist.ac.uk

Abstract. An increasing mass market for cryptographic products leads to greater pressure on companies to fabricate chips which will recover from, and correct, sporadic errors resulting from design and fabrication faults, inadequate testing, smaller technology, ionising radiation, random noise, and so on. Where encryption is subject to such errors, large quantities of data can become totally corrupted or inaccessible unless fault detection is an integral part of the hardware arithmetic. Here realistically cheap methods are examined for checking the correctness of the arithmetic computations which are the basis of the RSA cryptosystem and Diffie-Hellman key exchange. As in ordinary integer multiplication, a modular residue checker function is used to detect errors and trigger re-computation when necessary. The mechanism will also detect most permanent faults. Some suggestions are made on how to correct infrequent errors without using additional hardware.
Keywords: Computer arithmetic, cryptography, RSA, modular multiplication, modular exponentiation, soft errors, error correction, fault tolerance, checker circuit, testing, correctness, data integrity, Montgomery multiplication.

1 Introduction

Mass production of embedded cryptographic systems is fast approaching for applications ranging from electronic purses and e-commerce authentication to secure mobile video telephony. Chip technology for these has advanced to the point where random effects, such as noise and ionising radiation, are already causing so many errors that the aerospace industry regularly performs computations three times and takes a majority decision [1]. Indeed, some attacks on cryptosystems involve the introduction of such transient hardware errors to perform differential fault analysis [3]. But faults can occur at any point in the process from design to fabrication as well as during operation. Consequently, as with other products, incorporation of fault tolerance methods should mean increased yield from chip fabrication, less expensive testing and higher customer satisfaction during operation. The disaster with the Pentium division algorithm [2] illustrates the company critical issues of releasing faulty products even when errors are extremely rare. So, in the light of such experience, it has been suggested that checking should become an integral part of all arithmetic operations beyond those with the simplest implementations [2].

Ç.K. Koç and C. Paar (Eds.): CHES 2000, LNCS 1965, pp. 204–215, 2000.

Standard error correction coding techniques are not generally applicable to arithmetic operations. So incorrect functioning of the ALU cannot usually be detected this way. Moreover, whilst all 32-bit operations might be fully tested before each unit is shipped, this is not realistic for the larger co-processors which might soon be employed in a typical RSA implementation. Nevertheless, excellent test suites can still be built for RSA hardware [10]. Duplication and triplication of hardware for non-safety critical fault recognition is too expensive, and in any case does not solve design faults.

Whilst any error will almost certainly generate random junk which is immediately detected on decryption, it is not always easy to signal this and request the recomputation, especially when this then invokes two way communication between the parties involved. Indeed, storing incorrectly encrypted data or session keys on disk or smartcard memory may not be detected for some time. In the case of message signing, the inverse process of signature verification is often a relatively cheap way of checking the computation [8], §3. However, with RSA encryption [9], checking by decrypting (a large exponent) requires knowledge of a secret key, which may not be available, and is also much more expensive than the encryption (a small exponent). So this form of verification is generally impossible or uneconomic. Furthermore, it is well understood that the consequent re-encryption of the same data after a glitch can leak secret data from an embedded system [3]. Thus, correctness should be verified before any output is released and an identical recomputation avoided in making any correction.

The aim of this paper is to consider much more cost effective alternatives than decrypting everything or duplicating hardware. We first show how to apply a cheap residue check which, with high probability, will find any intermittent or random arithmetic fault. We will argue that it will also detect other errors caused by permanent physical and logical flaws which have passed unnoticed during design, production and testing or which develop during use. We then describe how to correct such errors by modifying arguments in such a way as to avoid performing the same flawed calculation again. The efficacy of the check is discussed as well as the checking frequency. We conclude with an assessment of the time and area costs of the method.

2 Notation

The RSA algorithm [9] uses a public modulus M which is the product of two large primes, typically of around 2^9 bits each. For keys d and e, encryption of plain text T in the range $[0, M-1]$ and decryption of cipher text C are defined by $C = T^e \bmod M$ and $T = C^d \bmod M$ respectively. One of the keys d, e is kept secret, and the two satisfy the property $de \equiv 1 \bmod \phi(M)$ where ϕ is Euler's totient function. The strength of the system depends on the difficulty of factorising M, which is required in order to deduce one key from the other.

Hardware implementations of the cryptosystem often use a high radix or base for representing numbers. Typically this is a power of 2 such as 2^{16} or 2^{32} corresponding to the size of multiplier available. Let r denote this radix and n

the number of base r digits in the modulus M. We will not encounter numbers larger than rM, so that a number A always has a representation

$$A = \sum_{i=0}^{n} a_i r^i$$

(The extra top digit may be required because occasionally numbers greater than M are encountered, in particular, just prior to modular reductions.) Exponentiation is performed by repeated modular multiplication, which in turn is performed by repeated modular addition. Thus the key operation is calculating products $P = (A \times B) \bmod M$ using a close relative of the following standard algorithm:

Classical Modular Multiplication Algorithm:

```
P <- 0 ;
For i <- n downto 0 do
Begin
   P  <- rP + a_i B ;
   q_i <-  P div M ;
   P  <-  P - q_i M ;
End
{ Post-condition: P = (A×B) mod M }
```

The initially generated sequence of digits q_j ($j = n, n-1, ..., i$) can be formed into an integer $Q_i = \sum_{j=i}^{n} q_j r^{j-i}$ and the initially consumed digits of A form a similarly defined integer A_i. Then it is easy to verify by induction that $P = A_i \times B - Q_i \times M$ and $0 \le P < M$ are invariants which hold at the end of each iteration of the loop. Hence the given post-condition holds when the loop terminates and, for $Q = Q_0$,

$$P = A \times B - Q \times M \tag{1}$$

Some dedicated hardware implementations of RSA with small radix r (typically $r = 2$ or 4) provide combinational logic circuitry for the equivalent of a complete modular addition cycle

$$P \quad \leftarrow \quad rP + a_i B - q_i M \tag{2}$$

Then, for speed, only an approximate value for q_i is used and this is calculated in advance from P. It is sufficiently accurate to keep P less than a small multiple of M, often $2M$ or rM. So a small, final modular subtraction may be necessary to obtain a result P in the range $[0, M-1]$. If we assume this extra modular correction is incorporated into P and Q then their final values still satisfy (1) and Q is again the integer quotient $(A \times B) \operatorname{div} M$.

There is a widely used alternative algorithm due to P. Montgomery [7] which processes the bits of A in the opposite order with a shift of P downwards instead of upwards. The advantage of this is primarily in hardware implementations rather than in software: successive modular reductions can commence without

waiting for carries to propagate over the full length of the adder. The algorithm is the following, but it computes a shifted modular product instead, namely $(A{\times}B{\times}R^{-1}) \bmod M$ where $R = r^{n+1}$.

Montgomery's Modular Multiplication Algorithm:

```
P <- 0 ;
For i <- 0 to n do
Begin
    q_i <- (P + a_i B)(-M^-1) mod r ;
    P   <- (P + a_i B + q_i M) div r ;
End
{ Post-condition: P ≡ (A×B×R^-1) mod M for R = r^n+1 }
```

For $M^{-1} \bmod r$ to be defined properly, we require M to be prime to r. Invariably, r is a power of 2 and M is odd, so this is not a significant restriction. Observe that the definition of q_i means that the division by r is exact. Hence $A{\times}B$ is computed, reduced by a multiple of M, and shifted by R. It is easy to obtain a bound on the size of the output P, e.g. $P < B{+}M$, which shows that it is the least non-negative residue $(A{\times}B{\times}R^{-1}) \bmod M$ to within a known, very small multiple of M [11]. The $\bmod r$ operation is fast because it only depends on the lowest digits of M, B and P, and the $\operatorname{div} r$ operation is fast because it only involves a hardware shift.

As with the classical algorithm above, the initially generated sequence of digits q_j $(j = 0, 1, ..., i)$ can be formed into an integer $Q'_i = \sum_{j=0}^{i} q_j r^j$ and the initially consumed digits a_j $(j = 0, 1, ..., i)$ of A form a similarly defined integer A'_i. Then it is easy to verify by induction that

$$P = (A'_i{\times}B + Q'_i{\times}M)/r^{-i-1} \tag{3}$$

is an invariant which holds at the end of each iteration of the loop. Taking $Q = Q'_n$, when the loop terminates,

$$P{\times}R = A{\times}B + Q{\times}M \tag{4}$$

So the post-condition holds. (The analogy with (1) is that Q is an r-adic approximation to the quotient $(-A{\times}B)/M$.) A small, final modular subtraction may be necessary to obtain a result P in the range $[0, M{-}1]$. If we assume this extra modular correction is reflected in a corresponding update to Q, then the final values of P and Q still satisfy (4).

3 A Simple Check for Soft Errors

A standard choice, [5] §7, for a checker function f in integer arithmetic is

$$f(A) = A \bmod D \tag{5}$$

where $D > 1$ is a suitable small number prime to at least $2r$, such as 15. This function is easily computed but fails to commute with the arithmetic operations of modular arithmetic. Ideally, for the arithmetic operation $\otimes$ which we wish to check, what is needed is a function f from integers $\bmod M$ to integers $\bmod D$ with the property

$$f(A \otimes B) = f(A) \otimes f(B)$$

for residues A and B in the ring of integers $\bmod M$. However (5) fails to have this property unless D divides M. The solution is to go back to the non-modular integers that the machine uses for its representation and take into account the modular subtractions made by the system. So, if P is the integer representing the result of the calculation of $A \otimes B$ during which Q subtractions of M are made, then

$$P = A \otimes B - Q \times M \tag{6}$$

The function f of (5) can be applied to this integer relation to obtain that

$$f(P) = f(A) \otimes f(B) - f(Q) \times f(M) \tag{7}$$

holds $\bmod D$ if all the calculations involved have been performed correctly.

This applies to any modular arithmetic operation $\otimes$ from addition to exponentiation and, in particular, to modular multiplication. From here on we will interpret $\otimes$ as the particular modular multiplication operation of interest to us. [1] So (6) translates into (1) or (4). These equations re-phrase the output of the multiplication process entirely in terms of non-modular arithmetic operations and, as stated, enable the checker function f to be applied. Then the main property (7) to check becomes, respectively,

$$f(P) \equiv f(A) \times f(B) - f(Q) \times f(M) \mod D \tag{8}$$

or

$$f(P) \times f(R) \equiv f(A) \times f(B) + f(Q) \times f(M) \mod D \tag{9}$$

A difference between the left and right sides guarantees an error somewhere (although perhaps in computing f rather than $\otimes$) and, conversely, we will see that agreement is rare when the computation of $A \otimes B$ does contain an error.

[1] In *"Method and apparatus for protecting public key schemes from timing and fault attacks"* (US patent 5,991,415, Nov 23, 1999), Adi Shamir recommends obtaining and checking $A^e \bmod M$ by computing $A^e \bmod MD$ first, reducing this $\bmod M$ for the result, and reducing it $\bmod D$ to check against $(A \bmod D)^e$. This avoids computing $Q \bmod D$. Similarly, any operation might be performed $\bmod MD$ and then reduced $\bmod M$ for the result and $\bmod D$ for the check.

4 The Choice of Modulus D

What is the best choice for D? The smaller D is, the cheaper and easier it is to compute the function $f = f_D$. However, we need to analyse the different possible faults to see how large D has to be to give the required degree of confidence in the correctness of the calculations. It turns out that almost all the hardware can be protected against a single fault with a very reasonable value for D.

To deal with register stuck-at faults, D should divide by a prime which does not divide $2r$. For most number representations likely to be used, any single bit error in the input to f changes that input by a number of the form $2^i r^j$. So, by the divisibility condition, this will be reflected in a different output value for f. Suppose the stuck-at fault is in register P and that register is written to, but not read from, during the multiplication. Then the left side of (7) will be incorrect when the faulty bit is stuck at the wrong value. So it will differ from the correct value computed for the right side. Hence this fault will be caught whenever it occurs, which will be in 50% of all cases on average. Of course, once an error is read from P, errors will start propagating further.

A similar argument applies to register M when it has a stuck-at fault. However, in this case all multiplications in an exponentiation are done correctly if the bit is stuck at the value which M should have, or they are all incorrect if the bit is stuck at the wrong value. Since $f(Q)$ will be 0 in $1/D$ of all cases, the equation (7) will not detect an error every time one occurs. However, over a single exponentiation which involves at least several multiplications, $f(Q)$ is unlikely always to be 0. So, if every multiplication is checked, the error should eventually be detected during the exponentiation, providing the calculation of $f(M)$ is based on the value of M kept in memory rather than the value in the faulty register used by M during the modular multiplication. Indeed, the possibility of errors in copying from and writing to memory illustrates the benefit of storing the checker function value with the number itself, in the same way as a parity bit.

Most registers used by a modular multiplication, apart from that holding M, will be both written to and updated a number of times, resulting in a propagation of errors. One might reasonably assume that this leads to the values on the left and right sides of (7) being essentially independent, so that $1-D^{-1}$ of all errors in multiplications are detected. (The undetected cases arise from the value in error being multiplied by 0.) Consequently, virtually all incorrect exponentiations will be detected, especially if each multiplication is checked, and permanent faults will be detected with greater probability than transient faults because more checks may contain the error.

A similar argument applies for faults in the combinational logic of a digit slice of an adder used to perform (2) or the equivalent step in Montgomery's method. The adder has three inputs, of which B and M are scaled by a digit and B may have a redundant form. At the level of the jth digit slice, the equation for the classical algorithm is

$$p_j + r \times c_{out} \quad \leftarrow \quad p_{j-1} + a_i \times b_j - q_i \times m_j + c_{in} \tag{10}$$

where c_{in} and c_{out} are carries from/to neighbouring slices and b_j and q_j may have redundant forms. Hardware for computing this may be repeated for every digit, or instead there will be a digit multiplier and adder which is reused for each digit position. For convenience, let us ignore the negative sign and assume all quantities are positive. (In practice there is a borrow to achieve this.) Typically the non-redundant digits might be bounded above by $r-1$, the redundant digits by $2r-1$ and the carry by $4r-2$. Then the expression on the right is bounded above by $4r^2-r-1$, which splits into a non-redundant digit of P and a carry still bounded by $4r-2$. Thus each line into, or out of, the combinational logic of the jth digit slice typically represents a value dr^j where d is a small power of 2 equal to, or less than, $2r^2$. Then summing the output values for all lines will give a total bounded above by $4r^2-1$. In this case, any error within the slice will make an absolute difference to the output also of the form dr^j where now $d < 4r^2$. Our desire is that any such difference should make a non-zero change to $f(P)$, i.e. the change should not be divisible by D. Thus any D larger than and prime to $4r^2$ is acceptable as it will detect all such single errors. In general, whatever the circuitry and bounds on the digit values, any value larger than the sum of all digit slice output lines would do for D. If some output values d cannot arise without multiple errors, a smaller choice for D might well be possible. All such possible values of d can easily be determined from the circuit design before fabrication, and the tendency will be for d to be a multiple of 2 times a small odd number.

The digit slice error may propagate in two ways, depending on whether it is transient or not. With a permanent fault, a substantial proportion of the addition cycles are likely to be affected in the same way. As RSA multiplications contain many addition cycles, $f(P)$ is most likely to change in a way which makes the differences between correct and incorrect values uniformly distributed $\bmod D$, even although they may all be multiples of the above d. Then the checker function will detect all but $1/D$ of the errors which occur. However, with a transient fault, the difference between the correct and computed values of P is shifted up or down by a power of r on each iteration. So its initial primeness to D is preserved. Eventually the error may affect the value of Q, but there will be a compensating deduction of a multiple of M from P which will not obscure the difference between the values of the left and right sides of (7). So such errors should always be spotted.

The rest of the combinational logic includes counters, clocks, control circuitry, etc. These subcircuits take less area than the multiplier or digit slices and could mostly be checked by duplication. However, errors there will tend to have a random effect on the outputs, yielding approximately a $1/D$ probability of the residue check falsely approving an incorrect calculation. Hence employing a large D could be an alternative to duplicating such hardware. The main exception is the exponentiation circuitry. Although this controls the sequence of multiplications and so cannot affect the truth of (7), it is usually implemented in software. So this remains unchecked because f only checks hardware arithmetic operations.

Finally, there may be specialised hardware for computing digits of Q. In most cases an error in a digit of Q will either lead to overflow/underflow because after several more iterations during which P is shifted, P will grow too large or become negative. Alternatively, the self-correcting nature of the choice of q_i will successfully compensate for the error. Either way, the possibility of over- or under- flow must be monitored because the equation (7) will not detect such errors: the compensating multiple of M will re-adjust the equation so that it still holds. So a final range check on P might not come amiss.

In summary, most of the hardware is protected against transient and permanent faults by the checker function. When typical redundant representations are used, errors are detected except in at most $1/D$ of cases if we are allowed to choose $D > 4r^2$ and prime to $2r$. For compatibility with the hardware multiplier, it is clearly advantageous to keep $D < r$, which is the built-in size of all non-redundant digits. The arguments above suggest that taking a large $D < r$ with some large prime factors would achieve most or even all of our requirements, its only disadvantage being to limit the probability of detecting some errors. This is efficiently held as a single digit, so we will assume such a choice is made for D. Other alternatives might be to pick a large two-digit D, i.e. one which is less than r^2, or even to use two co-prime values of D, each just less than r. This might be preferred for very small r (such as 2) to retain good detection rates.

5 Time and Area Costs for Checking

The choice of D has implications for the cost of computing f. However, since the processor cycle time is probably determined by the multiplier, it is likely that digit sums and digit products are computed in essentially the same time. We will assume r is a power of 2 and look at two possibilities.

First, suppose D is a divisor of $2^s \pm 1$ for some s. This is the standard situation analogous to the case of checking divisibility of a decimal number by 3, 9 or 11. Suppose A has a standard, non-redundant, binary representation. Then computing $f(A)$ simply requires computing the (possibly alternating) sum of s-bit digits of A (and perhaps repeating this on the result) and then reducing the result $\bmod D$. An obvious choice here is $D = r-1$, for which the digits of A are summed. The result for typical RSA implementations will be a two digit number whose digits are then themselves summed. If the result overflows one digit, D is subtracted by adding 1 to the lower digit to yield a single digit for $f(A)$ after $n+2$ additions overall. Without adding extra, dedicated hardware, taking $D = r-1$ is arguably the most economic solution. If A has a redundant representation, the extra bits must be added into the calculation in the same way, and this may double the number of additions required to obtain $f(A)$.

In general, computing $f(A)$ for some argument A can be performed iteratively from the most significant end using

$$f(A_i) = (f(A_{i+1}) \times f(r) + a_i) \bmod D \tag{11}$$

and $f(A_{n+1}) = f(0) = 0$, or computed similarly from the least significant end using $f(r^{-1})$. For the above choice of $D = r-1$, we have $f(r) = \pm 1$ so that the multiplication is avoided.

Alternatively to this choice, suppose a large $D<r$ is chosen for which $f(r)$ or $f(r^{-1})$, as appropriate, is small. Prime choices for D might be $r-16\pm1$ for $r = 2^{16}$ or $r-5$ for $r = 2^{32}$. Assume, in fact, that $f(r)^2(f(r)+1) < r$. By leaving most of the reduction mod D in (11) to the end, we can obtain $f(A_i) < (f(r)+2)r$ for each i by expressing $f(A_i) = m_i r + l_i$ as a two digit number, where $m_i \leq f(r)+1$ and computing $f(A_i) = m_{i+1} \times f(r)^2 + l_{i+1} \times f(r) + a_i$ instead. This converges and yields $f(A) \equiv m_0 \times f(r) + l_0 < 2D$ if D is large enough, so that one more subtraction of D gives $f(A)$. The cost of computing f therefore amounts to $2n+2$ digit multiply-accumulate operations in this case.

In the context of RSA, $f(M)$ need only be calculated once for a given modulus. Besides this and the exponent, the only other input to an exponentiation is the initial text T for which $f(T)$ must be calculated. Thereafter, for each multiplication, only the check values for the outputs need to be calculated, namely $f(P)$ and $f(Q)$. For the more expensive of above choices, this adds $4n + 4$ digit multiply-accumulate operations to the $2n^2$ required for a full length modular multiplication using (10). A further 3, resp. 4, such operations are required to check (7) via equations (8) and (9) respectively. So adding the checker function should be equivalent to adding at most 1 to the number of digits in M. By including another multiplier in an array of multipliers [11], [6], or extra cycles when there is a single multiplier, the cost can normally be spread over time and area so that both the time and area formulae reflect the increase in n by at most 1.

Finally, for very small values of r, such as $r = 2$ or 4, RSA hardware implementing (10) consists of a full length adder and no multiplier. Then a D of the order $4r^2$ is more appropriate than $D = r-1$. Computing $f(A)$ is straightforward using only additions so that the clock speed is maintained, but more digit additions are required. So, the work resulting from including the checker function corresponds to adding several more digits to n. However, as n is also greater, the proportion of extra work is not increased. It is in fact dependent on the size of D and how well it matches the digit base r.

6 Recovering from Transient Errors

When an error is detected, it may be unwise to continue computations since an attack on the system may be in progress. The checker function can indeed be used to defeat some attacks which operate by inducing transient errors. However, we will assume the system wishes recomputation to be performed. If errors are rare enough it is reasonable to cancel the exponentiation and just start again. This requires a single extra buffer for storing the original text T until the encryption/decryption has been approved. If the checking needs to be done on every multiplication, then, for most exponentiation schemes, it is the output of the previous multiplication which forms the only new argument to the multiplica-

tion which is in error. Thus, again, a single input needs to be buffered until the check is complete.

Since $f(Q)$ can be computed digit serially as its digits are generated, any error can be detected immediately $f(P)$ becomes available. With such large numbers, $f(P)$ would normally also be computed digit serially and is therefore not available until some time after P, unless P is also generated digit serially.

Suppose the modular multiplier uses redundancy to allow parallel digit operations on an array of multipliers and one modular multiplication starts immediately upon termination of the previous one. This is the classical model described by E. Brickell [4]. Now $f(P)$ can be computed using (11) and the check (7) just completed in the time to set up and perform the next modular multiplication. When an error is discovered, two modular multiplications must be discarded, namely the current one and the previous one for which the test has just detected an error. So, by buffering the new input of the current and previous modular multiplications so that such steps can be repeated when necessary, the exponentiation can proceed and be checked with a time penalty equivalent to $2k+1$ extra modular multiplications where k is the number of multiplications containing detected errors. On average, one would expect k to be very close to 0.

More recently, systolic and linear arrays have been combined with Montgomery's algorithm to provide modular multipliers [11], [6]. These avoid some of the drawbacks of the standard design, such as redundancy and digit broadcasting which have time and area penalties. So a slightly faster clock is possible. The arrays operate with digit serial I/O to the multiplier array and, by performing two streams of multiplications in parallel, can have the same throughput in terms of clock cycles despite the inherent problem of only being able to use cells on every alternate cycle. A single multiplication produces one digit only every other cycle, resulting in just over $4n$ time slots between the first digit input and last digit output. Now $f(P)$ can be computed as the digits of P are generated and the correctness check made within a single clock cycle of P being produced. In the case of the linear systolic array, it suffices to buffer the new inputs of the four multiplications currently in progress in the multiplier (one starting and one finishing in each of two interleaved streams) so that the ones just finishing can be recomputed if necessary. The buffers might even be shared between the two streams if the probability of a double error were sufficiently small.

Thus, in addition to the cost for detecting errors, the occurrence of random encryption/decryption errors can be corrected by recomputation with an area cost of only several full length buffers (the precise number being dependent on the implementation), and a time penalty of $2k+1$ extra modular multiplications where k is the number of detected errors. For the smallest radix, $r = 2$, the extra registers may easily double the total hardware area, but as r increases the proportion devoted to registers falls and the relative cost diminishes. However, this solution is still a much cheaper alternative than voting between three copies of the hardware, or using backup registers to enable re-computation when two copies of the hardware fail to agree.

7 Permanent Faults

We have now treated transient errors and seen how most of these can be successfully recognised and recovered from. Finally, permanent errors need consideration. Most design or fabrication faults should be caught during comprehensive production testing [10], but this is expensive and shortcuts are bound to lead to faulty products being delivered. Ideally, as a minimum, every combination of inputs should be tested (i) for every digit slice and (ii) for the computation of q_i. However, as the modulus M is not usually changed very frequently, some errors in the hardware logic may not surface at testing nor even occur during the chip's life.

Whilst the major test of correct encryption is that decryption does not yield rubbish, in RSA one key is always kept private so that such a test for a specific modulus may be denied. Thus, as encrypted text is indistinguishable from rubbish, some kind of on-board checking of output is desirable before destroying the plain text input.

The arguments already presented for detecting transient errors apply almost equally well to detecting permanent faults: the two are indistinguishable if the hardware at fault is only operated once. But, in general, we have already seen that repeated faults will cause (7) to detect all but at most $1/D$ of arithmetic and some other errors, but that logical errors in the computation of q_i are not discovered since both P and Q are affected equally. In particular, our experience of building previous chips suggests that the final adjustment to the last digit of Q which puts P into the interval $[0, M-1]$ is the most frequent cause of undetected errors, especially for P near a multiple of M. Such logical errors can be very infrequent. However, it is the correctness of the modular arithmetic which is the subject of this article. Such errors tend to keep recurring because the faulty hardware is either reused with the same values for every exponentiation or it is part of a digit operation which is executed a very large number of times with effectively random data. Hence they will almost certainly be detected.

Correction after transient errors is obtained simply by running the same hardware again with the identical inputs. Of course, this is useless for permanent errors. Instead, with the usual assumption that the errors are rare, rather than use alternative hardware which may contain the same design errors, the inputs can be modified in an attempt to avoid the errors.

An error with a particular digit slice might be avoided by a simple shift: $T^e \bmod M$ is computed via $T^e \bmod rM$, so that the combination of bits which cause the error might be avoided. This just requires a slight modification to the hardware or software which makes the final modular correction to bring the output of the modular multiplier into the correct range $[0, M-1]$. A bigger shift might avoid the use of the faulty digit slice entirely. Of course, this form of adjustment is not an option when Montgomery's method is used since the new modulus must stay prime to r, nor will it work if the same hardware is used for every digit position. Then, or if the problem is with the most or least significant digits of M, a similar solution of computing $T^e \bmod dM$ for a digit d prime to r may succeed. Other inputs than M to the digit operations all vary

so much and so frequently that the digit combination expressing the error will arise very frequently whatever input modifications are made. This is enough to make disposal and replacement of the chip the best solution.

8 Summary and Conclusion

The detection and correction of transient errors in a hardware implementation of the RSA cryptosystem is straightforward to implement and can be used to defeat certain types of active attack on embedded systems such as in smartcards. It can be done efficiently and reliably with acceptable time and area costs equivalent to an increase in the size of the modulus by one digit or less plus some extra buffering. Successful correction must usually assume the correctness of the hardware. However, the checker function and other outlined methods will also detect most logic errors and fabrication faults as well as transient ones. With minor extra work which could be supplied by software, these too might be corrected if they are sufficiently infrequent.

Incorporating a checker function such as (5) and keeping an eye out for overflows are increasingly essential with shrinking technology and may prevent the loss of considerable data when an error inevitably strikes.

References

1. J. M. Benedetto, "Economy-class Ion-defying ICs in Orbit", *IEEE Spectrum*, vol. 35, no. 3, March 1998, pp 36-41
2. M. Blum and H. Wasserman, "Reflections on the Pentium Bug", *IEEE Trans. Comp.*, vol. 45, no. 4, April 1996, pp 385-393
3. D. Boneh, R. DeMillo and R. Lipton, "On the importance of checking cryptographic protocols for faults", *Eurocrypt '97*, Lecture Notes in Computer Science, vol. 1233, Springer-Verlag, 1997, pp 37-51
4. E. F. Brickell, "A Fast Modular Multiplication Algorithm with Application to Two-Key Cryptography", *Advances in Cryptology - CRYPTO '82*, Chaum et al., Eds., New York, Plenum, 1983, pp 51-60
5. G. Gerwig and M. Kroener, "Floating Point Unit in standard cell design with 116 bit wide dataflow", *Proc 14th IEEE Symposium on Computer Arithmetic*, Adelaide, 14-16 April 1999, IEEE Press, 1999, pp 266-273
6. P. Kornerup, "A Systolic, Linear-Array Multiplier for a Class of Right-Shift Algorithms", *IEEE Trans. Comp.*, vol. 43, no. 8, April 1994, pp 892-898
7. P. L. Montgomery, "Modular Multiplication without Trial Division", *Math. Computation*, vol. 44, 1985, pp 519-521
8. J.-J. Quisquater and M. De Soete, "Speeding up smart card RSA computations with insecure coprocessors", *Proc. Smart Card 2000*, D. Chaum editor, Elsevier Science, 1991, pp 191-197
9. R. L. Rivest, A. Shamir and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems", *Comm. ACM*, vol. 21, 1978, pp 120-126
10. C. D. Walter, "Moduli for Testing Implementations of the RSA Cryptosystem", *Proc 14th IEEE Symposium on Computer Arithmetic*, Adelaide, 14-16 April 1999, IEEE Press, 1999, pp 78-85
11. C. D. Walter, "Systolic Modular Multiplication", *IEEE Trans. Comp.*, vol. 42, no. 3, March 1993, pp 376-378

A Design for Modular Exponentiation Coprocessor in Mobile Telecommunication Terminals

Takehiko Kato, Satoru Ito, Jun Anzai, and Natsume Matsuzaki

Advanced Mobile Telecommunications Security Technology Research Laboratories Co., Ltd.
BENEX S3 Building 12F, 3-20-8 Shinyokohama, Kohoku-ku, Yokohama, 222-0033 Japan
{tkato,anzai,matuzaki}@amsl.co.jp

Abstract. Following requirements are necessary when implementing public key cryptography in a mobile telecommunication terminal. (1) simultaneous high-speed double modular exponentiation calculation, (2) small size and low power consumption, (3) resistance to side channel attacks. We have developed a coprocessor that provides these requirements. In this coprocessor, right-to-left binary exponentiation algorithm was extended for double modular exponentiations by designing new circuit configuration and new schedule control methods. We specified the desired power consumption of the circuit at the initial design stage. Our proposed method resists side channel attacks that extract secret exponent by analyzing the target's power consumption and calculation time.

1 Introduction

The use of public key cryptography in mobile telecommunication is on the increase. Small size, lightweight and low power consumption are necessary for mobile telecommunication terminals. These devices, because they are small, are easily lost or stolen. They have a risk to be disassembled or analyzed by the third party.

Public key cryptography requires large-scale calculations, using modular exponentiation factors of up to 1024 bits. The low powered MPU used in a typical mobile telecommunication terminal takes a long time to perform these calculations. It can take several seconds to perform a modular exponentiation in software.

There are many cases when double or more modular exponentiations are required in the verification of signature based on discrete log such as DSA [1] or Nyberg-Rueppel signature [2], Cramer-Shoup scheme [3] and Anzai-Matsuzaki- Matsumoto scheme [4][5]. For other examples, RSA use a modular exponentiation, but more modular exponentiations are required to check the certificate of CA.

Recently, there have been examples of side channel attacks, which use information leaked during cryptographic processing. Circuits that are resistant to such attacks are needed. Side channel attacks include power analysis attacks, timing attacks and electromagnetic emission attacks. There are many studies of each of these symmetric cryptographs and public key cryptographs, some of which we will describe next.

Paul Kocher et al. tested timing attacks on Diffie-Hellman, RSA and DSA in [6]. They discovered that by carefully measuring the time required to perform symmetric key operation, attackers could find fixed Diffie-Hellman exponents, could find factor

Ç.K. Koç and C. Paar (Eds.): CHES 2000, LNCS 1965, pp. 216 - 228, 2000.

RSA keys, and broke other cryptography. Messerges et al. examined power analysis attack on the modular exponentiation of public key cryptography in [7]. Goubin et al. studied power analysis attacks on RSA and described countermeasures in [8]. Handschuh et al. tested probing attacks using a monitor oracle in [9]. As we have said, a lot of research is being done on side channel attack method, and coprocessor performing encryption algorithms must be resistant to these types of attacks. To achieve this goal, calculation time should be kept constant and current variation should vary as little as possible.

Therefore, we will develop a coprocessor that fulfils the following requirements:

- simultaneous high-speed double modular exponentiations,
- small size and low power consumption,
- resistance to side channel attacks.

After clearing problems of conventional circuits by basic investigations, we consider countermeasures. However, these countermeasures cannot satisfy our requirements. We propose new method in section 4.

2 Basic Investigations

As shown below, the modular exponentiation calculation $T=A^B$ mod C is performed using the square-and-multiply algorithm. Here, A is base, B is exponent and C is modulus. The left-to-right circuit (LRC) is based upon the left-to-right binary exponentiation algorithm [11] and the right-to-left circuit (RLC) is based upon the right-to-left binary exponentiation algorithm [11]. The RLC process the modular square and modular multiply in parallel.

Now we will compare RLC and LRC in terms of the three requirements mentioned above. Here a "loop" means one modular square calculation or one modular multiply calculation. In LRC, when the B is "0", only a modular square is performed and it loops once. When the B is "1", both a modular square and a modular multiply are performed and it loops twice. On the other hand, in RLC, whether the B is "0" or "1" there is only one loop because of parallel processing.

2.1 Calculation Time, Power Consumption, and Number of Gates

The power consumption of the circuit can be estimated using simulation data available at the circuit design stage. Using requirements in Table.1, we estimated the power consumption when LRC and RLC were installed in an ASIC (Fujitsu CE61).

Table 1. Requirement for Power Consumption Analysis and Simulation

Analysis tool	PROVERD/PWR (Fujitsu LSI technology)
Simulation	Verilog XL
Clock frequency	20MHz (50ns)
Measurement interval	500 ns (per every 10 clocks)

We estimated the current consumption of LRC and RLC using 8 bits B when A and C of 1024 bits each. The results are shown in Table.2. In this paper, the current

consumption is also called the power consumption. Current consumption [microsec*mA] equal average current [mA] multiplied by calculation time [microsec]. Number of gates is 28326 for LRC and 37049 for RLC each.

Table 2. Comparison between LRC and RLC on ASIC

	LRC			RLC		
B	Calculation time [microsec]	Average current [mA]	Current Consumption [microsec*mA]	Calculation time [microsec]	Average current [mA]	Current Consumption [microsec*mA]
1000 0000	2240	23.7	52994	2360	31.9	75336
1010 1010	3160	23.7	74927	2370	34.7	82147
1111 0000	3160	23.7	74977	2370	34.6	82019
1111 1111	4400	23.8	104553	2390	38.1	91032

As seen above, the current consumption of LRC and RLC is almost same. The calculation time of RLC is shorter than that of LRC. The number of gates required for of RLC is larger than that of LRC.

2.2 Resistance to Power Analysis Attacks and Timing Attacks

2.2.1 Current Waveform of RLC

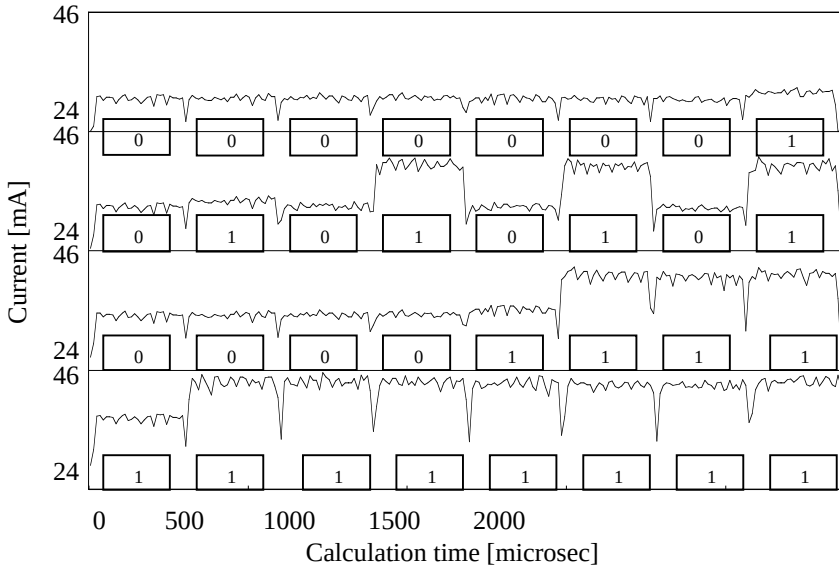


Fig. 1. Current Waveform of RLC

The current waveform of RLC was measured using a base and modulus of 1024 bits each and exponents of 8 bits. From Fig.1, we can readily see the difference in current variation when the exponent is "1" and when it is "0". The variation is high at “1” and low at “0”. By monitoring these fluctuations, we can easily determine the value of exponent. In public key cryptography systems such as RSA, ElGamal, etc., it is critical to keep the exponents secret. If RLC is used, we must provide a way to prevent power analysis attacks. In RLC, the calculation time is constant regardless of the number of "1"s in the exponent, making it resistant to timing attacks.

2.2.2 Current Waveform of LRC

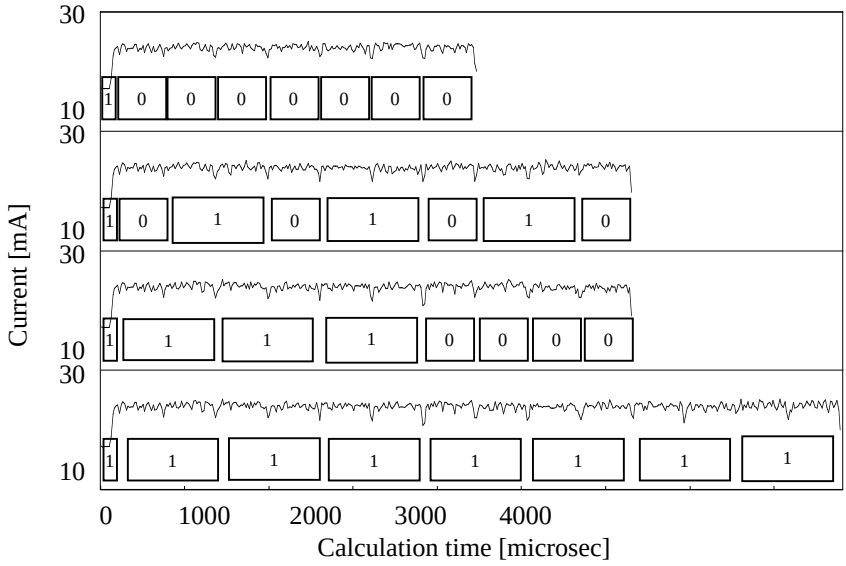


Fig. 2. Current Waveform of LRC

From Fig.2, we can see current variations corresponding to exponents are smaller. It is more difficult to determine the exponent value by monitoring the current variation. However, the calculation time variation can be more easily observed, being proportional to the Hamming weight. This means that although LRC is more resistant to power analysis attacks, it is more vulnerable to timing attacks.

In practice, signal leakage is minute, and is usually masked by noise. Integrated-and dump filters or other technologies are in use [7].

2.3 Overall Comparison between RLC and LRC

Now we will compare RLC and LRC based on the above discussion.

Table 3. Overall Comparison between RLC and LRC

	calculation time	number of gates	power consumption	timing attacks	power analysis attacks
RLC	allowed	not allowed	fairly good	difficult	possible
LRC	not allowed	allowed	fairly good	possible	difficult

RLC has the advantages of calculation time and resistance to timing attacks. On the other hand, LRC has the advantages of fewer gates and resistance to power analysis attacks.

Semiconductor manufacturing technology has great progress in miniaturization, lessening the impact of gate costs. We see calculation time as a more important factor than the number of gates. Our preference, therefore, is RLC and this is what we will subsequently discuss.

3 Countermeasures

We decided to adopt RLC because of its reduced calculation time, in spite of the risk that the modular exponent might be decoded from current waveform analysis. We will perform double modular exponentiations by running two RLC in parallel. This configuration will be called D-RLC. We also considered dual LRC (D-LRC), but these double both the number of gates required and the power consumption.

Multiple modular exponentiation methods were considered in [10], but these systems required a lot of memory, and were considered unsuitable for mobile telecommunication terminals. We studied the faster system of simultaneous double modular exponentiations. Studies are being done on efficient multiple modular exponentiation calculations in [11]. However, most of them need large memory-intensive tables making them unsuitable for mobile telecommunication terminals. For that reason, we didn't take them into consideration.

We considered countermeasure against both power analysis attacks and timing attacks. We used a dummy calculation (DC) to forcibly the RLC to always perform both a modular square circuit and a modular multiply. This DC emulates the idle circuit using previously calculated data for instance, and every circuit is constantly operated even if the exponent is "0".

Using DC, the modular exponentiation calculations in both RLC and LRC show the same current waveforms and current consumption as that of all exponents in “1”. By adjusting the calculation time to give double time for "0" bits, LRC can also be made resistant to timing attacks.

The method that varies calculation time using a blind signature is proposed in [6]. This method is effective for power analysis attacks. Goubin et al. divided plain text into multiple parts and calculated each, and then combined the results. This method is effective against power analysis attacks because it alters the pattern of electromagnetic emission. These last two methods may increase calculation time, number of gates or required MPU processing power.

We can see that for a circuit to be resistant to power analysis attacks and timing attacks, it must operate with constant current variation and calculation time regardless of input values. But this may result in excessive current consumption and calculation time which is a problem in practical use. In the next section, we will discuss a way to make a practical system with sufficient resistance to power analysis attacks and timing attacks.

Therefore, a different approach is necessary. Next, we will show our proposed method.

4 Our Proposed Method (OPM)

Our proposed method consists of a new circuit configuration and a new schedule control method.

4.1 New Circuit Configuration

Using DC, the results were the slowest calculation time or the highest current consumption. We realized that in many cases double modular exponentiation calculations performed for public key cryptography. Modular squares are always performed, but modular multiplies are performed only when the exponent is "1", never when the exponent is "0". This means that shared modular multiply units were a possibility. As shown in Fig.3, we first used two separate RLC. Then we combined the two separate modular multiply units into one for shared use. This results in fewer gates.

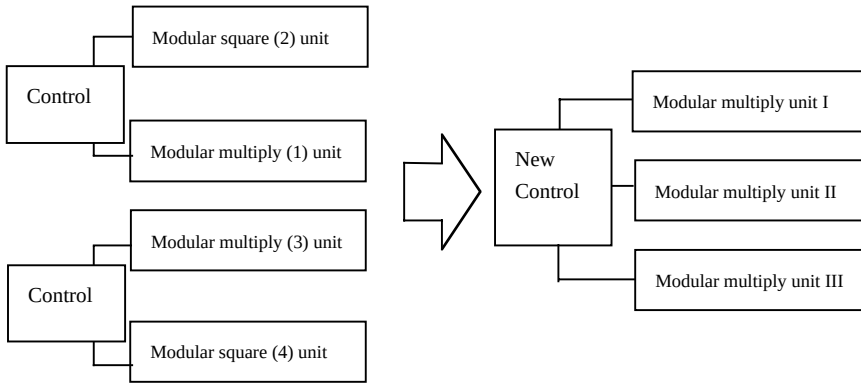


Fig. 3. New Circuit Configuration

The modular multiply unit consists of two 16 bits multipliers using the method that modular calculation per partial multiply are performed. In the new circuit configuration, the shared modular multiply unit II cannot be used for double modular exponentiation calculations when both exponents are equal to "1". In this case, one of the calculations may have to be delayed until the modular multiply unit becomes idle.

4.2 New Schedule Control Method

Here we propose a new method of control scheduling that avoids the delay problem mentioned above.

Double modular exponentiations are divided into two modular squares and two modular multiplies for each i-th bit of exponent B. These four instructions enter the three modular multiply units (I, II and III) in order. But some exception handling is necessary. In the control part, the instructions correspond to exponents are stored in a register FIFO. During the calculation phase, the register FIFO is monitored, and the instructions are executed. Fig.4 shows the process flow.

The control and calculation parts are performed in parallel. In the control part, four instructions ($(1)_i$-$(4)_i$) correspond to exponent i-th bit of B entered four control register (FIFO(0)-FIFO(3)). In the calculation part, three modular multiply units are performed. The FIFO with the under-bar contains the most significant bit calculation, and the asterisk means either calculation.

Two modular squares and two modular multiplies are performed in three modular multiply units for each i-th bit of B in Fig.4. The expression $(1)_i$ or $(3)_i$ shows a modular multiply and the $(2)_i$ or $(4)_i$ shows a modular square in Fig.3. The one bit calculation is carried out in one or two loops. One loop consists of one operation of the three modular multiply units. The shadow part corresponds to the input exponent. The oblique line is uncalculated part. The system disallows:

- a modular square or modular multiply of a different exponent in same loop (ex.$(2)_i$ and $(2)_{i+1}$ in the same loop),
- a modular square and modular multiply of different i-th bit of same exponent in same loop (ex.$(4)_i$ and $(3)_{i+1}$ in the same loop)

We call this prohibition law.

Examples of new schedule control method are shown in Fig.5.

Fig.5 shows the how modular multiply units I and III are able to simultaneously operate when both exponents B1 and B2 are "1". In some cases, however, modular multiply unit III is blocked (see aforementioned prohibition law). We considered avoiding this prohibition law by changing the order of the calculation. In right-to-left binary exponentiation algorithm, the 1 bit result is used in the next calculation. In this case, the modular multiply requires the results of both previous modular multiply and modular square. But the modular square needs only the results from the previous modular square. By preprocessing the modular square and storing the results in two 1024 bits buffers, we can solve the problem. Fig.6 shows to avoid part 2 of the prohibition law (i.e.,$(4)_i$ and $(3)_{i+1}$ in the same loop). As we can see in the 2nd part of Fig.5, modular multiply unit III cannot process the 4th loop $(4)_3$. But we can preprocess $(4)_2$ and replace $(3)_2$ in the 2nd loop(see Fig.6). Then we can process $(4)_3$ in the 3rd loop. The uncalculated part of Fig.5 is replaced by preprocessing.

```
//Length establishment of exponents
b1msb = MSB(B1); b2msb = MSB(B2);
last = MAX(b1msb, b2msb);
//Control part
for (i=0 ; i <= last ; i++){
      if (B1(i) == 0 && B2(i) == 0){
                        FIFO(0,1) = ((2),  (4)); }
      if (B1(i) == 1 && B2(i) == 0){
            if (i == last){ FIFO(0)    = (_(1)       ); }
            else   { FIFO(0,1,2) = (  (1), (2), (4)); }}
      if (B1(i) == 0 && B2(i) == 1){
            if (i == last){ FIFO(0)    = (   _(3)   ); }
            else       { FIFO(0,1,2) = ((2), (3), (4)); }}
      if (B1(i) == 1 && B2(i) == 1){
            if (i == last){ FIFO(0,1)    = ((1),   _(3)   ); }
            else   { FIFO(0,1,2,3) = ((1), (2), (3), (4)); }}

//Calculation part

while(1){
      if ( FIFO(0,1) == ((2), (4)) || FIFO(0,1) == ((4), (2)) ||
                              FIFO(0,1,2) == ((4), (1), (3)) ||FIFO(1) == (_*) ){
                  Modular_multiply_unit_I(FIFO(0));
                  Modular_multiply_unit_II(FIFO(1));
      }else if ( FIFO(0,1) == ((2), (1)) || FIFO(0,1) == ((4),  (3)) || FIFO(0) == (_*) ){
                  Modular_multiply_unit_I(FIFO(0));
      }else{
                  Modular_multiply_unit_I(FIFO(0));
                  Modular_multiply_unit_II(FIFO(1));
                  Modular_multiply_unit_III(FIFO(2));}
      if ( FIFO(0) == _* || FIFO(1) == _* || FIFO(2) == _* )
                  break; }}
```

Fig. 4. Calculation Process Flow of New Schedule Control Method

Modular multiply unit	Input exponent B1 =	1	0	0	0	0	0	0	0	
	Input exponent B2 =	1	1	1	1	1	1	1	1	
	Start						End			
I	$(2)_1$	$(2)_2$	$(2)_3$	$(2)_4$	$(2)_5$	$(2)_6$	$(2)_7$	$\bar{}(1)_8$		
II	$(3)_1$	$(3)_2$	$(3)_3$	$(3)_4$	$(3)_5$	$(3)_6$	$(3)_7$	$\bar{}(3)_8$		
III	$(4)_1$	$(4)_2$	$(4)_3$	$(4)_4$	$(4)_5$	$(4)_6$	$(4)_7$			

Modular multiply unit	Input exponent B1 =	1	0	1	0	1	0	1	0	
	Input exponent B2 =	1	1	1	1	1	1	1	1	
I	$(2)_1$	$(1)_2$	$(4)_2$	$(3)_3$	$(2)_4$	$(2)_5$	$(1)_6$	$(4)_6$	$(3)_7$	$\bar{}(3)_8$
II	$(3)_1$	$(2)_2$	$(2)_3$	$(4)_3$	$(3)_4$	$(3)_5$	$(2)_6$	$(2)_7$	$(4)_7$	
III	$(4)_1$	$(3)_2$		$(1)_4$	$(4)_4$	$(4)_5$	$(3)_6$		$\bar{}(1)_8$	

Modular multiply unit	Input exponent B1 =	1	1	1	1	0	0	0	0	
	Input exponent B2 =	1	1	1	1	1	1	1	1	
I	$(2)_1$	$(2)_2$	$(2)_3$	$(2)_4$	$(1)_5$	$(4)_5$	$(3)_6$	$(2)_7$	$\bar{}(1)_8$	
II	$(3)_1$	$(3)_2$	$(3)_3$	$(3)_4$	$(2)_5$	$(1)_6$	$(4)_6$	$(3)_7$	$\bar{}(3)_8$	
III	$(4)_1$	$(4)_2$	$(4)_3$	$(4)_4$	$(3)_5$	$(2)_6$	$(1)_7$	$(4)_7$		

Modular multiply unit	Input exponent B1 =	1	1	1	1	1	1	1	1	
	Input exponent B2 =	1	1	1	1	1	1	1	1	
I	$(1)_1$	$(4)_1$	$(3)_2$	$(2)_3$	$(1)_4$	$(4)_4$	$(3)_5$	$(2)_6$	$(1)_7$	$(4)_7$	$\bar{}(3)_8$
II	$(2)_1$	$(1)_2$	$(4)_2$	$(3)_3$	$(2)_4$	$(1)_5$	$(4)_5$	$(3)_6$	$(2)_7$	$\bar{}(1)_8$	
III	$(3)_1$	$(2)_2$	$(1)_3$	$(4)_3$	$(3)_4$	$(2)_5$	$(1)_6$	$(4)_6$	$(3)_7$		

Fig. 5. Examples of New Schedule Control Method

Modular multiply unit	Input exponent B2 =	1	0	1	0	1	0	1	0	
	Input exponent B3 =	1	1	1	1	1	1	1	1	
I	$(2)_1$	$(1)_2$	$(3)_2$	$(3)_3$	$(3)_4$	$(3)_5$	$(2)_6$	$(2)_7$	$\_(1)_8$	
II	$(3)_1$	$(2)_2$	$(2)_3$	$(1)_4$	$(4)_4$	$(4)_5$	$(3)_6$	$(3)_7$	$\_(3)_8$	
III	$(4)_1$	$(4)_2$	$(4)_3$	$(2)_4$	$(2)_5$	$(1)_6$	$(4)_6$	$(4)_7$		

Fig. 6. Schedule Replacing Example by Preprocessing

5 Evaluation

We evaluated the method that prevent discovery of modular exponents by third party monitoring of the current consumption patterns and calculation time of the circuitry.

OPM is resistant to timing attacks and power analysis attacks because:

- one bit processing of exponent is spread over one or two loops performed by the modular multiply units,
- various kind of exponents are mixed in the same loop,
- if the exponents are reverse (B1, B2 are reversed B2, B1), the current waveform is changed corresponding to the processing.

The number of loop changes not only based on the combination of the i-th "0" and "1", but also the (i-1)-th combination or the (i+1)-th combination. The calculation time required by OPM does not increase proportionally to the Hamming weight as it does in LRC. The larger the combination of the i-th exponents B1 and B2 per loop is, the larger the safety margin will be. In OPM, even in one loop, there are two possibilities to determine that only i-th bit is performed and (i+1)-th bits are performed. For this reason, it is not possible to determine whether the combination of exponents is "0 and 1", "1 and 0", "1 and 1" or "0 and 0". Fig.7 shows the waveform of OPM corresponding to Fig.5. OPM is resistant to power analysis attacks and timing attacks in practical use. Fig.5 and Fig.7 demonstrate our hypothesis.

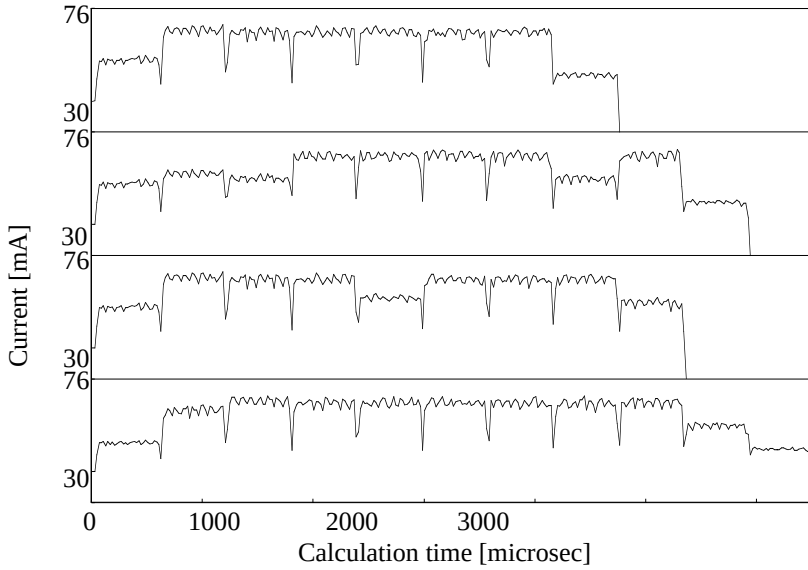


Fig. 7. Current Waveform of OPM

OPM resists to side channel attacks in practical use. However, further enhancement is possible. We add DC (see section 4) when double modular exponentiation calculations are performed. This DC forces the operation of all three modular multiply units. If a calculation requires the use of only two units, DC is performed in the unused unit. Fig.5 shows the DC via oblique lines.

Fig.8 shows the current waveform resulting from three types of double modular exponentiations. We compared OPM+DC, D-RLC+DC and D-LRC+DC. It is difficult to distinguish between "0" and "1". Three methods show the same current waveform each for every exponent.

Table 4 shows the results of three methods where the base and modulus are 1024 bits each and the exponents are 8 bits. The values of OPM were obtained from the average of seven patterns for each B1 and B2, 8080, 80ff, f0f0, aaaa, f0ff, aaff, ffff (hexadecimal digit). Followings are indicated from Table.4:

- OPM shows the best current consumption of double exponentiations,
- OPM shows fairly good characteristics of number of gates and calculation time compares best other method

For OPM, we anticipated an increase in the number of gates required since gate requirements are proportional to the number of modular multipliers (we use three modular multiply units). But OPM needed only about 10% more gates due to the total circuit scale expansion from the addition of control circuits, etc. when compared with the total circuit scale.

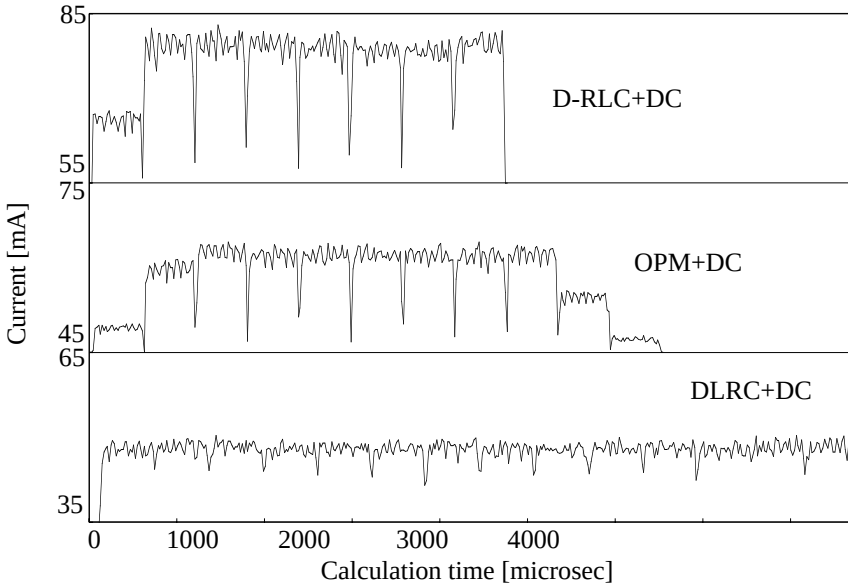


Fig. 8. Current Waveform of D-LRC+DC, D-RLC+DC and OPM+DC

Although the average current consumption is higher and more gates are required, OPM has the advantages of reduced calculation time and lower power consumption.

The coprocessor using OPM featuring high speed, low power consumption, small size and resistance to power analysis attacks and timing attacks are ideal for mobile telecommunication terminals.

Table 4. Overall Comparison between OPM, D-RLC+DC and D-LRC+DC

	Number of gate [gates]	Average calculation time [microsec]	Current consumption of double exponentiations [microsec*mA]
OPM	62367 (1.1)	2723 (0.62)	154375 (0.74)
D-RLC+DC	74164 (1.31)	2390 (0.54)	182065 (0.87)
D-LRC+DC	56688 (1)	4400 (1)	209106 (1)

*(….) shows relative values

By replacing from modular multiply to add on elliptic curve, the concept of OPM could be used in elliptic curve cryptosystems.

6 Conclusion

Our coprocessor design features the following characteristics:

- simultaneous double modular exponentiations performed at high speed within practical time
- small size and low power consumption
- resistance to side channel attacks

This coprocessor provides all of these well-balanced characteristics, making it ideal for mobile telecommunication terminals.

References

1) "Digital Signature Standards", Federal Information Processing Standard publication X, 1993 February 1
2) K.Nyberg, A.Rueppel, "Message Recovery for Signature Schemes Based on the Discrete Logarithm Problem", Advanced in Cryptology-EUROCRYPT' 94, Springer-Verlag.
3) R.Cramer, V.Shoup, "A Practical Public Key Cryptosystem Provably Secure against Adaptive Chosen Ciphertext Attack", Lecture Note in Computer Science. Advanced in Cryptology-CRYPTO'98, Springer-Verlag, pp.13-25.
4) J.Anzai, N.Matsuzaki, T.Matsumoto, "A Quick Group Key Distribution Scheme with Entity Revocation", Advanced in Cryptology-ASIACRYPTO'99, Springer-Verlag, pp.333-347.
5) N.Matsuzaki, J.Anzai, T.Matsumoto, "Light Weight Broadcast Exclusion using Secret Sharing", Fifth Australasian Conference on Information Security and Privacy, Springer-Verlag, pp. 313-327.
6) P.C.Kocher, "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS and Other Systems", Advanced in Cryptology-CRYPTO'96, Springer-Verlag, pp.104-113.
7) T.S.Messerges, E.A.Dabbish, R.H.Sloan, "Power Analysis Attacks of Modular Exponentiation in Smartcards", Cryptographic Hardware and Embedded Systems-CHES'99, Springer-Verlag, 1999, pp.144-157.
8) L.Goubin, J.Patarin, "DES and Differential Power Analysis : The Duplication Method", Cryptographic Hardware and Embedded Systems-CHES'99, Springer-Verlag, 1999, pp.158-172.
9) H.Handschuh, P.Paillier, J.Stern, "Probing Attacks on Tamper-Resistant Devices", Cryptographic Hardware and Embedded Systems-CHES'99, Springer-Verlag, 1999, pp.303-315.
10) A.Andreasyan, G.Khachatrian, "New Double Exponentiation Algorithms", Third International Workshop on practice and Theory in Public Key Cryptography PKC2000, The Poster Papers Collection p.9-15, ISBN 0-73262-130-5, Monash Univ.
11) A.J.Menezes, P.C.Oorchot, S.A.Vanstone, "HANDBOOK of APPLIED CRYPTOGRAPHY", CRC press, pp.614-615, pp.620-627.

How to Explain Side-Channel Leakage to Your Kids

David Naccache[1] and Michael Tunstall[2]

[1] Gemplus, 34 rue Guynemer
Issy-les-Moulineaux, F-92447, France
david.naccache@gemplus.com
[2] Gemplus, Card Security Group, B.P. 100
Gemenos, F-13881, France
michael.tunstall@gemplus.com

Abstract. This paper will attempt to explain some of the side-channel attack techniques in a fashion that is easily comprehensible by the layman.
What follows is a presentation of three different attacks (power, timing and fault attacks) that can be carried out on cryptographic devices such as smart-cards.
For each of the three attacks covered, a puzzle and it's solution will be given, which will act as an analogy to the attack.
How these attacks can be applied to real devices will also be discussed.

1 Timing Attacks

When an algorithm is executed on a device it will take a certain amount of time to complete. In some instances the amount of time the algorithm takes to execute will vary depending on the secret information that is normally not available to an external observer. An animated PowerPoint slide-show (game) and it's winning strategy give an example of how this technique can be used.

The story was originally told by Eli Biham at the dinner that followed the Ph.D. defenses of Helena Handschuh and Pascal Paillier.

2 Power Attacks

A cryptographic device will consume a varying amount of current as it executes an algorithm. By making observations one can attempt to deduce information about what is occuring.

The following is a situation where this technique can be applied: A paparazzi is investigating the lives of a Royal couple. He follows then to a restaurant and then to their home. He is under the impression that they have had an argument, but as the two are public figures they will not permit themselves to argue in public.

Ç.K. Koç and C. Paar (Eds.): CHES 2000, LNCS 1965, pp. 229–230, 2000.

To simplify the situation we will make the assumption that their home (castle?) consists of two rooms each with one lightbulb and no other electronic equipment. There are not any windows or convenient keyholes either and the reporter wishes to find out whether or not the two are still talking to each other.

As suggested at the beginning of this section the solution revolves around the amount of current consumed by the two lightbulbs. The reporter needs to find access to the electricity meter (which in our scenario is outside the Royal property). By looking at the speed that the disk inside the meter is rotating the reporter is able to determine whether one or two lights are turned on.

3 Fault Generation

Finally, as an algorithm is being executed by a device it is possible to physically attack the device to change the output of the algorithm, a potentially strong attack against cryptographic devices. It is also possible to attack the device in a manner that will change its behavior, creating other opportunities to attack the device. This as well will be illustrated using an animated PowerPoint slide-show.

On Boolean and Arithmetic Masking against Differential Power Analysis

Jean-Sébastien Coron[1] and Louis Goubin[2]

[1] Gemplus Card International, 34 rue Guynemer
Issy-les-Moulineaux, F-92447, France
jean-sebastien.coron@gemplus.com
[2] Bull SmartCards and Terminals
68 route de Versailles - BP45
78431 Louveciennes Cedex - France
Louis.Goubin@bull.net

Abstract. Since the announcement of the Differential Power Analysis (DPA) by Paul Kocher and al., several countermeasures were proposed in order to protect software implementations of cryptographic algorithms. In an attempt to reduce the resulting memory and execution time overhead, Thomas Messerges recently proposed a general method that "masks" all the intermediate data.
This masking strategy is possible if all the fundamental operations used in a given algorithm can be rewritten with masked input data, giving masked output data. This is easily seen to be the case in classical algorithms such as DES or RSA.
However, for algorithms that combine Boolean and arithmetic functions, such as IDEA or several of the AES candidates, two different kinds of masking have to be used. There is thus a need for a method to convert back and forth between Boolean masking and arithmetic masking.
In the present paper, we show that the 'BooleanToArithmetic' algorithm proposed by T. Messerges is not sufficient to prevent Differential Power Analysis. In a similar way, the 'ArithmeticToBoolean' algorithm is not secure either.
Keywords: Physical attacks, Differential Power Analysis, Electric consumption, AES, IDEA, Smartcards, Masking Techniques.

1 Introduction

Paul Kocher and al. introduced in 1998 ([10]) and published in 1999 ([11]) the concept of *Different Power Analysis* attack, also known as DPA. It belongs to a general family of attacks that look for information about the secret key of a cryptographic algorithm, by studying the electric consumption of the electronic device during the execution of the computation.

The initial focus was on symmetrical cryptosystems such as DES (see [10,14]) and the AES candidates (see [1,3,6]), but public-key cryptosystems have since been shown to be also vulnerable to the DPA attacks (see [15,5,9]).

Therefore, the research for countermeasures has considerably increased. In [6], Daemen and Rijmen proposed several countermeasures, including the insertion of dummy code, power consumption randomization and balancing of data.

Ç.K. Koç and C. Paar (Eds.): CHES 2000, LNCS 1965, pp. 231–237, 2000.

But these methods were proven to be insufficient: in [4], Chari and al. suggested that signal processing can be used by clever attackers to remove dummy code or to cancel the effects of randomization and data balancing. They propose a better approach, consisting in splitting all the intermediate variables. A similar "duplication" method was proposed as a particular case by Goubin and al. in [9]

However, these general methods generally increase dramatically the amount of memory needed, or the computation time, as was pointed by Chari and al. in [3]. Moreover, it has been shown in [8] that even inner rounds can be aimed by "Power-Analysis"-type attacks, so that the splitting should be performed on all rounds of the algorithm. This makes the issue of the memory and time computation overhead even more crucial, especially for embedded systems such as smart cards.

In [13], Thomas Messerges investigated on DPA attacks applied on the AES candidates. He developped a general countermeasure, consisting in masking all the inputs and outputs of each elementary operations used by the microprocessor. This generic technique allowed him to evaluate the impact of these countermeasures on the five AES algorithms.

This masking strategy is possible if all the fundamental operations used in a given algorithm can be rewritten with masked input data, giving masked output data. This is easily seen to be the case for the DES algorithm, because a single masking (using the XOR operation) can be used throughout the computation of the 16 rounds. For RSA, a masking using the multiplication operation in the multiplicative group modulo n is also sufficient.

However, for algorithms that combine Boolean and arithmetic functions, two different kinds of masking have to be used. There is thus a need for a method to convert back and forth between Boolean masking and arithmetic masking. This is typically the case for IDEA [12] and for three AES candidates: MARS [2], RC6 [16] and TWOFISH [17].

Thomas Messerges proposed in [13] an algorithm in order to perform this conversion between a "$\oplus$ mask" and a "$+$ mask". Unfortunately, we show in the present paper that the 'BooleanToArithmetic' algorithm proposed by T. Messerges is not sufficient to prevent Differential Power Analysis. In a similar way, the 'ArithmeticToBoolean' algorithm is not secure either. A detailed attack is described.

2 The "Differential Power Analysis" Attack

The "Differential Power Analysis" attack, developped by Paul Kocher and Cryptographic Research (see [10,11], see also [7]), starts from the fact that the attacker can get much more information (than the knowledge of the inputs and the outputs) during the execution of the computation, such as for instance the electric consumption of the microcontroller or the electromagnetic radiations of the circuit.

The "Differential Power Analysis" (DPA) is an attack that allows to obtain information about the secret key (contained in a smartcard for example), by performing a statistical analysis of the electric consumption records measured for a large number of computations with the same key.

Let us consider for instance the case of the DES algorithm (Data Encryption Standard). It executes in 16 steps, called "rounds". In each of these steps, a transformation F is performed on 32 bits. This F function uses eight non-linear transformations from 6 bits to 4 bits, each of which is coded by a table called "S-box".

The DPA attack on the DES can be performed as follows (the number 1000 used below is just an example):

Step 1: We measure the consumption on the first round, for 1000 DES computations. We denote by E_1, ..., E_{1000} the input values of those 1000 computations. We denote by C_1, ..., C_{1000} the 1000 electric consumption curves measured during the computations. We also compute the "mean curve" MC of those 1000 consumption curves.

Step 2: We focus for instance on the first output bit of the first S-box during the first round. Let b be the value of that bit. It is easy to see that b depends on only 6 bits of the secret key. The attacker makes an hypothesis on the involved 6 bits. He computes – from those 6 bits and from the E_i – the expected (theoretical) values for b. This enables to separate the 1000 inputs E_1, ..., E_{1000} into two categories: those giving $b = 0$ and those giving $b = 1$.

Step 3: We now compute the mean MC' of the curves corresponding to inputs of the first category (*i.e.* the one for which $b = 0$). If MC and MC' show an appreciable difference (in a statistical meaning, *i.e.* a difference much greater than the standard deviation of the measured noise), we consider that the chosen values for the 6 key bits were correct. If MC and MC' do not show any sensible difference, we repeat step 2 with another choice for the 6 bits.

Note: In practice, for each choice of the 6 key bits, we draw the curve representing the difference between MC and MC'. As a result, we obtain 64 curves, among which one is supposed to be very special, *i.e.* to show an appreciable difference, compared to all the others.

Step 4: We repeat steps 2 and 3 with a "target" bit b in the second S-box, then in the third S-box, ..., until the eighth S-box. As a result, we finally obtain 48 bits of the secret key.

Step 5: The remaining 8 bits can be found by exhaustive search.

Note: It is also possible to focus (in steps 2, 3 and 4) on the set of the four output bits for the considered S-boxes, instead of only one output bit. This is what we actually did for real smartcards. In that case, the inputs are separated into 16 categories: those giving 0000 as output, those giving 0001, ..., those giving 1111. In step 3, we may compute for example the mean MC' of the curves corresponding to the last category (*i.e.* the one which gives 1111 as output). As a result, the mean MC' is computed on approximately $\frac{1}{16}$ of the curves (instead of approximately half of the curves with step 3 above): this may compel us to use a number of DES computations greater than 1000, but it generally leads to a more appreciable difference between MC and MC'.

This attack does not require any knowledge about the individual electric consumption of each instruction, nor about the position in time of each of these instructions. It applies exactly the same way as soon as the attacker knows the outputs of the algorithm and the corresponding consumption curves. It only relies on the following fundamental hypothesis:

Fundamental Hypothesis*: There exists an intermediate variable, that appears during the computation of the algorithm, such that knowing a few key bits (in practice less than 32 bits) allows us to decide whether two inputs (respectively two outputs) give or not the same value for this variable.*

3 Review of Countermeasures

Several countermeasures against DPA attacks can be conceived. For instance:

1. Introducing random timing shifts, so that the computed means do not correspond any longer to the consumption of the same instruction. The crucial point consists here in performing those shifts so that they cannot be easily eliminated by a statistical treatment of the consumption curves.
2. Replacing some of the critical instructions (in particular the basic assembler instructions involving writings in the carry, readings of data from an array, etc) by assembler instructions whose "consumption signature" is difficult to analyze.
3. For a given algorithm, giving an explicit way of computing it, so that DPA is provably unefficient on the obtained implementation. The masking strategy, detailed below is an example of this third kind of method.

4 The Masking Method

In the present paper, we focus on the "masking method", initially suggested by Chari and al. in [3], and studied further in [4].

The basic principle consists in programming the algorithm so that the fundamental hypothesis above is not true any longer (*i.e.* an intermediate variable never depends on the knowledge of an easily accessible subset of the secret key). In a concrete way, using a secret sharing scheme, each intermediate that appears in the cryptographic algorithm is splitted. Therefore, an attacker has to analyze multiple point distributions, which makes his task grow exponentially in the number of elements in the splitting.

In [13], Messerges applied this fundamental idea for all the elementary operations that can occur in the AES algorithms. For algorithms that combine Boolean and arithmetic functions, such as MARS, RC6 and TWOFISH, two different kinds of masking have to be used:

Boolean masking: $x' = x \oplus r$
Arithmetic masking: $x' = (x - r) \bmod 2^k$

Here the variable x is masked with random r to give the masked value x'.

The conversion from boolean masking to arithmetic masking as described in [13] works as follows:

BooleanToArithmetic
Input: (x', r) such that $x = x' \oplus r$.
Output: (A, r) such that $x = A + r$
Randomly select: $C = 0$ or $C = -1$
$B = C \oplus r$; /* $B = r$ or $b = \bar{r}$ /*
$A = B \oplus x'$; /* $A = x$ or $A = \bar{x}$ /*
$A = A - B$; /* $A = x - r$ or $A = \bar{x} - \bar{r}$ /*
$A = A + C$; /* $A = x - r$ or $A = \bar{x} - \bar{r} - 1$ /*
$A = A \oplus C$; /* $A = x - r$ /*
Return(A, r);

The conversion from the arithmetic masking to the boolean masking can be done with a similar algorithm.

The conversion from one type of masking to another should be done in such a way that it is not vulnerable to DPA attacks. The previous algorithm takes as input the couple (x', r) such that $x = x' \oplus r$. The unmasked data is x and the masked data is x'. The algorithm works by unmasking x' using the XOR operation and then remasking it using the addition operation.

The issue is that the variable x or $\bar{x}$ is computed during the execution of the algorithm. It is stated in [13] that a DPA attack will not work against this algorithm because the attacker does not know whether x or $\bar{x}$ is processed. This is true for a DPA selecting one bit of x: since x and $\bar{x}$ are processed with equal probability, the processed bit is decorrelated from the key and the single-bit DPA does not work. This is not the case if we perform a DPA with 2 selected bits, as shown in the next section.

5 A DPA Attack against the Conversion Algorithm

The attack is based on the fact that if 2 bits of x are equal, the corresponding bits are also equal in $\bar{x}$. Consequently, we modify the DPA attack described in section 2. Instead of selecting the curves from the predicted value of a given bit of x, we consider 2 bits and divide the power samples into 2 groups: in the first group, the 2 bits are equal, and in the second group they are distinct. The classification is not affected by the processing of x and $\bar{x}$. Consequently, if the power consumption when 2 bits are equal differs from the power consumption when 2 bits are distinct, the 2-bits DPA works: the proper key hypothesis should show a peak, while the others will be mostly flat, so that the all the key bits will be recovered.

Consider the four conditional laws for the power consumption and denote their respective mean values $\mu_{00}, \mu_{01}, \mu_{10}, \mu_{11}$. For the proper key hypothesis, the mean value of the first group is:

$$\frac{\mu_{00} + \mu_{11}}{2}$$

and the mean value of the second group is:

$$\frac{\mu_{01} + \mu_{10}}{2}$$

The mean of the difference between the two groups is thus:

$$D = \frac{\mu_{00} + \mu_{11} - \mu_{01} - \mu_{10}}{2} \quad (1)$$

Consequently, the 2-bits DPA works if $D \neq 0$.

We would like to stress that our attack is not a high-order DPA. A high-order DPA [11] consists in looking at joint probability distributions of multiple points in the power signal. As shown in [4], a high-order DPA attack requires a number of experiments exponential in the number of points considered. Instead, our attack concentrates on a single point in the power signal. Consequently, the number of required experiments should be of the same order as for a single-bit DPA.

6 Conclusion

We have described a DPA attack against the conversion algorithm proposed in [13]. Our attack is a straighforward extension of the classical DPA attack. We did not have time to perform the experiments to validate our attack in practice but we think that the threat is real and such algorithm for converting from boolean masking to arithmetic masking should be avoided.

A natural research direction is to find an efficient algorithm for converting from boolean masking to arithmetic masking and conversely, in which all intermediate variables are decorrelated from the data to be masked, so that it is secure against DPA.

Acknowledgements. We would like to thank the anonymous referees for their helpful comments.

References

1. Eli Biham and Adi Shamir, "Power Analysis of the Key Scheduling of the AES Candidates", in *Proceedings of the Second Advanced Encryption Standard (AES) Candidate Conference*, March 1999. `http://csrc.nist.gov/encryption/aes/round1/Conf2/aes2conf.htm`.
2. C. Burwick, D. Coppersmith, E. D'Avignon, R. Gennaro, S. Halevi, C. Jutla, S.M. Matyas, L. O'Connor, M. Peyravian, D. Safford, and N. Zunic, "MARS - A Candidate Cipher for AES", NIST AES Proposal, Jun 98.
3. Suresh Chari, Charantjit S. Jutla, Josyula R. Rao and Pankaj Rohatgi, "A Cautionary Note Regarding Evaluation of AES Candidates on Smart-Cards", in *Proceedings of the Second Advanced Encryption Standard (AES) Candidate Conference*, `http://csrc.nist.gov/encryption/aes/round1/Conf2/aes2conf.htm`, March 1999.
4. Suresh Chari, Charantjit S. Jutla, Josyula R. Rao and Pankaj Rohatgi, "Towards Sound Approaches to Counteract Power-Analysis Attacks", in *Proceedings of Advances in Cryptology – CRYPTO'99*, Springer-Verlag, 1999, pp. 398-412.

5. Jean-Sébastien Coron, "Resistance Against Differential Power Analysis for Elliptic Curve Cryptosystems", in *Proceedings of Workshop on Cryptographic Hardware and Embedded Systems*, Springer-Verlag, August 1999, pp. 292-302.
6. John Daemen and Vincent Rijmen, "Resistance Against Implementation Attacks: A Comparative Study of the AES Proposals", in *Proceedings of the Second Advanced Encryption Standard (AES) Candidate Conference*, `http://csrc.nist.gov/encryption/aes/round1/Conf2/aes2conf.htm`, March 1999.
7. John Daemen, Michael Peters and Gilles Van Assche, "Bitslice Ciphers and Power Analysis Attacks", in *Proceedings of Fast Software Encryption Workshop 2000*, Springer-Verlag, April 2000.
8. Paul N. Fahn and Peter K. Pearson, "IPA: A New Class of Power Attacks", in *Proceedings of Workshop on Cryptographic Hardware and Embedded Systems*, Springer-Verlag, August 1999, pp. 173-186.
9. Louis Goubin and Jacques Patarin, "DES and Differential Power Analysis – The Duplication Method", in *Proceedings of Workshop on Cryptographic Hardware and Embedded Systems*, Springer-Verlag, August 1999, pp. 158-172.
10. Paul Kocher, Joshua Jaffe and Benjamin Jun, "Introduction to Differential Power Analysis and Related Attacks", `http://www.cryptography.com/dpa/technical`, 1998.
11. Paul Kocher, Joshua Jaffe and Benjamin Jun, "Differential Power Analysis", in *Proceedings of Advances in Cryptology – CRYPTO'99*, Springer-Verlag, 1999, pp. 388-397.
12. X. Lai and J. Massey, "A Proposal for a New Block Encryption Standard", in *Advances in Cryptology - EUROCRYPT '90 Proceedings*, Springer-Verlag, 1991, pp. 389-404.
13. Thomas S. Messerges, "Securing the AES Finalists Against Power Analysis Attacks", in *Proceedings of Fast Software Encryption Workshop 2000*, Springer-Verlag, April 2000.
14. Thomas S. Messerges, Ezzy A. Dabbish and Robert H. Sloan, "Investigations of Power Analysis Attacks on Smartcards", in *Proceedings of USENIX Workshop on Smartcard Technology*, May 1999, pp. 151-161.
15. Thomas S. Messerges, Ezzy A. Dabbish and Robert H. Sloan, "Power Analysis Attacks of Modular Exponentiation in Smartcards", in *Proceedings of Workshop on Cryptographic Hardware and Embedded Systems*, Springer-Verlag, August 1999, pp. 144-157.
16. R.L. Rivest, M.J.B. Robshaw, R. Sidney and Y.L. Yin, "The RC6 Block Cipher", v1.1, August 20, 1998.
17. B. Schneier, J. Kemsey, D. Whiting, D. Wagner, C. Hall and N. Ferguson, "Twofish: A 128-Bit Block Cipher", AES submission available at: `http://www.nist.gov/aes`.

Using Second-Order Power Analysis to Attack DPA Resistant Software

Thomas S. Messerges

Motorola Labs, Motorola
1301 E. Algonquin Road, Room 2712, Schaumburg, IL 60196
Tom.Messerges@motorola.com

Abstract. Under a simple power leakage model based on Hamming weight, a software implementation of a data-whitening routine is shown to be vulnerable to a first-order Differential Power Analysis (DPA) attack. This routine is modified to resist the first-order DPA attack, but is subsequently shown to be vulnerable to a second-order DPA attack. A second-order DPA attack that is optimal under certain assumptions is also proposed. Experimental results in an ST16 smartcard confirm the practicality of the first and second-order DPA attacks.

1 Introduction

Recently there has been increased concern over the vulnerabilities of cryptographic algorithms to leakage attacks [1]. These attacks exploit the fact that a hardware device can sometimes leak information when running a cryptographic algorithm. One source of leaked information is the time-varying power consumption of a device executing a cryptographic algorithm. In 1998, Kocher et al. introduced a leakage attack that uses a technique called Differential Power Analysis (DPA) [2]. Attacks using DPA have been shown to be quite successful at breaking the security of smartcards [3]. Researchers have reported power analysis attack against many algorithms, [e.g., 4-7] and have also developed countermeasures that can resist such attacks [e.g., 8-9].

The main focus of past research has been on first-order DPA attacks. However, higher-order DPA attacks [2] also need to be understood. For example, countermeasures that prevent first-order DPA attacks may not be effective against higher-order attacks. In my investigations, I assume that power leakage can be described by a simple model based on Hamming weights. I use this model to show that a naive implementation of a data-whitening routine is vulnerable to a first-order DPA attack. I then implement a countermeasure to protect this routine from attack, but this new routine is subsequently shown to be vulnerable to a second-order DPA attack. Finally, I show that this second-order DPA attack is approximately optimal under certain reasonable assumptions. Experimental results in an ST16 smartcard manufactured by ST Microelectronics are used to confirm the practicality of my attacks.

1.1 Definitions

A higher-order DPA attack is defined by Kocher et al. [2] as a DPA attack that combines one or more samples within a single power trace. During a first-order DPA attack, the

Ç.K. Koç and C.Paar (Eds.): CHES 2000, LNCS 1965, pp. 238 - 251, 2000.

attacker monitors power consumption signals and calculates the individual statistical properties of the signals at each sample time. In a higher-order DPA attack, the attacker calculates joint statistical properties of the power consumption at multiple sample times within the power signals. For the purpose of this paper, the definition of an *n*th-order DPA attack is given as follows.

Definition 1. An ***nth-order DPA attack*** *makes use of n different samples in the power consumption signal that correspond to n different intermediate values calculated during the execution of an algorithm.*

The attacks described in this paper are proven to be *sound*. The definition of a sound DPA attack is given as follows.

Definition 2. A DPA attack against an algorithm's secret key is ***sound*** *when it is theoretically possible for an attacker to use power consumption information to learn the value of all the bits of the secret key.*

In general, a sound attack may or may not be practical. Evaluating the practicality of a sound attack will usually require direct experimentation or a thorough simulation of a specific implementation. In this paper, I will confirm the soundness of two attacks, a first-order DPA attack and a second-order DPA attack. I will then examine the practicality of these attacks using an ST16 smartcard. These results likely represent the first documented analysis of an actual second-order DPA attack.

1.2 Power Leakage Model

For the attacks described in this paper, I assume that the processor will leak information about the Hamming weight of the data being processed. I also assume that processing data with higher Hamming weight will consume more power than processing data with lower Hamming weight and that this relationship is roughly linear. Such assumptions are not unreasonable since my research has confirmed that many present-day smartcard processors can exhibit precisely these characteristics.

Let the power consumption at a particular time j be represented by $P[j]$. The value of $P[j]$ can be split into three parts. The first part represents the power contribution that varies with the Hamming weight of the data being processed. The second part represents a constant additive portion and the third part represents noise. This simple linear relationship for $P[j]$ can be written as

$$P[j] = \varepsilon \cdot d[j] + L + n \tag{1}$$

where, $d[j]$ represents the Hamming weight of the intermediate data result at time j, ε represents the incremental amount of power for each extra '1' in the Hamming weight, L represents the additive constant portion of the total power, and n represents the noise. The noise n is assumed to have zero mean, thus when sufficient statistical averaging is used, this noise can usually be ignored.

1.3 Power Attack Countermeasures

Goubin et al. [8] proposed a strategy, called the "duplication method", to protect the DES algorithm from first-order DPA attacks. Their countermeasure works by splitting secret data into two random halves and operating on each half separately. Such an approach causes the power consumption signals to be randomized, thus thwarting DPA. Similar techniques were also proposed to protect the advanced encryption standard algorithms from power attacks [9]. As a generalization, Chari et al. [10] suggested a countermeasure that splits the data into k shares. They proved that the amount of analysis needed to attack such a scheme increases exponentially with respect to k.

Secret splitting schemes protect against first-order DPA attacks, but they may leave an implementation susceptible to higher-order attacks. For some situations, this susceptibility might not be an issue because higher-order attacks are considered to be more difficult. For instance, in a recent paper Daemen et al. summarize that second-order DPA attacks require more complex analysis, increased memory and processing requirements, and an increased number of power consumption measurements [11]. One goal of this paper is to probe the complexity of a second-order DPA attack by investigating a specific example attack. Such research is necessary to ensure the design of secure countermeasures.

1.4 Example Data-Whitening Routines

To better understand the concept of a second-order DPA attack, it is useful to consider some simple examples. The pseudocode for algorithm segments, W_1 and W_2, are given in Fig. 1. These algorithms begin by combining the input *PTI* data with a secret key. This combining step is sometimes referred to as a "whitening process" and is used as a first step in some algorithms; i.e., a specific example is in the Twofish encryption algorithm [12]. The whitening of the input data is performed using the XOR operation. The W_1 algorithm immediately performs this XOR operation at line *A*. Unfortunately,

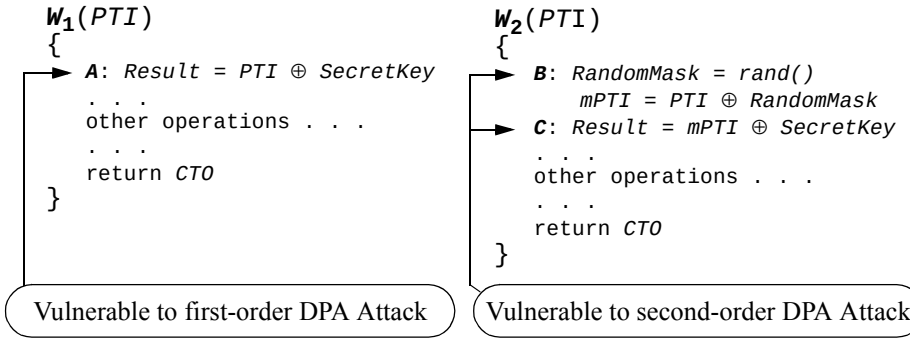


Fig. 1. Routines that Are Vulnerable to DPA Attacks
The routine on the left is vulnerable to a first-order DPA attack at line *A*. The routine on the right is safe from first-order attacks, but is vulnerable to second-order DPA. An attacker can mount a second-order DPA attack by using joint statistics on the power consumed when executing the operations at lines *B* and *C*.

the XOR operation at line *A* can leak information about the secret key. Thus, the W_1 algorithm is potentially susceptible to a first-order DPA attack.

In an attempt to avoid this DPA attack, the W_2 algorithm takes an indirect approach to the whitening operation. First, a random mask is generated at line *B*. Then, the XOR of the *PTI* data and the random mask are computed to produce intermediate result *mPTI*. Next, the XOR of *mPTI* and the secret key is computed at line *C*. The random mask is generated internally and is not observable to an attacker. Thus, when considered separately, the results of the operations at lines *B* and *C* leak only random information, and a DPA attack is prevented. However, when considered jointly, the operations at lines *B* and *C* are vulnerable to a second-order DPA attack.

2 Comparison of First and Second-Order DPA Attacks

The analysis in this section looks at specific attacks against the algorithms shown in Fig. 1. The attacks described here are proven to be sound. The specific steps for one possible DPA attack against the W_1 algorithm are now outlined in the following proposition.

Proposition 1. When the W_1 algorithm is implemented in an N-bit processor, where there is a linear relationship between the instantaneous power consumption and the Hamming weight of the data being processed, the following DPA attack is sound:

1. *Repeat for i equal to* 0 *through* $N-1$ {
2. *Repeat for b* = 0 *to* 1 {
3. *Calculate the average power signal $A_b[j]$ by repeating the following:*{
4. *Set the ith bit of the PTI input to b.*
5. *Set the remaining PTI bits to random values.*
6. *Collect the algorithm's power signal.* } }
7. *Create the DPA bias signal $T[j] = A_0[j] - A_1[j]$.*
8. *T[j] will have a positive bias spike when the ith secret key bit is a one, and will have a negative DPA bias spike when i*th *secret key bit is a zero.* }

Proof. Let j^* be the sample time that corresponds to the time at which the result of line *A* in the W_1 routine is calculated. Also, let the power consumption at this time be represented by *P*. Thus, using the model of Equation (1), $P = d\varepsilon + L + n$, where *d* represents the Hamming weight of the variable *Result* at line *A* of W_1.

Denote the *i*th bits of the *SecretKey* and the *PTI* data as k_i and p_i, respectively. The expected value of the Hamming weight *d* depends on the values of k_i and p_i as follows:

$$\mathsf{E}\left[d \mid k_i \oplus p_i = 0\right] = \frac{N-1}{2} \qquad \mathsf{E}\left[d \mid k_i \oplus p_i = 1\right] = \frac{N+1}{2}$$

When $k_i = 0$, equations for $A_0[j^*]$ and $A_1[j^*]$ can be written in terms of the expected values of P

$$A_0[j^*] \approx \mathsf{E}[P \mid k_i = 0, p_i = 0] = \mathsf{E}[d\varepsilon + L + n \mid k_i = 0, p_i = 0] = \frac{N-1}{2}\varepsilon + L \quad (2)$$

$$A_1[j^*] \approx \mathsf{E}[P \mid k_i = 0, p_i = 1] = \mathsf{E}[d\varepsilon + L + n \mid k_i = 0, p_i = 1] = \frac{N+1}{2}\varepsilon + L \quad (3)$$

Taking the difference of Equations (2) and (3) yields

$$T[j^*] = A_0[j^*] - A_1[j^*] \approx -\varepsilon \qquad \text{when } k_i = 0 \quad (4)$$

Similarly, when $k_i = 1$, equations for $A_0[j^*]$ and $A_1[j^*]$ can be written in terms of the expected values of power consumption P

$$A_0[j^*] \approx \mathsf{E}[P \mid k_i = 1, p_i = 0] = \mathsf{E}[d\varepsilon + L + n \mid k_i = 1, p_i = 0] = \frac{N+1}{2}\varepsilon + L \quad (5)$$

$$A_1[j^*] \approx \mathsf{E}[P \mid k_i = 1, p_i = 1] = \mathsf{E}[d\varepsilon + L + n \mid k_i = 1, p_i = 1] = \frac{N-1}{2}\varepsilon + L \quad (6)$$

Taking the difference of Equations (6) and (5) yields

$$T[j^*] = A_0[j^*] - A_1[j^*] \approx \varepsilon \qquad \text{when } k_i = 1 \quad (7)$$

So, it is clear from Equations (4) and (7) that there should be a positive bias spike when $k_i = 1$ and a negative bias spike when $k_i = 0$. Thus, Proposition 1 is a sound DPA attack. □

2.1 Second-Order DPA attack

Now, consider the W_2 algorithm on the right side of Fig. 1. This algorithm is not vulnerable to the DPA attack of Proposition 1. Instead of directly calculating the XOR of the *SecretKey* and *PTI*, this algorithm first generates a random variable *RandomMask* to mask the value of *PTI*. The secret key is used at line *C*, but the Hamming weight of the result is random. Thus, the power consumption of the result at line *C* cannot be correlated to the values of the secret key or the *PTI* data. The W_2 algorithm seems secure against a first-order DPA attack, yet a second-order DPA attack is definitely possible.

Proposition 2. When the W_2 algorithm is implemented in an N-bit processor, where there is a linear relationship between the instantaneous power consumption and the Hamming weight of the data being processed, the following second-order DPA attack is sound:

1. *Repeat for i equal to* 0 *through* $N-1$ {
2. *Repeat for b* = 0 *to* 1 {
3. *Calculate average statistic* $\bar{S}_b = |P_B - P_C|$ *by repeating the following:*{
4. *Set the ith bit of the PTI input to b.*
5. *Set the remaining PTI bits to random values.*
6. *Collect the algorithm's instantaneous power consumption as lines B and C. Call these values P_B and P_C, respectively.* } }
7. *Calculate the DPA bias statistic* $T = \bar{S}_0 - \bar{S}_1$.
8. *If $T > 0$ then the ith key bit is a one, otherwise it is a zero.*

Proof. The power consumption at lines B and C of W_2, respectively P_B, P_C, can be modeled using the linear relationships:

$$P_B = d_B\varepsilon_B + L_B \text{ and } P_C = d_C\varepsilon_C + L_C \tag{8}$$

where, d_B represents the Hamming weight of the data *RandomMask* at line B, d_C represent the Hamming weight of the data *Result* at line C, ε_B and ε_C represent the extra amount of power for each '1' in the data at lines B and C, and L_B and L_C represent the constant portions of the total power at lines B and C. The noise contributions are ignored, since when averaging is used, these contributions will be removed. Also, to simplify the proof, I initially assume that

$$L_B = L_C \text{ and } \varepsilon_B = \varepsilon_C \tag{9}$$

My experimental results confirmed that the assumptions of equality in Equation (9) are true for the implementation I considered. However, in the general case these equalities may not hold. For this proof I will first consider the case where Equation (9) holds. Then, at the end of this proof I will comment on the more general case.

In the second-order DPA attack of Proposition 2, the value of $\left|P_B - P_C\right|$ is used as a statistic to determine the value of the ith bit of the key. The value of $\left|P_B - P_C\right|$ can be rewritten by using Equations (8) and (9),

$$\left|P_B - P_C\right| = \varepsilon\left|d_B - d_C\right|$$

where, $\varepsilon = \varepsilon_B = \varepsilon_C$. Now, refer back to the W_2 algorithm in Fig. 1. Let the ith bit of the variable *SecretKey* be k_i, the ith bit of the random variable *RandomMask* be r_i and the ith bit of *PTI* be p_i. Recall that the variables d_B and d_C are random variables corresponding to the Hamming weights of the N-bit data processed at lines B and C, respectively. The expected value of d_B is dependent on r_i and the expected value of d_C is dependent on the values of r_i, k_i and p_i

$$\begin{aligned} \mathsf{E}\left[d_B \mid r_i = 1\right] &= \mathsf{E}\left[d_C \mid r_i \oplus k_i \oplus p_i = 1\right] = (N+1)/2 \\ \mathsf{E}\left[d_B \mid r_i = 0\right] &= \mathsf{E}\left[d_C \mid r_i \oplus k_i \oplus p_i = 0\right] = (N-1)/2 \end{aligned} \tag{10}$$

Assuming that the variable *RandomMask* is uniformly distributed and using Equation (10), the values of $\overline{S}_0$ and $\overline{S}_1$ in the attack of Proposition 2 can now be calculated. Recall that when $\overline{S}_0$ is calculated, the ith bit of *PTI* is set to a zero, and that when $\overline{S}_1$ is calculated, the ith bit of *PTI* is set to a one. Thus, when $k_i = 0$, the value of $\overline{S}_0$ can be derived

$$\begin{aligned} \overline{S}_0 &= \frac{1}{2}\mathsf{E}\left[\varepsilon\left|d_B - d_C\right| \,\middle|\, r_i = k_i = p_i = 0\right] + \frac{1}{2}\mathsf{E}\left[\varepsilon\left|d_B - d_C\right| \,\middle|\, r_i = 1,\ k_i = p_i = 0\right] \\ &= 0 \end{aligned} \tag{11}$$

and the value of $\overline{S}_1$ can be derived

$$\overline{S}_1 = \frac{1}{2}\mathrm{E}\left[\varepsilon\left|d_B - d_C\right| \middle| p_i = 1,\ r_i = k_i = 0\right] + \frac{1}{2}\mathrm{E}\left[\varepsilon\left|d_B - d_C\right| \middle| r_i = p_i = 1,\ k_i = 0\right] \quad (12)$$
$$= \varepsilon$$

The combination of Equations (11) and (12) yields

$$T = \overline{S}_0 - \overline{S}_1 = -\varepsilon$$

In the case where $k_i = 1$, the derivation of $\overline{S}_0$ and $\overline{S}_1$ is very similar except that the results are swapped, $\overline{S}_0 = \varepsilon$ and $\overline{S}_1 = 0$. Therefore, when $T < 0$, then $k_i = 0$, and when $T > 0$, then $k_i = 1$. Hence, the sign of T indicates the value of k_i and the attack in Proposition 2 is a sound second-order DPA attack. □

Remark. The proof of Proposition 2 is based on the assumption in Equation (9) that certain parameters are equal. When this equality assumption is not true, the situation can be handled through a process of normalization. Instead of calculating $\overline{S}_0$ and $\overline{S}_1$ by directly using P_B and P_C, normalized versions of P_B and P_C can be used. Normalized versions of P_B and P_C are calculated by subtracting the mean and dividing by the variance. For example, a normalized version of P_B is calculated as

$$\text{normalized } P_B = (P_B - \mathrm{E}[P_B])/\mathrm{var}[P_B] \quad (13)$$

By using normalized values for P_B and P_C, the equality assumption of Equation (9) is effectively forced to be true, thus resulting in a sound attack.

3 Experimental Results

In this section I provide experimental results showing the practical aspects of the previously described DPA attacks. The EEPROM memory of an ST16 smartcard was programmed with versions of the W_1 and W_2 algorithms from Fig. 1. Then, the attacks from Propositions 1 and 2 were implemented and tested against this smartcard. Statistical results from measuring the smartcard's power consumption were collected and analyzed.

In a first-order DPA attack, knowledge of design information is not required. In a second-order DPA attack, however, knowledge of the algorithm code and the processor operation is much more important. Without such knowledge, attackers will not know which points in the power consumption signal are important. For example, in the W_2 algorithm these points correspond to the execution of lines *B* and *C*. Attackers that do not know which points to observe will need to resort to additional statistical analysis to find these points. Although possible, such an approach makes an attack more difficult, especially as the order of the DPA attack grows. To avoid these complications in my experiments, I assumed that the attacker knows exactly which points in the power trace to monitor.

With knowledge of which points in a power trace to monitor, it becomes easy to implement both DPA attacks. The hardware used for this experiment was simply a PC,

a digital sampling oscilloscope, and a smartcard reader. The smartcard reader was physically modified to allow for easy power measurements. This modification entailed placing an 18 ohm resistor in series between the smartcard and reader's ground pins. Power consumption was monitored by sampling the voltage across this resistor. The smartcard was clocked at 3.57 Mhz and the power signal was asynchronously sampled by setting the oscilloscope's sampling rate to 1.0 Gsamples/second.

Implementors of smartcard systems often wonder how much sample data an attacker will need for a successful attack. I designed an experiment that looks at this question for my given ST16 implementations. In my experiments, DPA attacks were run and power signals were collected. As each power signal was collected, an updated value of T was calculated. The accuracy of my attacks increased as the number of power signals used to calculate T increased. Thus, as the number of power signals increased, the sign of T converged to be either positive or negative, depending on the value of the bit being attacked.

My experiments were run a number of times and typical graphs showing the convergence of T versus the number of power signals are given in Figs. 2 and 3. The results of the first-order DPA attack in Proposition 1 are shown in Fig. 2. The plots in this figure show the convergence of T for each bit of the byte being attacked. In this example, the byte being attacked is equal to 0x6B. Thus, for bit #0, T should converge to a positive value; for bit #1, T should converge to a positive value; for bit #2, T should converge to a negative value, etc. My experimental result confirmed that this attack is practical. Fewer than 50 power signals were needed for T to converge to the correct value for all bits. In fact, for most bits, T converged with much fewer than 50 signals.

My second experimental results, that confirm the attack of Proposition 2, are given in Fig. 3. Again, it is clear that this attack is practical. An interesting observation is that T converges at different rates for different bits in a byte. For some bits, T converged quickly; fewer than 50 power signals were needed. However, for other bits, T converged more slowly. For example, in Fig. 3, bit #5 requires about 2,500 power signals before T stabilizes to the correct sign. In general, the convergence of T in the second-order attack is slower and more erratic than in the first-order attack. Surprisingly, however, for some bits, T converges nearly as fast for both attacks.

It should be stressed, however, that these experimental results apply only to my specific ST16 implementation that was being tested. Implementors of other systems will need to test or simulate their own implementations to accurately assess any vulnerabilities.

4 Developing an Optimal Second-Order DPA Attack

The attack from Proposition 2 is based on the statistic $\bar{S} = E[|P_B - P_C|]$. The formula for statistic $\bar{S}$ was chosen using an ad hoc approach based on the linear model of the power consumption signal. Although, an attack using $\bar{S}$ was experimentally shown to be practical, statistics that use other combinations of P_B and P_C may lead to even better attacks. For example, Chari et al. [10] suggest an alternate statistic, based on multiplying P_B and P_C, rather than taking their difference. Finding the optimal statistic for a second-order DPA attack is the topic that will now be investigated.

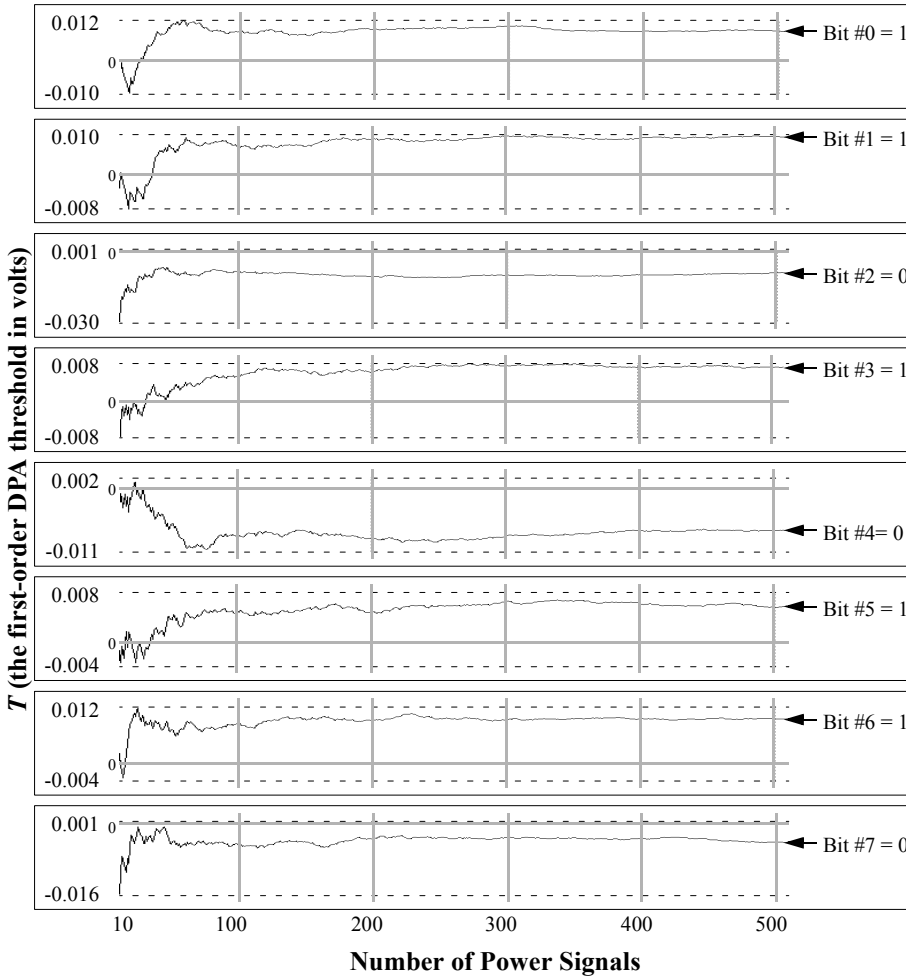


Fig. 2. First-Order DPA Threshold versus the Number of Power Signals
The above plot shows the convergence of T versus the number of power signals. The byte being attacked is equal to 0x6B and the resulting convergence plots for each bit of this byte are given above. The horizontal shaded lines denote the axis where T equals zero. A positive value for T indicates a bit is a one and a negative value indicates a bit is a zero. In all cases, T converges to the correct bit using fewer than 50 power signals.

A DPA attack against a secret bit of a key is a perfect example of a classic decision problem. Given noisy power consumption data, an attacker needs to decide whether a key bit is a zero or a one. An optimal decision is made when the probability of a wrong decision is minimized. Given a set of power consumption data, one can compute probabilities to determine the optimal decision. Let k_i, r_i and p_i be defined as before and let Ψ represent all of the observed power consumption data. During an attack, I assume that adversaries know p_i, so it is sufficient for them to determine $k_i \oplus p_i$. In an optimal

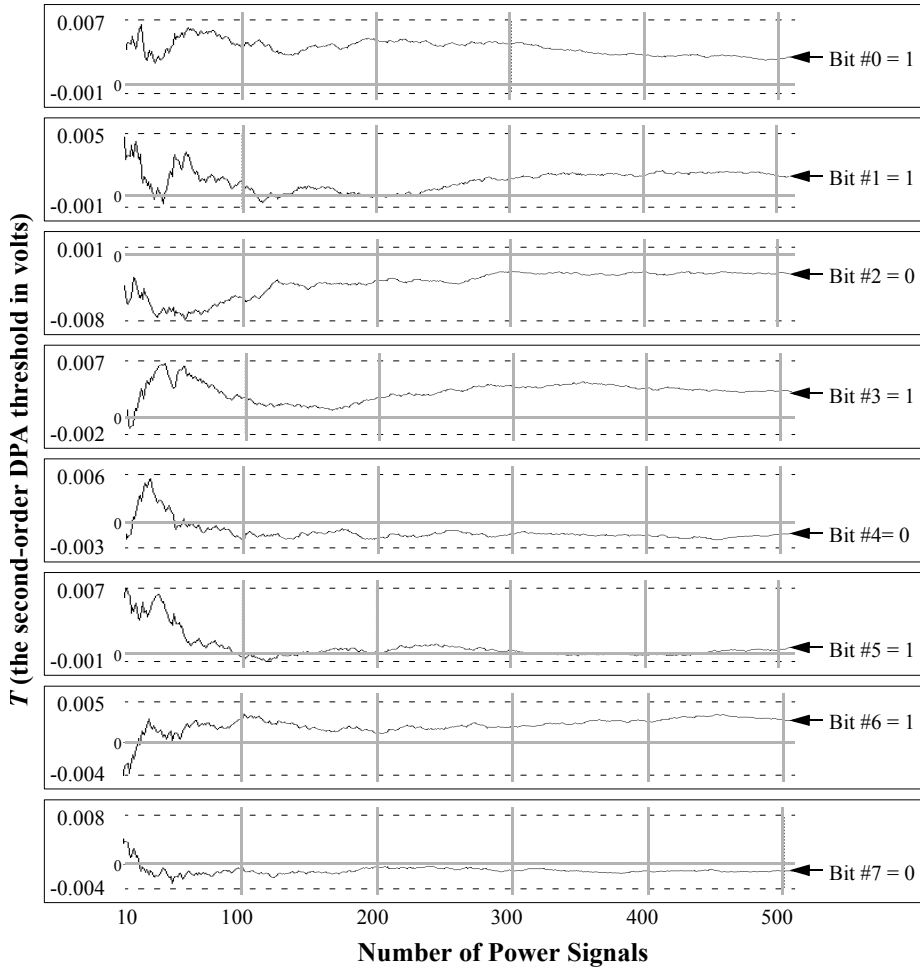


Fig. 3. Second-Order DPA Threshold versus the Number of Power Signals
The above plot shows the convergence of T versus the number of power signals. The byte being attacked is equal to 0x6B and the resulting convergence plots for each bit of this byte are given above. The horizontal shaded lines denote the axis where T equals zero. A positive value for T indicates a bit is a one and a negative value indicates a bit is a zero. In most cases T converges to the correct bit using fewer than 50 power signals. However, in the case of bit #5, T requires more than 2,500 power signals to converge.

attack, the attacker will choose the value for $k_i \oplus p_i$ that was more likely to have produced the observed data Ψ. Symbolically, the decision problem is reduced to solving the inequality

$$\Pr[\Psi | k_i \oplus p_i = 0] \lesssim \Pr[\Psi | k_i \oplus p_i = 1] \tag{14}$$

The job of the attacker is to calculate both of the probabilities in Equation (14). The larger of the probabilities indicates the most likely value for $k_i \oplus p_i$. To begin calculating these probabilities, let the observed power consumption data Ψ be represented by N vectors, where each vector is the power consumption data from a single run of the algorithm. In a second-order DPA attack, the kth vector is composed of two elements (b_k, c_k), where b_k and c_k represent the instantaneous power consumptions for the two instructions being attacked. In the W_2 algorithm, b_k and c_k represent the normalized power consumption of the instructions at lines B and C, respectively.

The power consumption values b_k and c_k are random variables having probability density functions f_b and f_c, respectively. The distributions for these variables is assumed to be Guassian and the distributions are

$$f_b(b_k) \sim \mathrm{N}(0, \hat{\sigma}^2) \qquad f_c(c_k) \sim \mathrm{N}(0, \hat{\sigma}^2)$$

The probability distributions for b_k and c_k, conditioned on k_i, r_i and p_i, are also Guassian.

$$\begin{aligned} f_b(b_k \mid r_i = 0) &\sim \mathrm{N}(-\frac{\varepsilon}{2}, \sigma^2) & f_b(b_k \mid r_i = 1) &\sim \mathrm{N}(\frac{\varepsilon}{2}, \sigma^2) \\ f_c(c_k \mid r_i \oplus k_i \oplus p_i = 0) &\sim \mathrm{N}(-\frac{\varepsilon}{2}, \sigma^2) & f_c(c_k \mid r_i \oplus k_i \oplus p_i = 1) &\sim \mathrm{N}(\frac{\varepsilon}{2}, \sigma^2) \end{aligned} \tag{15}$$

For shorthand, the results of Equation (15) can be written as

$$\begin{aligned} f_b^- &= f_b(b_k \mid r_i = 0) & f_b^+ &= f_b(b_k \mid r_i = 1) \\ f_c^- &= f_c(c_k \mid r_i \oplus k_i \oplus p_i = 0) & f_c^+ &= f_c(c_k \mid r_i \oplus k_i \oplus p_i = 1) \end{aligned}$$

Using this shorthand notation and the assumption that r_i is equally likely to be a one or a zero, the joint conditional probability distributions of b_k and c_k can be shown to be

$$\begin{aligned} f_{b,c}(b_k, c_k \mid k_i \oplus p_i = 0) &= \frac{1}{2} f_b^- f_c^- + \frac{1}{2} f_b^+ f_c^+ \\ f_{b,c}(b_k, c_k \mid k_i \oplus p_i = 1) &= \frac{1}{2} f_b^- f_c^+ + \frac{1}{2} f_b^+ f_c^- \end{aligned} \tag{16}$$

The joint conditional probabilities of Equation (16) can now be used to solve the decision problem that was given in Equation (14).

Theorem 1. An optimal second-order DPA attack using N vectors (b_k, c_k), where each vector is assumed to be independent and b_k and c_k are assumed to be jointly normal random variables, reduces to the decision problem

$$\prod_{k=0}^{N-1} \cosh(b_k + c_k) \lessgtr \prod_{k=0}^{N-1} \cosh(b_k - c_k)$$

Proof. Using Equation (16) and the assumption that the (b_k, c_k) vectors are independent for different values of k, the probabilities in Equation (14) can be shown to be

$$\Pr[\Psi | k_i \oplus p_i = 0] = \prod_{k=0}^{N-1} f_{b,c}(b_k, c_k | k_i \oplus p_i = 0) = \frac{1}{2^N} \prod_{k=0}^{N-1} (f_b^- f_c^- + f_b^+ f_c^+)$$

$$\Pr[\Psi | k_i \oplus p_i = 1] = \prod_{k=0}^{N-1} f_{b,c}(b_k, c_k | k_i \oplus p_i = 1) = \frac{1}{2^N} \prod_{k=0}^{N-1} (f_b^- f_c^+ + f_b^+ f_c^-)$$

The decision problem is to solve an inequality, so terms appearing in both equations can be dropped. The decision problem can now be written as

$$\prod_{k=0}^{N-1} (f_b^- f_c^- + f_b^+ f_c^+) \lessgtr \prod_{k=0}^{N-1} (f_b^- f_c^+ + f_b^+ f_c^-) \tag{17}$$

Now, the appropriate Gaussian distribution functions can be substituted in for f_b^-, f_c^-, f_b^+ and f_c^+. The left side of Equation (17) can now be written as

$$\prod_{k=0}^{N-1} \left(\exp\left\{ -\frac{1}{2} \left[\frac{\left(b_k + \frac{\varepsilon}{2}\right)^2 + \left(c_k + \frac{\varepsilon}{2}\right)^2}{\sigma^2} \right] \right\} + \exp\left\{ -\frac{1}{2} \left[\frac{\left(b_k - \frac{\varepsilon}{2}\right)^2 + \left(c_k - \frac{\varepsilon}{2}\right)^2}{\sigma^2} \right] \right\} \right)$$

$$\prod_{k=0}^{N-1} \left(\exp\left\{ -\frac{1}{2} \left[\frac{b_k^2 + c_k^2 + b_k\varepsilon + c_k\varepsilon + \frac{\varepsilon^2}{2}}{\sigma^2} \right] \right\} + \exp\left\{ -\frac{1}{2} \left[\frac{b_k^2 + c_k^2 - b_k\varepsilon - c_k\varepsilon + \frac{\varepsilon^2}{2}}{\sigma^2} \right] \right\} \right)$$

and the right side of Equation (17) can be written as

$$\prod_{k=0}^{N-1} \left(\exp\left\{ -\frac{1}{2} \left[\frac{\left(b_k + \frac{\varepsilon}{2}\right)^2 + \left(c_k - \frac{\varepsilon}{2}\right)^2}{\sigma^2} \right] \right\} + \exp\left\{ -\frac{1}{2} \left[\frac{\left(b_k - \frac{\varepsilon}{2}\right)^2 + \left(c_k + \frac{\varepsilon}{2}\right)^2}{\sigma^2} \right] \right\} \right)$$

$$\prod_{k=0}^{N-1} \left(\exp\left\{ -\frac{1}{2} \left[\frac{b_k^2 + c_k^2 + b_k\varepsilon - c_k\varepsilon + \frac{\varepsilon^2}{2}}{\sigma^2} \right] \right\} + \exp\left\{ -\frac{1}{2} \left[\frac{b_k^2 + c_k^2 - b_k\varepsilon + c_k\varepsilon + \frac{\varepsilon^2}{2}}{\sigma^2} \right] \right\} \right)$$

Again, the terms that appear in both equations can be cancelled, resulting in the left side of Equation (17) being reduced to

$$\prod_{j=0}^{N-1}\left(\exp\left\{-\frac{\varepsilon}{2\sigma^2}[b_j+c_j]\right\}+\exp\left\{\frac{\varepsilon}{2\sigma^2}[b_j+c_j]\right\}\right) = \prod_{j=0}^{N-1}\cosh\left[\frac{\varepsilon}{2\sigma^2}(b_j+c_j)\right]$$

and the right side of Equation (17) being reduced to

$$\prod_{k=0}^{N-1}\left(\exp\left\{-\frac{\varepsilon}{2\sigma^2}[b_k-c_k]\right\}+\exp\left\{\frac{\varepsilon}{2\sigma^2}[b_k-c_k]\right\}\right) = \prod_{k=0}^{N-1}\cosh\left[\frac{\varepsilon}{2\sigma^2}(b_k-c_k)\right]$$

Finally, the common factor of $\frac{\varepsilon}{2\sigma^2}$ can be removed, resulting in the decision problem given by Theorem 1. □

Remark. When $|b_k - c_k| \gg 1$, the ad hoc attack in Proposition 2 is a close approximation to the optimal decision problem. This can be seen since when $|b_k - c_k| \gg 1$,

$$\prod_{k=0}^{N-1}\cosh(b_k-c_k) \propto \sum_{k=0}^{N-1}|b_k-c_k|$$

Thus, the optimal attack statistic is approximately proportional to the ad hoc attack statistic. To verify this result, I repeated my previous experiments using the optimal attack. The results confirmed that the optimal and ad hoc second-order attacks are approximately equal.

5 Countermeasures to Higher-Order DPA Attacks

The long-term solution to these attacks it to develop hardware that does not leak secret information. Examples for potentially secure hardware have been reported by Moore et al. [13] and Kessel [14]. Statistical tests such as those suggested in [15] can be used to evaluate such hardware. Until such hardware is deployed, many of the same countermeasures that are effective against first-order DPA attacks may also help resist higher-order DPA attacks. Adding random time delays that are difficult for an attacker to remove is one such countermeasure. Also, keeping the implementation details secret can be very effective against higher-order attacks. Unlike first-order DPA, higher-order DPA is more complex if these details are not known to the attacker. Secret splitting schemes, such as the one proposed by Chari et al. [10], may also be effective.

6 Conclusions

Whichever countermeasures are chosen, designers will need to test their implementations for specific vulnerabilities. The theoretical analysis and the example of a practical second-order DPA attack provided in the paper will hopefully help future designers make their implementations more secure.

References

1. J. Kelsey, B. Schneier, D. Wagner, and C. Hall, "Side Channel Cryptanalysis of Product Ciphers," in *Proceedings of ESORICS '98*, Springer-Verlag, September 1998, pp. 97-110.
2. Paul Kocher, Joshua Jaffe, and Benjamin Jun, "Differential Power Analysis," in proceedings of *Advances in Cryptology–CRYPTO '99*, Springer-Verlag, 1999, pp. 388-397.
3. Thomas S. Messerges, Ezzy A. Dabbish, and Robert H. Sloan, "Investigations of Power Analysis Attacks on Smartcards," *Proceedings of USENIX Workshop on Smartcard Technology*, May 1999, pp. 151-161.
4. Eli Biham and Adi Shamir, *"*Power Analysis of the Key Scheduling of the AES Candidates*," Second Advanced Encryption Standard Candidate Conference*, March 1999, `http://www.nist.gov/aes` .
5. S. Chari, C. Jutla, J.R. Rao, and P. Rohatgi, *"*A Cautionary Note Regarding Evaluation of AES Candidates on Smart-Cards*," Second Advanced Encryption Standard Candidate Conference*, http://www.nist.gov/aes, March 1999.
6. Thomas S. Messerges, Ezzy A. Dabbish, and Robert H. Sloan, "Power Analysis Attacks of Modular Exponentiation in Smartcards," in proceedings of *Workshop on Cryptographic Hardware and Embedded Systems*, Springer-Verlag, August 1999, pp. 144-157.
7. Jean-Sébastien Coron, "Resistance Against Differential Power Analysis for Elliptic Curve Cryptosystems," in proceedings of *Workshop on Cryptographic Hardware and Embedded Systems*, Springer-Verlag, August 1999, pp. 292-302.
8. Louis Goubin and Jacques Patarin, "DES and Differential Power Analysis – The Duplication Method," in proceedings of *Workshop on Cryptographic Hardware and Embedded Systems*, Springer-Verlag, August 1999, pp. 158-172.
9. Thomas S. Messerges, "Securing the AES Finalists Against Power Analysis Attacks," in proceedings of *Fast Software Encryption Workshop 2000*, Springer-Verlag, April 2000.
10. Suresh Chari, Charanjit S. Jutla, Josyula R. Rao and Pankaj J. Rohatgi, "Towards Sound Approaches to Counteract Power-Analysis Attacks," in proceedings of *Advances in Cryptology–CRYPTO '99*, Springer-Verlag, 1999, pp. 398-412.
11. Joan Daemen, Michael Peeters and Gilles Van Assche, "Bitslice Ciphers and Power Analysis Attacks," in proceedings of *Fast Software Encryption Workshop 2000*, Springer-Verlag, April 2000.
12. Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall and Niels Ferguson, *The Twofish Encryption Algorithm : A 128-Bit Block Cipher*, John Wiley & Sons, 1999, ISBN: 0471353817.
13. S. W. Moore, R. Anderson and M. Kuhn, "Self-timed Technology to Reduce Smartcard Fraud," in proceedings of *ACiD-WG Workshop*, Grenoble, February 2000.
14. Joep Kessels, "Applying Asynchronous Circuits in Contactless Smartcards," in proceedings of *ACiD-WG Workshop*, Grenoble, February 2000.
15. Jean-Sébastien Coron, Paul Kocher and David Naccache, "Statistics and Secret Leakage," in proceedings of *Financial Cryptography*, Springer-Verlag, February 2000.

Differential Power Analysis in the Presence of Hardware Countermeasures

Christophe Clavier[1], Jean-Sébastien Coron[2], and Nora Dabbous[2]

[1] Gemplus Card International, Avenue du Pic de Bretagne, BP 100
Gemenos, F-13881, France
[2] Gemplus Card International, 34 rue Guynemer
Issy-les-Moulineaux, F-92447, France
{christophe.clavier,jean-sebastien.coron,nora.dabbous}@gemplus.com

Abstract. The silicon industry has lately been focusing on side channel attacks, that is attacks that exploit information that leaks from the physical devices. Although different countermeasures to thwart these attacks have been proposed and implemented in general, such protections do not make attacks infeasible, but increase the attacker's experimental (data acquisition) and computational (data processing) workload beyond reasonable limits.
This paper examines different ways to attack devices featuring random process interrupts and noisy power consumption.
Keywords: Power analysis, smart card, hardware countermeasure, random process interrupt.

1 Introduction

In past decades, cryptanalysis focused on exploiting mathematical weaknesses in algorithms to break into the targeted systems. As a result, modern cryptosystems are generally designed to better withstand logical threats and attackers are concentrating on analysis of side channel leakage. Among these, timing attacks, Simple Power Attacks (SPAs), Differential Power Attacks (DPAs) and TEMPEST are certainly best known [7,8,1].

Most of the time, the cryptographic kernels of products used are not isolated in perfectly tamper-proof locations. It has long been known that execution time, power consumption, radio frequencies, magnetic field values, etc. could leak some information on sensitive data. After a first glance, cryptographers had concluded that these would only be able to reveal partial information, therefore not causing a real danger. It was only in 1996 that Paul Kocher demonstrated that side channel attacks were effective enough to recover secret keys in numerous cryptosystems. Differences in execution time were the first to be exploited [8] and in 1999 it was shown that power consumption measurements, if carefully analyzed, could also reveal sensitive information [7]. Now that these pitfalls have been uncovered, analyzed and better understood, different countermeasures are being studied in order to minimize the side channel attacks' impact by reducing the signals that can be exploited to perform these attacks, or making them useless.

Ç.K. Koç and C. Paar (Eds.): CHES 2000, LNCS 1965, pp. 252–263, 2000.

Following a more or less uniform reaction pattern, most manufacturers came-up with software and hardware means to hide and randomize sensitive data. This paper focuses on DPA on systems in which hardware countermeasures have been implemented. The experiments described below were successfully carried out on DES, proving that the some countermeasures, initially thought to be heuristicly sufficient, do not guarantee the claimed security level.

Section 2 briefly recalls DPA and explains how to perform the attack on devices featuring *random process interrupts* (RPIs) and noisy power consumption. Section 3 focuses on a first method to eliminate the chip's hardware protection. Section 4 improves this method, as long as the guidelines in section 5 are taken into account.

2 DPA in the Presence of Random Process Interrupts

Power attacks isolate information correlated to operations or manipulated data by examining devices' power consumption. Following Kocher's terminology [7], Simple Power Analysis (SPA) consists in directly analyzing a device's power consumption, whereas Differential Power Analysis (DPA) spots correlation between the data being manipulated and the side channel information.

2.1 Differential Power Attacks

DPA can be easily performed on the first DES round if the plaintext is available or on the last DES round if the ciphertext is known. We will recall the basic DPA attack on DES round one. Given the plaintext and the round subkey, the attacker can calculate the input to the S-box functions and, by table look-up, their output. As is, DPA on DES is performed on one S-box at a time and allows to determine key bits six by six by targeting the output of one S-box. To perform a DPA, different *power consumption curves* (PCCs) of the device must first be collected. In the basic attack, PCCs are grouped according to one among the 4 S-box output bits observed. If the bit is a 1 the power values are added, if it's a 0 they are subtracted to calculate a *differential curve*. An attacker supposedly has no information on the key so, when performing DPA, he must calculate 64 differential traces, one for each of the 2^6 6-bit partial subkey combinations. A spike will appear in the differential curve that was plotted by using the correct subkey bits where the selection function is correlated to the value of the bit being manipulated. The trace will only feature moderate noise (in this model correlations between different key values are neglected) in the other 63 differential curves obtained with incorrect subkey bits. Theoretically, any bit among the 4 S-box output bits could be analyzed to classify the PCCs. A differential trace obtained by analyzing the other bits could be calculated to confirm the results. More on this will be said in section 4.

As the goal of this research was to test the effectiveness of hardware countermeasures, we executed a DPA plaintext attack. The implementation of a DPA ciphertext attack is straightforward, instead of making assumptions on S-box

inputs and analyzing the outputs, the inverse approach would be followed. Attackers most probably have knowledge of the ciphertext rather than the plaintext and therefore would run an attack on DES round 16. All results reported in this paper are valid for DPA ciphertext attacks as well.

As explained in [7], the differential trace is calculated as:

$$\Delta_D[j] = \frac{\sum_{i=1}^{N} D(P_i,K_s)T_i[j]}{\sum_{i=1}^{N} D(P_i,K_s)} - \frac{\sum_{i=1}^{N} (1-D(P_i,K_s))T_i[j]}{\sum_{i=1}^{N} (1-D(P_i,K_s))} = \varepsilon_1 - \varepsilon_0$$

where K_s are the six unknown key bits, P_i the i-th known plaintext, $D(P_i, K_s)$ the selection function, $T_i[j]$ the j-th sample of the PCC and $\Delta_D[j]$ the j-th element of the differential trace.

The number of PCCs necessary to perform the attack heavily depends on the measurement conditions: the lower the noise, the fewer curves are necessary. We refer the reader to [4] for several useful guidelines. For the spike to be identified

$$\varepsilon_1 - \varepsilon_0 > \sigma/\sqrt{N}, \tag{1}$$

must hold, where σ represents the noise and N the number of necessary PCCs.

Better acquisition equipment and higher sampling rates yield lower noise. Although chip-dependent, table 1 gives a rough idea about the required N as a function of the acquisition experiment's sampling rate S expressed in MHz for a card running at 3.68 Mhz. The card used to obtain such values featured no countermeasures designed to thwart DPA attacks.

Table 1. N as a Function of the Attacker's Equipment Sampling Rate.

N	600	500	120	100
S	50	100	500	1000

2.2 Random Process Interrupts

One of the most common countermeasures against DPA is the introduction of *random process interrupts* (RPIs). Instead of executing all the operations sequentially, the CPU interleaves the code's execution with that of dummy instructions so that corresponding operation cycles do not match because of time shifts. This has the effect of smearing the peaks across the differential trace due to a desyncronisation effect, known in digital signal processing under the name of *incoherent averaging* [9]. The time shifts can be considered as added noise. Needless to say, RPIs do not make the attack theoretically infeasible but increase N considerably.

Assuming that RPIs occur with a constant probability p, even if a spike should be seen on the differential trace because the correct key was guessed, the

spike might remain confused with the noise because it was spread over consecutive cycles. Due to RPIs, the spike that actually appears follows a gaussian distribution, thoroughly characterized by a mean position μ and a variance v that can be precisely calculated.

Suppose the spike on the differential trace should be seen after n cycles. If RPIs occurred, a spike will appear after $n + C_n$ cycles, where the delay $C_n = \sum_{i=1}^{n} c_i$, c_i being the i-th cycle, with $c_i = 1$ if an RPI occurred and $c_i = 0$ if not.

The mean position for the spike is:

$$\mu =< C_n + n >= \sum_{i=1}^{n} < c_i > + n = np + n,$$

and the variance v is:

$$v =< C_n^2 > - < C_n >^2 = \sum_{i=1}^{n} \mathrm{Var}(c_i) = n(1-p)p \cong np.$$

We can thus estimate the standard deviation[1] $\delta \cong \sqrt{np}$, which means that (for all experimental purposes) the spike will be distributed over a $\pm\delta$ range centered around μ. In other words, we consider that the spike was distributed over $k = 2\delta \cong 2\sqrt{np}$ consecutive cycles. The spike will thus be visible if:

$$\frac{(\varepsilon_1 - \varepsilon_0)}{k} > \frac{\sigma}{\sqrt{N'}}, \tag{2}$$

We therefore infer from (2) and (1) that the number of RPI-protected PCCs necessary to put the DPA back on it's feet is:

$$N' = k^2 N.$$

As a characteristic example, if the DPA spike should be seen after $n = 1600$ cycles (which can typically be the case for a spike observed after the first DES round) then $p = 12\%$ yields $k \cong 28$. This means that the number of RPI-protected PCCs necessary to re-run the same attack must be multiplied by a factor of 784. For research purposes this attack was indeed successfully performed, but in reality such an attack is improbable because of the number of PCCs that should be acquired. In the next section, we will describe a method that allows to consistently reduce the number of PCCs necessary to run the attack.

3 Spike Re-construction by Integration

The spike's amplitude $(\varepsilon_1 - \varepsilon_0)$ can be re-constructed in order to decrease the number of power consumption curves needed. Because of desynchronization, the spike's amplitude is divided by a value bound by k, but it's original amplitude can be restored by integrating the RPI-protected signal over k consecutive cycles. The new signal value is $k \times \frac{(\varepsilon_1 - \varepsilon_0)}{k}$, the new noise value is $\frac{\sigma}{\sqrt{N''}} \times \sqrt{k}$. Consequently the spike will be visible if

[1] we intentionally use δ instead of the letter σ that we reserve for further use.

$$\varepsilon_1 - \varepsilon_0 > \frac{\sigma}{\sqrt{N''}} \times \sqrt{k}.$$

Therefore, to restore the same signal to noise ratio as in (1), the following equation must be satisfied:

$$N'' = kN.$$

As this method implies integrating PCC values on k consecutive cycles, we called it *Sliding Window DPA* (SW-DPA). Implementing SW-DPA involves two steps. First of all, a classic (Kocher-style) differential curve must be obtained. Unless a very high number of PCCs is used, even for the correct key guess no spike will appear because of the RPIs. For the spikes to appear, RPI-protected PCCs must be integrated. This step consists of adding points on k consecutive cycles from the differential PCC obtained in step one. To visualize this operation, the reader may imagine a comb with k teeth, each corresponding to a point on the differential PCC created in step one. The distance between two consecutive teeth on the comb must match the number of time samples separating two consecutive cycles. Integration is obtained by adding the power value of the points indicated by the comb.

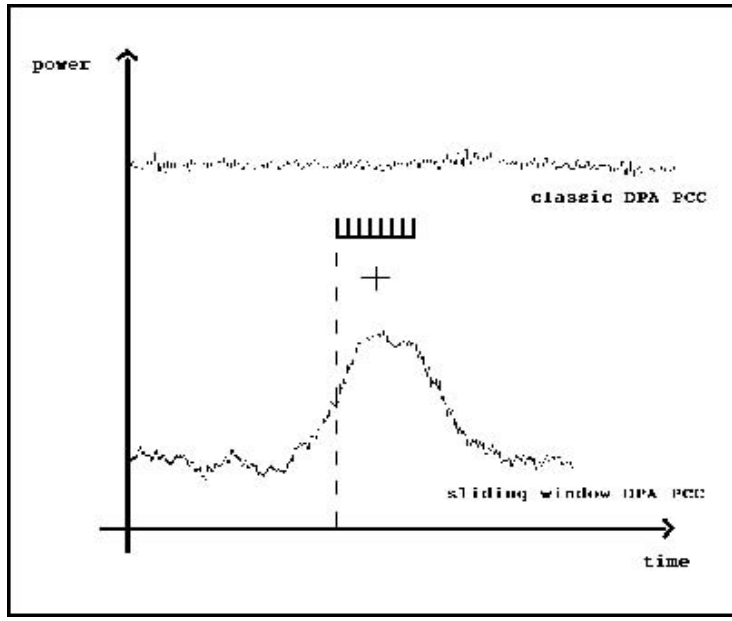


Fig. 1. The Integration Operation.

If the same figures as before are considered (that is $k = 28$) the attack can be implemented by increasing by a factor of 28 the number of necessary curves.

If a DPA can be performed using 120 unprotected power consumption curves acquired at 500 MHz, then only 3360 RPI-protected curves would be necessary.

Figure 2 shows real-life differential curves obtained with an integration window of 30 for a right (upper curve) and a wrong (lower curve) key guess.

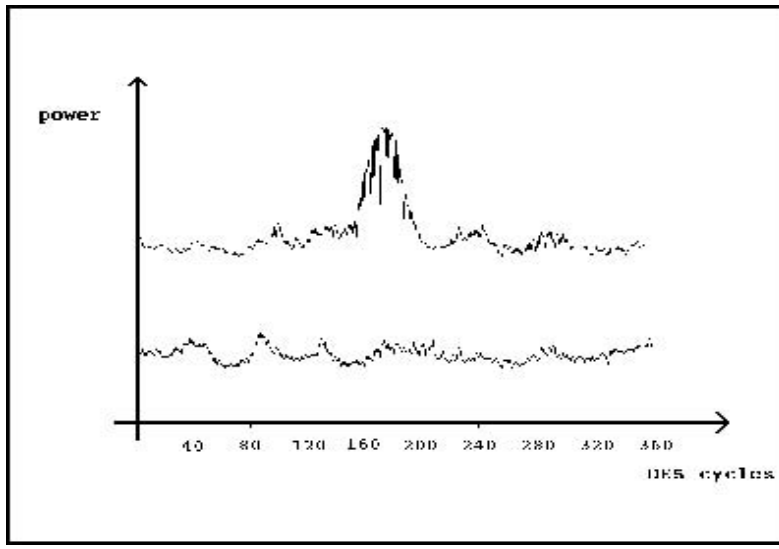


Fig. 2. Differential Trace for Correct and Wrong Key Guesses.

We'd like to clarify that all cycle number values are reported to show the reader how wide the DPA spike is. They are not intended as an absolute reference from the beginning of the DES execution, as this number greatly depends on the way the function is implemented.

Figure 3 shows a zoomed in view of the differential trace spike for the correct key. It can clearly be seen that what looks like a single spike in the larger view is made up of different "spike portions". This is the integration's effect, which adds up the fractions of all distributed spikes only when accurately centered.

4 The Hamming Integration Variant

When determining the key by classic DPA, PCCs are classified by observing only one out of the four S-box output bits. Experimentally we obtained 4 differential traces per S-box, each one by examining a different S-box output bit, and noticed that some output bits leak more information than others. To successfully perform a DPA, an attacker could predetermine which bits yield better spikes for correct

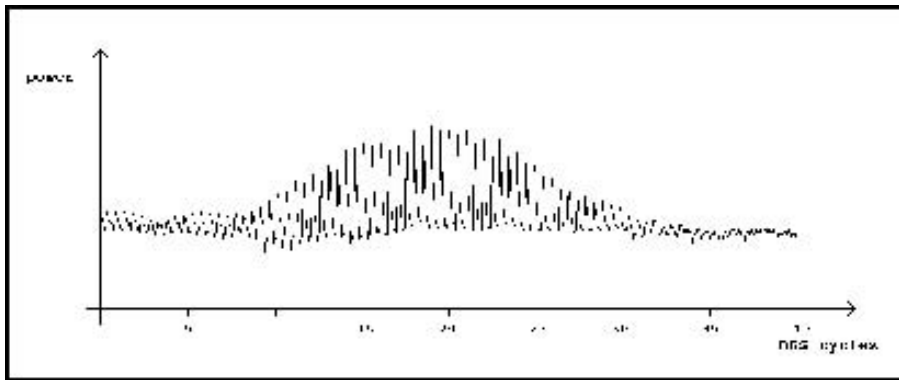


Fig. 3. Enlarged View of the Right Guess Spike.

key guesses. Our approach, though, was to take advantage of the information gathered from all 4 S-box output bits simultaneously.

Let us assume that the chip's power consumption is proportional to the output's *Hamming weight*. If only one S-box output bit is observed and PCCs are classified according to this bit's value, the spike's amplitude will be proportional to:

$$< H >_{s_i=1} - < H >_{s_i=0} = (1 + \tfrac{3}{2}) - (0 + \tfrac{3}{2}) = 1,$$

where H is the power consumption at a certain instant and s_i is the i-th S-box output bit. In particular, we set $< H >_{s_i=1} = (1 + \frac{3}{2})$ because we are considering S-box output bits for which one bit is certainly equal to one, whereas the remaining three bits are equal to one with probability one half.

The signal to noise ratio is equal to:

$$\text{SNR} = \frac{1}{\frac{\sigma}{\sqrt{N}}},$$

where σ is the differential curve's standard deviation and N the number of curves considered.

If, instead, all 4 S-box output bits are observed simultaneously, a new PCC classification criterion must be designed. Curves could be classified according to the total S-box Hamming weight, that is curves for which four or three ones appear could be grouped in one class while, on the other hand, curves for which zero or a single one are shown could be grouped in a second class, and curves with two ones and two zeros in the output could be discarded. In this case the spike's amplitude would be proportional to:

$$< H >_{b=4,3} - < H >_{b=0,1} = \frac{16}{5} - \frac{4}{5} = \frac{12}{5} = 2.4, \tag{3}$$

where b is the number of ones. In particular, $< H >_{b=4,3} = \frac{16}{5}$ is the Hamming weight mean when three or four ones appear in all possible combinations of four bit strings.

We call this method the Hamming integration variant. This, combined with an integration in case RPIs had been inserted, yields a much higher spike.

The signal to noise ratio using this method is equal to:

$$\text{SNR} = \frac{\frac{12}{5}}{\frac{\sigma}{\sqrt{\frac{10}{16} \times N}}} = 1.9 \times \frac{1}{\frac{\sigma}{\sqrt{N}}},$$

where σ is the differential curve's standard deviation and $10N/16$ is the number of curves considered. In the above calculation we take into account the fact that $6N/16$ curves are discarded because a PCCs is not processed whenever two ones (and two zeros) appear as S-box output.

The 1.9 SNR between the two different methods is a theoretical result. As stated before, some output bits leak more information than others and the experimental ratio between the observed spikes greatly depends on the targeted output bit. This can clearly be seen on figure 4, for which the curves were obtained by examining two different S-box output bits for the first DES S-box. The upper curve was obtained by SW-DPA on the first S-box output bit, whereas the lower curve was obtained by SW-DPA on the fourth one.

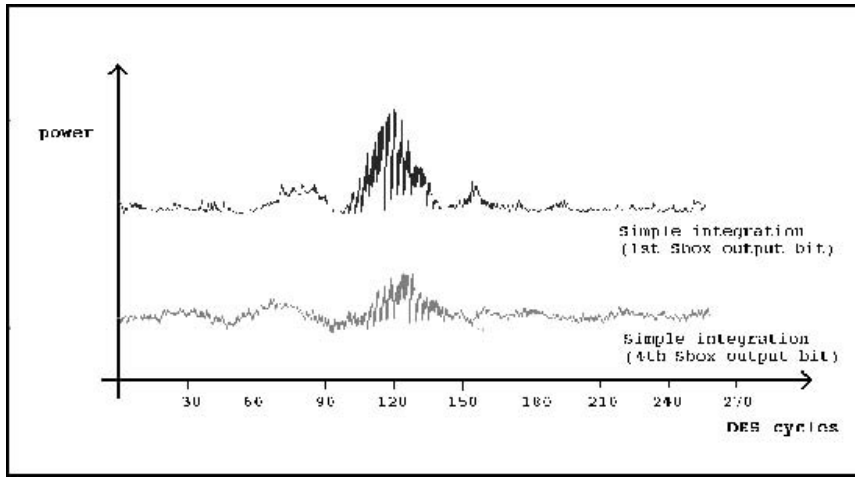


Fig. 4. Differential trace obtained on first S-box applying SW-DPA on the first output bit (upper curve) and applying SW-DPA on the fourth output bit (lower curve).

5 Redefining the Selection Function

A very strong correlation exists between the chip's power consumption and the operation being executed. This value is quite high during data transfer between the CPU and the external RAM, so this operation, performed after an S-box output is determined, is usually targeted for DPA. The assumption made to create and interpret the differential trace curve is that the power consumption is different when the S-box output bit is a 0 or a 1. Power consumption, though, does not *only* depend on the output value, but also on the transitions that occur on the bus (*c.f.* to [4] for instance). Assuming ordinary CMOS inverter implementation, a high power consumption is to be expected when a 1 is being written onto a bus line previously discharged, or when a 0 is being written onto a bus line previously charged. Values in these two cases are, of course, not the same but, as this difference is not essential for the purpose of this study, it has been neglected hereafter.

The differential trace obtained by classic DPA will show a spike for the correct key even if the bus' status is not taken into account: the two power consumption groups in which curves are classified will still contain the same elements and at most an error on the spike's sign will be made (but this is irrelevant for the attack's purpose). On the other hand, the bus' status must be considered when observing simultaneously all four output bits for otherwise information could be lost.

When applying the Hamming variant, the power consumption of four bus lines is simultaneously analyzed. Values 0 to 15, corresponding to all possible combinations of ones and zeros on the four bus lines, could have been represented previously. To reduce the number of possibilities that must be studied, only values from 0 to 7 can be considered, as in our simplified model we postulate that power consumption due to transitions from 1 to 0 or from 0 to 1 are equivalent.

Let us erroneously classify PCCs according to the S-box output bits, neglecting the bus line's previous state. Two groups will result:

Table 2. Power Consumption Curve Classification According to S-Box Output.

High Hamming Weight	Low Hamming Weight
1111	0000
1110	0001
1101	0010
1011	0100
0111	1000

For the correct guess, we expect a spike amplitude proportional to 2.4 (3).

The bus lines on which a transition occur are the ones for which $S_i \oplus B_i = 1$, where S_i is the i-th S-box output bit and B_i is the i-th bus line. Therefore, to

correctly group curves according to power consumption, they should be classified according to the number of ones that result from $S_i \oplus B_i$.

Let us suppose that the value previously on the bus was 0011. If this information is neglected, the classification will yield what is reported in table 3.

Table 3. Power consumption curve classification having neglected the previous value on the bus.

High Hamming Weight			Low Hamming Weight		
S-box	bus	S-box $\oplus$ bus	S-box	bus	S-box $\oplus$ bus
1111	0011	1100	0000	0011	0011
1110	0011	1101	0001	0011	0010
1101	0011	1110	0010	0011	0001
1011	0011	1000	0100	0011	0111
0111	0011	0100	1000	0011	1011

From columns 3 and 6 in the table 3, having neglected the value previously on the bus,we infer that the spike's height will be proportional to:

$$< H >_{b=4,3} - < H >_{b=0,1} = \frac{10}{5} - \frac{10}{5} = 0,$$

therefore all useful information is lost.

However, in case the value previously present on the bus is 1000, or any other configuration in which three ones, or three zeros, are present, the spike's height would be proportional to:

$$< H >_{b=4,3} - < H >_{b=0,1} = \frac{13}{5} - \frac{7}{5} = \frac{6}{5},$$

therefore some, but not all, useful information is lost.

Only if the value on the bus had previously been 0000, or equivalently 1111, even by neglecting this value the spike's height would be proportional to 2.4.

If the previous bus state is unknown, in order to find the correct key guess by applying the Hamming integration, the attack must be run for all 8 possibilities. For the correct key, the following is observed:

- one differential curve with a high spike for the correct previous value on the bus
- four differential curves with a medium spike for a mistake on one bit (or three bits) on the previous value on the bus
- three flat differential curves for a mistake on two bits on the previous value on the bus

To be able to perform this attack, the state of the bus line before the instruction targeted by DPA must be constant or else a correct power curve classification cannot be performed. This value is constant when the previous operation concerning the bus is an opcode loading, or when data, constant but not dependent on the source code, transits on the bus.

Figure 5 shows a differential trace obtained by Hamming integration knowing the previous value on the bus compared to one resulting from SW-DPA on the S-box output bit that leaks the most.

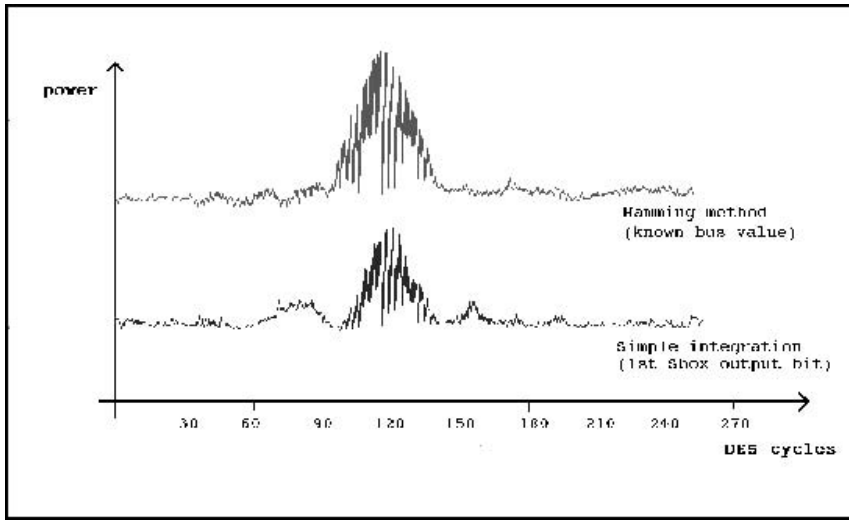


Fig. 5. Differential trace obtained on first S-box applying Hamming integration (upper curve) and applying SW-DPA on the first output bit (lower curve).

6 Conclusions

This paper shows that DPA can still be applied to chips on which hardware measures thought to provide DPA resistance had been implemented. The first attack proposed consists in applying a sliding window to the classic DPA described in [7]. The loss of synchronization caused by RPIs and the consequent increased number of necessary PCCs to perform the attack is calculated. The Hamming integration variant is slightly more complicated because, since 4 output bits are considered simultaneously, the previous value on the correspondent bus lines must be determined. As this information is usually unknown to the attacker, all possibilities must be examined experimentally. The advantage of the variant is a higher SNR, or sucess of the attack with a reduced number of PCC. As the second method involves a greater computational cost, it could be applied only to a restricted number of probable secret keys when the first method leaves some doubt.

To be secure, cryptographic devices should incorporate both hardware and software countermeasures to decrease the feasibility of side channel attacks. It

is also important to prove the validity of the countermeasures implemented, as heuristic assumptions are often not enough.

Acknowledgements

We would like to thank David Naccache for all his invaluable suggestions. We would also like to thank Olivier Benoit and Pascal Moitrel for the experimental results. Finally we would like to thank an anonymous referee for his detailed analysis of the paper which helped us provide the necessary improvements.

References

1. R. Anderson, M. Kuhn, *Tamper resistance – a cautionary note*, The second USENIX workshop on electronic commerce, pp. 1-11, 1996.
2. S. Chari, C. S. Jutla, J. R. Rao, P. Rohatgi, *Towards Sound Approaches to Counteract Power-Analysis Attacks*, Springer-Verlag, LNCS 1666, pp. 398–411, Crypto'99, 1999.
3. J.-S Coron, *Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems*, Springer-Verlag, LNCS 1717, pp. 292–302, CHES'99, 1999.
4. J.-S. Coron, P. Kocher, D. Naccache, Statistics and Secret Leakage, FC'00, to appear in the LNCS series, 2000.
5. P. N. Fahn, P. K. Pearson, *IPA: A New Class of Power Attacks*, Springer-Verlag, LNCS 1717, pp. 173–186, CHES'99, 1999.
6. L. Goubin, J. Patarin, *DES and Differential Power Analysis*, Springer-Verlag, LNCS 1717, pp. 158–172, CHES'99, 1999.
7. P. Kocher *Differential Power Analysis*, Springer-Verlag, LNCS 1666, pp. 388–397, Crypto'99, 1999.
8. P. Kocher *Timing Attacks on Implementation of Diffie-Hellman, RSA, DSS, and Other Systems*, Springer-Verlag, LNCS 1109, pp. 104–113, Crypto'96, 1996.
9. R. G. Lyons, *Understanding Digital Signal Processing*, Addison Wesley Longman, pp. 327–330, 1997.

Montgomery Multiplier and Squarer in GF(2^m)

Huapeng Wu

The Centre for Applied Cryptographic Research
Department of Combinatorics and Optimization
University of Waterloo, Waterloo, Canada
h3wu@cacr.math.uwaterloo.ca

Abstract. Montgomery multiplication in GF(2^m) is defined by $a(x)b(x)$ $r^{-1}(x)$ mod $f(x)$, where the field is generated by irreducible polynomial $f(x)$, $a(x)$ and $b(x)$ are two field elements in GF(2^m), and $r(x)$ is a fixed field element in GF(2^m). In this paper, first we present a generalized Montgomery multiplication algorithm in GF(2^m). Then by choosing $r(x)$ according to $f(x)$, we show that efficient architecture for bit-parallel Montgomery multiplier and squarer can be obtained for the fields generated with irreducible trinomials. Complexities in terms of gate counts and time propagation delay of the circuits are investigated and found to be comparable to or better than that of polynomial basis or weakly dual basis multiplier for the same class of fields.

1 Introduction

Finite field has applications in combinatorial designs, sequences, error-control codes, and cryptography. Finite field arithmetic operations have been paid much attention recently mainly because its use in cryptography, especially in elliptic curve cryptosystems. Research in this area has been characterized by its strong flavor of implementation both in software and in hardware. For example, fields of characteristic two are prevailingly used because a ground field operation can be readily implemented with a VLSI gate.[1]

In this paper, we first give a generalized Montgomery multiplication algorithm in GF(2^m). Then by choosing the fixed field element $r(x)$ according to the irreducible polynomial, we show that efficient multiplication and squaring architectures can be obtained using the generalized algorithm of Montgomery multiplication in GF(2^m). The implementation complexities in terms of the number of gates (equivalent to the number of ground field operations) and time propagation delay are lower than or as good as these of previously proposed multipliers for the same class of fields. The main implementation results are summarized in the two theorems.

[1] A multiplication operation in GF(2) can be implemented using an AND gate, while an addition operation in GF(2) can be implemented with an XOR gate.

Ç.K. Koç and C. Paar (Eds.): CHES 2000, LNCS 1965, pp. 264–276, 2000.

2 Preliminaries

Montgomery multiplication was first proposed for integer modular multiplication that can avoid trial division [2]. Later it was extended to finite field multiplication in GF(2^m) [1]. It was shown that the operation can be made simple if certain type of $r(x)$ is selected [1]. In the following, we give a brief review of the Montgomery multiplication in GF(2^m) proposed in [1].

Let $f(x)$ be the irreducible polynomial that defines the field GF(2^m) and $r(x)$ be a fixed element in GF(2^m). Since $\gcd(f(x), r(x)) = 1$, we can use the extended Euclidean algorithm to determine $f'(x)$ and $r'(x)$ that satisfy

$$r(x)r'(x) + f(x)f'(x) = 1. \tag{1}$$

Clearly $r'(x) = r^{-1}(x)$ is the inverse of $r(x)$. Given two field elements $a(x), b(x) \in$ GF(2^m), then an analogue for Montgomery multiplication in GF(2^m) can be given by [1]

$$c(x) = a(x)b(x)r^{-1}(x) \bmod f(x), \tag{2}$$

and an algorithm to compute (2) is shown below:

Algorithm 1. Montgomery multiplication in GF(2^m) [1]
Input: $a(x), b(x), r(x), f'(x)$
Output: $c(x) = a(x)b(x)r^{-1}(x) \bmod f(x)$
Step 1. $t(x) \Leftarrow a(x)b(x)$
Step 2. $u(x) \Leftarrow t(x)f'(x) \bmod r(x)$
Step 3. $c(x) \Leftarrow [t(x) + u(x)f(x)]/r(x)$

The correctness of Algorithm 1 can be easily checked. Note that

$$\begin{aligned}\deg[c(x)] &\leqslant \max\{\deg[t(x)], \deg[u(x)] + \deg[f(x)]\} - \deg[r(x)] \\ &= \max\{2m-2, \deg[r(x)] - 1 + m\} - \deg[r(x)] \\ &= \max\{2m - 2 - \deg[r(x)], m-1\}.\end{aligned}$$

Thus, to have $\deg[c(x)] \leqslant m-1$, the degree of $r(x)$ must be chosen not less than $m-1$. Since $f(x)$ and $f'(x)$ can be considered as constants, it is noted in [1] that efficient multiplication can be achieved if $r(x)$ is properly chosen. In fact, $r(x)$ was chosen to be the monomial x^m in [1] and Algorithm 1 is equivalent to a polynomial multiplication, two constant multiplications in GF(2^m) and one addition in GF(2^m).

3 Generalized Montgomery Multiplication in GF(2^m)

For bit-parallel realization of Montgomery multiplication in GF(2^m), we find that efficient multiplier architecture can be obtained if $r(x)$ is chosen according to the irreducible polynomial $f(x)$. For example, if the field is generated with a trinomial $f(x) = x^m + x^k + 1$, then $r(x)$ is selected to be the term of the second

low degree in the trinomial. This choice of $r(x) = x^k$ turns out to be very helpful in obtaining low complexity multiplier and squarer architectures. However, Algorithm 1 can not directly be used for these cases since k can be less than $m-1$. This leads us to consider a generalized form of Montgomery multiplication in GF(2^m). In the following, we first present a generalized Montgomery algorithm in GF(2^m), then compare it with Algorithm 1.

Algorithm 2. Generalized Montgomery multiplication in GF(2^m)
Input: $a(x), b(x), r(x), f(x), f'(x)$
Output: $c(x) = a(x)b(x)r^{-1}(x) \bmod f(x)$
Step 1. $t(x) \Leftarrow a(x)b(x)$
Step 2. $u(x) \Leftarrow t(x)f'(x) \bmod r(x)$
Step 3. $\tilde{c}(x) \Leftarrow [t(x) + u(x)f(x)]/r(x)$
Step 4. If $\deg(\tilde{c}) > m-1$, then $c(x) \Leftarrow \tilde{c}(x) \bmod f(x)$, else $c(x) \Leftarrow \tilde{c}(x)$

The correctness check for the algorithm is similar to that of Algorithm 1.

The degree range of $\tilde{c}(x)$ can be estimated. Since $0 \leqslant \deg[a(x)] \leqslant m-1$ and $0 \leqslant \deg[b(x)] \leqslant m-1$, it follows $0 \leqslant \deg[t(x)] \leqslant 2m-2$. Assume $\deg[r(x)] = k$, then from Step 2 we have $0 \leqslant \deg[u(x)] \leqslant k-1$. From Step 3, we have

$$\deg[\tilde{c}(x)] \leqslant \max\{2m-k-2, m-1\}. \tag{3}$$

When $\deg[r(x)] = k < m-1$, the degree of $\tilde{c}(x)$ is $2m-k-2$ and higher than $m-1$. In this case, one more step of modulo reduction (Step 4) is needed.

Compared to Algorithm 1, Algorithm 2 extends the degree range that $r(x)$ can be chosen from. Algorithm 1 can be considered as a specific case of Algorithm 2. For example, when $r(x)$ is chosen such that $\deg r(x) \geqslant m-1$, then $\tilde{c}(x)$ obtained at Step 3 in Algorithm 2 has a degree equal to or less than $m-1$. In this case, Step 4 will not be performed and the algorithm is the same as Algorithm 1. In fact, Algorithm 2 looks more similar to the original Montgomery algorithm [2] than Algorithm 1. This is because Step 4 in Algorithm 2 corresponds to the final subtraction step in the original Montgomery algorithm [2]. In Algorithm 1 this step has been omitted provided that some condition has been applied to how to choose $r(x)$.

4 Montgomery Multiplier in GF(2^m)

Consider the irreducible polynomial $f(x) = x^m + x^k + 1$, $\frac{m}{2} \leqslant k \leqslant m-1$, and the fixed field element $r(x) = x^k$ for the Montgomery multiplication in GF(2^m) (Algorithm 2). From the Extended Euclidean Algorithm, we obtain $r^{-1}(x) = 1 + x^{m-k}$ and $f'(x) = 1$ that satisfy

$$r(x)r^{-1}(x) + f(x)f'(x) = 1.$$

To solve the coefficients of the product $c(x)$ in terms of these of $a(x)$ and $b(x)$ and thus to find efficient multiplier architectures, we proceed with each step of Algorithm 2 as follows.

4.1 Step 1 in Algorithm 2

Let polynomial basis representations of $a(x)$ and $b(x)$ be given by $a(x) = \sum_{i=0}^{m-1} a_i x^i,\ a_i \in \mathrm{GF}(2)$, and $b(x) = \sum_{i=0}^{m-1} b_i x^i,\ b_i \in \mathrm{GF}(2)$, respectively. Then $t(x) = a(x)b(x)$ can be obtained as follows;

$$t(x) = a(x)b(x) = \sum_{i=0}^{2m-2} t_i x^i, \tag{4}$$

where t_i's are given by

$$t_i = \begin{cases} \displaystyle\sum_{j=0}^{i} a_j b_{i-j}, & 0 \leqslant i \leqslant m-1, \\ \displaystyle\sum_{j=i-m+1}^{m-1} a_j b_{i-j}, & m \leqslant i \leqslant 2m-2. \end{cases} \tag{5}$$

It can be seen that total m^2 bit multiplications and $(m-1)^2$ bit additions in GF(2) are required to compute $t_i,\ i = 0, 1, \ldots, 2m-2$. An implementation of (5) is straightforward, and the gate counts and time delays incurred with signals $t_i,\ i = 0, 1, \ldots, 2m-2$, are listed in Table 1. We denote the time delays of an AND gate and an XOR gate by T_A and T_X, respectively.

Table 1. Complexity and Time Delay Involved in Implementing $t(x)$.

Signal	# AND gates	# XOR gates	Time delay
$t_0 = a_0 b_0$	1	0	T_A
$t_1 = a_0 b_1 + a_1 b_0$	2	1	$T_A + T_X$
$t_2 = a_0 b_2 + a_1 b_1 + a_2 b_0$	3	2	$T_A + 2T_X$
$t_3 = a_0 b_3 + a_1 b_2 + a_2 b_1 + a_3 b_0$	4	3	$T_A + 2T_X$
$\vdots = \vdots$	$\vdots$	$\vdots$	$\vdots$
$t_{m-2} = a_0 b_{m-2} + \cdots + a_{m-2} b_0$	$m-1$	$m-2$	$T_A + \lceil \log_2 (m-1) \rceil T_X$
$t_{m-1} = a_0 b_{m-1} + \cdots + a_{m-1} b_0$	m	$m-1$	$T_A + \lceil \log_2 m \rceil T_X$
$t_m = a_1 b_{m-1} + \cdots + a_{m-1} b_1$	$m-1$	$m-2$	$T_A + \lceil \log_2 (m-1) \rceil T_X$
$\vdots = \vdots$	$\vdots$	$\vdots$	$\vdots$
$t_{2m-3} = a_{m-2} b_{m-1} + a_{m-1} b_{m-2}$	2	1	$T_A + T_X$
$t_{2m-2} = a_{m-1} b_{m-1}$	1	0	T_A
Total:	m^2	$(m-1)^2$	$T_A + \lceil \log_2 m \rceil T_X$

In the following, we will solve the rest three steps of Algorithm 2 and show that they can be realized at one single implementation step.

4.2 Step 2 in Algorithm 2

Substitute $t(x)$ in this step using (4)

$$\begin{aligned} u(x) &= t(x)f'(x) \bmod r(x) \\ &= t_0 + t_1 x + t_2 x^2 + \cdots + t_{2m-2} x^{2m-2} \bmod x^k \\ &= t_0 + t_1 x + t_2 x^2 + \cdots + t_{k-1} x^{k-1}. \end{aligned} \tag{6}$$

Clearly, the degree of $u(x)$ is not higher than that of $r(x)$. If $r(x)$ is chosen to have a low degree then we have a simple $u(x)$.

4.3 Step 3 in Algorithm 2

Define

$$t_L(x) \triangleq t_0 + t_1 x + t_2 x^2 + \cdots + t_{k-1} x^{k-1}, \tag{7}$$

and

$$t_H(x) \triangleq t_k + t_{k+1} x + \cdots + t_{2m-2} x^{2m-k-2}. \tag{8}$$

From (4) (7) and (8), it can be seen that

$$t(x) = t_L(x) + x^k t_H(x), \tag{9}$$

and from (6) and (7) it follows

$$u(x) = t_L(x). \tag{10}$$

Substitute $t(x)$ and $u(x)$ in Step 3 with (9) and (10), respectively, and note that $f(x) = x^m + x^k + 1$, we have

$$\begin{aligned} \tilde{c}(x) &= [t(x) + u(x)f(x)]/r(x) \\ &= [t_L(x) + x^k t_H(x) + t_L(x)(x^m + x^k + 1)]/x^k \\ &= [x^k t_H(x) + x^k(x^{m-k} + 1)t_L(x)]/x^k \\ &= t_H(x) + x^{m-k} t_L(x) + t_L(x). \end{aligned} \tag{11}$$

When $k = m - 1$, from (3) we have $\deg \tilde{c}(x) \leqslant m - 1$. Clearly, in this case the degree of $\tilde{c}(x)$ has already been reduced to the proper range and Step 4 in Algorithm 2 is not necessary.

Extend each of the three terms at the right hand side of (11) for the case of $k = m - 1$, and from (7) and (8) we have

$$\begin{aligned} t_H(x) &= t_{m-1} + t_m x + \cdots + t_{2m-2} x^{m-1}, \\ x t_L(x) &= t_0 x + t_1 x^2 + \cdots + t_{m-2} x^{m-1}, \\ t_L(x) &= t_0 + t_1 x + \cdots + t_{m-2} x^{m-2}. \end{aligned} \tag{12}$$

Then by comparing (12) with (11), and note $\tilde{c}(x) = c(x) = \sum_{i=0}^{m-1} c_i x^i$, we can write c_i as follows

$$\begin{aligned} c_0 &= t_0 + t_{m-1}, \\ c_1 &= t_0 + t_1 + t_m, \\ c_2 &= t_1 + t_2 + t_{m+1}, \\ &\vdots \\ c_{m-2} &= t_{m-3} + t_{m-2} + t_{2m-3}, \\ c_{m-1} &= t_{m-2} + t_{2m-2}. \end{aligned}$$

Rewrite the above expressions as

$$c_i = \begin{cases} t_0 + t_{m-1}, & i = 0, \\ t_{i-1} + t_i + t_{m-1+i}, & i = 1, 2, \ldots, m-2, \\ t_{m-2} + t_{2m-2}, & i = m-1. \end{cases} \tag{13}$$

It can be seen from (13) that each c_i can be obtained with 2 bit additions in GF(2), except that c_0 and c_{m-1} require one bit operation each. Thus, a bit-parallel realization of (13) needs $2m-2$ XOR gates. Since the maximal number of terms on the right hand side of each equation in (13) is three, the maximal time propagation delay is $2T_X$.

When $\frac{m}{2} \leqslant k < m-1$, from (3) we have $\deg \tilde{c}(x) > m-1$. In this case, a step of modulo reduction is still needed.

4.4 Step 4 in Algorithm 2

From (8) we divide $t_H(x)$ into two parts: $t_H(x) = t_H^{(1)}(x) + t_H^{(2)}(x)$, where

$$t_H^{(1)}(x) \triangleq t_k + t_{k+1}x + \cdots + t_{k+m-1}x^{m-1}, \tag{14}$$

and

$$t_H^{(2)}(x) \triangleq t_{k+m}x^m + t_{k+m+1}x^{m+1} + \cdots + t_{2m-2}x^{2m-k-2}. \tag{15}$$

Substitute $t_H(x)$ in (11) with $t_H^{(1)}(x) + t_H^{(2)}(x)$ and note that $c(x) = \tilde{c}(x) \bmod f(x)$, we have

$$\begin{aligned} c(x) &= \tilde{c}(x) \bmod f(x) \\ &= [t_H^{(1)}(x) + t_H^{(2)}(x) + x^{m-k}t_L(x) + t_L(x)] \bmod f(x) \\ &= t_H^{(1)}(x) + x^{m-k}t_L(x) + t_L(x) + [t_H^{(2)}(x)] \bmod f(x). \end{aligned} \tag{16}$$

Apply the modulo operation to each term on the right hand side of (15), it follows

$$\begin{aligned} t_{k+m}x^m \bmod f(x) &= t_{k+m}(1+x^k), \\ t_{k+m+1}x^{m+1} \bmod f(x) &= t_{k+m+1}(x+x^{k+1}), \\ &\vdots \\ t_{2m-2}x^{2m-k-2} \bmod f(x) &= t_{2m-2}(x^{m-k-2}+x^{m-2}). \end{aligned}$$

Adding the above $m-k-1$ equations together, we obtain

$$t_H^{(2)}(x) \bmod f(x) = \sum_{i=0}^{m-k-2} t_{m+k+i}x^i + \sum_{i=k}^{m-2} t_{m+i}x^i.$$

Split $t_H^{(2)}(x)$ into two parts:

$$t_H^{(2,1)}(x) \bmod f(x) \stackrel{\triangle}{=} \sum_{i=0}^{m-k-2} t_{m+k+i}x^i, \tag{17}$$

$$t_H^{(2,2)}(x) \bmod f(x) \stackrel{\triangle}{=} \sum_{i=k}^{m-2} t_{m+i}x^i. \tag{18}$$

Substitute $t_H^{(2)}(x)$ with $t_H^{(2,1)}(x)+t_H^{(2,2)}(x)$ in (16) and note that $c(x) = \sum_{i=0}^{m-1} c_i x^i$, it follows

$$\sum_{i=0}^{m-1} c_i x^i = t_L(x) + x^{m-k}t_L(x) + t_H^{(1)}(x) + t_H^{(2,1)}(x) + t_H^{(2,2)}(x). \tag{19}$$

Rewrite the equations (7), (14), (17), (18) and extend the term $x^{m-k}t_L(x)$ using (7), we have the following five equations for the five terms on the right hand side of (19), respectively:

$$\begin{array}{lrl} (a) & t_L(x) = t_0 + t_1x + t_2x^2 + \cdots + t_{k-1}x^{k-1} & [0, k-1] \\ (b) & x^{m-k}t_L(x) = t_0x^{m-k} + t_1x^{m-k+1} + \cdots + t_{k-1}x^{m-1} & [m-k, m-1] \\ (c) & t_H^{(1)}(x) = t_k + t_{k+1}x + \cdots + t_{k+m-1}x^{m-1} & [0, m-1] \\ (d) & t_H^{(2,1)}(x) = t_{m+k} + t_{m+k+1}x + \cdots + t_{2m-2}x^{m-k-2} & [0, m-k-2] \\ (e) & t_H^{(2,2)}(x) = t_{m+k}x^k + t_{m+k+1}x^{k+1} + \cdots + t_{2m-2}x^{m-2} & [k, m-2] \end{array} \tag{20}$$

The last column in the above array is the degree range of the terms on the right-hand side of each equation. Now we are ready to solve the coefficients c_i by comparing (19) with (20).

In the following, we consider three cases:

Case 1: If $\frac{m+1}{2} < k < m-1$. We have $m-k-2 < m-k < k-1 < k$. By comparing (19) with (20), we can solve c_i's (the coefficient of the term x^i in $c(x)$). When $0 \leqslant i \leqslant m-k-2$, it can be seen from (20) that c_i takes on the terms from equations $(a), (c)$ and (d). When $i = m-k-1$, c_{m-k-1} has only two terms, one is from equation (a) and the other from (c). When i runs through from $m-k$ to $k-1$, c_i picks up the terms from equations $(a), (b)$ and (c). When $k \leqslant i \leqslant m-2$, c_i has three terms: one from equation (b), one from (c) and the other from (e). Finally, c_{m-1} has two terms from equations (b) and (c), respectively. We can write c_i's as follows

$$\begin{array}{rlllll}
 & (a) & (b) & (c) & (d) & (e) \\
c_0 = & t_0 & & +t_k & +t_{k+m} & \\
c_1 = & t_1 & & +t_{k+1} & +t_{k+m+1} & \\
 & \vdots\ \vdots & & \vdots & \vdots & \\
c_{m-k-2} = & t_{m-k-2} & & +t_{m-2} & +t_{2m-2} & \\
c_{m-k-1} = & t_{m-k-1} & & +t_{m-1} & & \\
c_{m-k} = & t_{m-k} & +t_0 & +t_m & & \\
 & \vdots\ \vdots & \vdots & \vdots & & \\
c_{k-1} = & t_{k-1} & +t_{2k-m-1} & +t_{2k-1} & & \\
c_k = & & +t_{2k-m} & +t_{2k} & & +t_{k+m} \\
 & \vdots & \vdots & \vdots & & \vdots \\
c_{m-2} = & & +t_{k-2} & +t_{m+k-2} & & +t_{2m-2} \\
c_{m-1} = & & +t_{k-1} & +t_{m+k-1} & &
\end{array} \tag{21}$$

where all the terms at the column $(a), (b), \ldots$ are from the equations $(a), (b), \ldots$ in (20), respectively. Now we can estimate the complexity to obtain the coefficients of the product from t_i's. From (21), it can be seen that $2m-2$ bit addition in GF(2) are used to solve c_i's. The longest time delay to generate c_i from t_i is $2T_X$.

Case 2. If $k = \frac{m+1}{2}$. We have $m-k-2 < m-k = k-1 < k$. By comparing (19) to (20) the coefficients of $c(x)$ can be written as follows

$$\begin{array}{rlllll}
 & (a) & (b) & (c) & (d) & (e) \\
c_0 = & t_0 & & +t_k & +t_{k+m} & \\
c_1 = & t_1 & & +t_{k+1} & +t_{k+m+1} & \\
 & \vdots\ \vdots & & \vdots & \vdots & \\
c_{m-k-2} = & t_{k-3} & & +t_{2k-3} & +t_{2m-2} & \\
c_{m-k-1} = & t_{k-2} & & +t_{2k-2} & & \\
c_{m-k} = & t_{k-1} & +t_0 & +t_{2k-1} & & \\
c_k = & & +t_1 & +t_{2k} & & t_{k+m} \\
 & \vdots & \vdots & \vdots & & \vdots \\
c_{m-2} = & & +t_{k-2} & +t_{m+k-2} & & +t_{2m-2} \\
c_{m-1} = & & +t_{k-1} & +t_{m+k-1} & &
\end{array} \tag{22}$$

It can be seen that a realization of the above expressions requires $2m-2$ ground field operations. Since the most terms to sum up for each c_i is three, the maximal time delay is $2T_X$.

Case 3. If $k=\frac{m}{2}$. We have $m-k-2=k-2<k-1<m-k=k$. The coefficients of the Montgomery product can be obtained from (19) and (20) as follows:

$$\begin{array}{rlllll} & (c) & (a) & (d) & (b) & (e) \\ c_0 = t_k & & +(t_0 & +t_{k+m}) & & \\ c_1 = t_{k+1} & & +(t_1 & +t_{k+m+1}) & & \\ \vdots & \vdots & \vdots & \vdots & & \\ c_{k-2} = t_{m-2} & & +(t_{k-2} & +t_{2m-2}) & & \\ c_{k-1} = t_{m-1} & & +t_{k-1} & & & \\ c_k = t_m & & & & +(t_0 & +t_{k+m}) \\ c_{k+1} = t_{m+1} & & & & +(t_1 & +t_{k+m+1}) \\ \vdots & \vdots & & & \vdots & \vdots \\ c_{m-2} = t_{m+k-2} & & & & +(t_{k-2} & +t_{2m-2}) \\ c_{m-1} = t_{m+k-1} & & & & +t_{k-1} & \end{array} \tag{23}$$

Note that the resued partial sums are put in the brackets. Then it can be seen from (23) that $2m-2-(k-1)=\frac{3}{2}m-1$ bit additions in GF(2) are required to compute $c_0,\ldots,c_{m-1}$ from $t_0,\ldots,t_{2m-2}$. The time delay incurred here is still $2T_X$.

4.5 Bit-Parallel Multiplier Architecture

From the above discussion, it can be seen that a bit-parallel Montgomery multiplication in GF(2^m) is decided by (5) and one of the expressions (21), (22) and (23). A diagram for the bit-parallel multiplier architecture is shown in Fig. 1. The upper two modules (one all-AND-gate circuits and one all-XOR-gate circuits) are used to perform polynomial multiplication (Step 1 in Algorithm 2), while the module at the bottom (all-XOR-gate circuits) corresponds to the implementation of Steps 2 to 4 in Algorithm 2.

It can be seen from Table 1 that m^2 AND agtes and $(m-1)^2$ XOR gates are required for generating t_i. Then the coefficients of $c(x)$ can be generated from t_i using one of (21), (22) and (23). Obviously, the total number of gates required are

$$\begin{array}{rl} m^2 & \text{AND gates,} \\ m^2-1 & \text{XOR gates,} \end{array}$$

if the irreducible trinomial is $f(x)=x^m+x^k+1$, $\frac{m}{2}<k\leqslant m-1$.

When $f(x)=x^m+x^{\frac{m}{2}}+1$, the complexity is only

$$\begin{array}{rl} m^2 & \text{AND gates,} \\ m^2-\frac{m}{2} & \text{XOR gates.} \end{array}$$

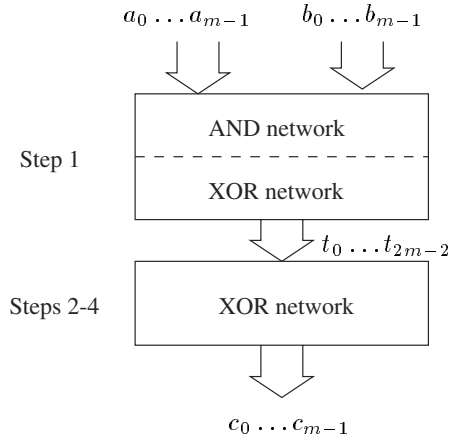


Fig. 1. Bit-Parallel Montgomery Multiplier Architecture when $f(x) = x^m + x^k + 1$ and $r(x) = x^k$.

Total time delay of the multiplier is not greater than $T_A + (\lceil \log_2 m \rceil + 2)\, T_X$. In many cases the total propagation delay is less than the above bound. Note from the Table 1 that the time delay incurred with different t_i is different. In fact, circuits for generating t_i has a time delay $\lceil \log_2(i+1) \rceil T_X$ if $i \leqslant m-1$, and $\lceil \log_2(2m-i-1) \rceil T_X$ if $i \geqslant m$. From (13), (21), (22) and (23), it can be seen that most c_i's is a sum of three terms. Write them as $c_i = t_{i1} + t_{i2} + t_{i3}$, where we assume that the time delays for generating t_{i1}, t_{i2}, and t_{i3} are d_1, d_2, and d_3, respectively. If $d_1 \leqslant d_2 \leqslant d_3$, then it can be seen that the propagation delay for generating c_i depends on d_2 and d_3 if the circuit is designed using

$$c_i = (t_{i1} \oplus t_{i2}) \oplus t_{i3}.$$

The time delay incurred with the above logic equation for generating c_i is

$$T_{c_i} = \max\{d_2 + 2, d_3 + 1\}.$$

Using this method, we search and find the maximal time delays incurred with the expressions (13), (21), (22) and (23).

4.6 Complexity Results and Example

We summarize the implementation results on Montgomery multiplier in GF(2^m) as follows:

Theorem 1. *Let the finite field GF(2^m) be defined by irreducible trinomial $f(x) = x^m + x^k + 1$, $\frac{m}{2} \leqslant k \leqslant m-1$. Then a bit-parallel Montgomery multiplier in GF(2^m) can be constructed from the expression (5), and one of the expressions (13), (21), (22) and (23). The complexity and time propagation delay are given as follows.*

1. *The complexity is* m^2 *AND gates and* m^2-1 *XOR gates. The incurred time delay is* $T_A+(\lceil\log_2(m-2)\rceil+2)T_X$, *if* $k=m-1$.
2. *The complexity is* m^2 *AND gates and* m^2-1 *XOR gates. The incurred time delay is* $T_A+(\lceil\log_2(m-\frac{k}{2})\rceil+2)T_X$, *if* $\frac{m+1}{2}\leqslant k\leqslant m-1$.
3. *The complexity is* m^2 *AND gates and* m^2-1 *XOR gates. The incurred time delay is* $T_A+(\lceil\log_2 k\rceil+2)T_X$, *if* $k=\frac{m+1}{2}$.
4. *The complexity is* m^2 *AND gates and* $m^2-\frac{m}{2}$ *XOR gates. The incurred time delay is* $T_A+(\lceil\log_2(m-1)+1)T_X$, *if* $k=\frac{m}{2}$.

4.7 Montgomery Squarer in GF(2^m)

When Algorithm 2 is used for squaring operation, only the first step needs to be changed. We rewrite Algorithm 2 for Montgomery squaring in GF(2^m) as follows

Algorithm 3. Generalized Montgomery squaring in GF(2^m)
Input: $a(x), r(x), f(x), f'(x)$
Output: $c(x)=a^2(x)r^{-1}(x) \bmod f(x)$
Step 1. $t(x)\Leftarrow a^2(x)$
Step 2. $u(x)\Leftarrow t(x)f'(x) \bmod r(x)$
Step 3. $\tilde{c}(x)\Leftarrow[t(x)+u(x)f(x)]/r(x)$
Step 4. If $\deg(\tilde{c})>m-1$, then $c(x)\Leftarrow\tilde{c}(x) \bmod f(x)$, else $c(x)\Leftarrow\tilde{c}(x)$

With the same selection of the field $f(x)=x^m+x^k+1$ and the fixed element $r(x)=x^k$, we proceed with Algorithm 3 step by step.

Step 1. From $t(x)=a^2(x)$, we have

$$\sum_{i=0}^{m-1}a_ix^{2i}=\sum_{i=0}^{2m-2}t_ix^i.$$

It can be seen from the above expression

$$t_i=\begin{cases}a_{\frac{i}{2}}, & i=0,2,\ldots,2m-2;\\ 0, & i=1,3,\ldots,2m-3.\end{cases}\tag{24}$$

Not like multiplication, there is no bit operations needed here to obtain t_i.

Step 2-4. These three steps are very similar to these in Algorithm 2, and many intermediate results obtained in the last section can also be used here.

In the following we only consider the case that $k=m-1$ and m is even. For the other cases the deduction is similar. From (12) and (24), we have

$$\begin{aligned}&(a)\quad t_L(x)=a_0+a_1x^2+a_2x^4+\cdots+a_{\frac{m-2}{2}}x^{m-2} && [0,m-2]\\ &(b)\ xt_L(x)=a_0x+a_1x^3+\cdots+a_{\frac{m-2}{2}}x^{m-1} && [1,m-1]\\ &(c)\quad t_H(x)=a_{\frac{m}{2}}x+a_{\frac{m+2}{2}}x^3+\cdots+a_{m-1}x^{m-1} && [1,m-1]\end{aligned}\tag{25}$$

Note that the expression (a) in (25) has only even power terms and (b) and (c) have only odd power terms. Comparing (25) to (11) and note $c(x) = \tilde{c}(x)$ when $k = m - 1$, the coefficients c_i can be obtained as follows

$$c_i = \begin{cases} a_{\frac{i}{2}}, & i = 0, 2, \ldots, m-2; \\ a_{\frac{i-1}{2}} + a_{\frac{m+i-1}{2}}, & i = 1, 3, \ldots, m-1. \end{cases} \tag{26}$$

It can be seen that $\frac{m}{2}$ bit additions in GF(2) are required to compute c_i using (26). Then we know that to implement a bit-parallel Montgomery squarer needs only $\frac{m}{2}$ XOR gates. The time delay for this Montgomery squarer is equivalent to the delay of one XOR gate T_X.

The implementation results can be summarized as follows:

Theorem 2. *Let the finite field GF(2^m) be defined by irreducible trinomial $f(x) = x^m + x^k + 1$, $\frac{m}{2} \leqslant k \leqslant m-1$. Then a bit-parallel Montgomery squarer in GF(2^m) can be built with $\left\lceil \frac{m-1}{2} \right\rceil$ XOR gates and the time propagation delay is T_X.*

5 Comparison

Table 2. Comparison of Bit-Parallel Multipliers.

Proposals	# AND	# XOR	Time delay
$f(x) = x^m + x + 1$			
Wu, Hasan and Blake [6]	m^2	$m^2 - 1$	$T_A + (\lceil \log_2 m \rceil + 1)T_X$
Sunar and Koc [3]	m^2	$m^2 - 1$	$T_A + (\lceil \log_2 m \rceil + 1)T_X$
Wu [5]	m^2	$m^2 - 1$	$T_A + (\lceil \log_2 (m-2) \rceil + 2)T_X$
Presented here	m^2	$m^2 - 1$	$T_A + (\lceil \log_2 (m-2) \rceil + 2)T_X$
$f(x) = x^m + x^k + 1,\ 1 < k < \frac{m}{2}$			
Wu, Hasan and Blake [6]	m^2	$m^2 - 1$	$T_A + \left(\left\lceil \log_2 \left\lceil \frac{m+k-1}{2} \right\rceil \right\rceil + 2 \right) T_X$
Sunar and Koc [3]	m^2	$m^2 - 1$	$T_A + (\lceil \log_2 m \rceil + 2)T_X$
Wu [5]	m^2	$m^2 - 1$	$T_A + (\lceil \log_2 (m-1) \rceil + 2)T_X$
Presented here	m^2	$m^2 - 1$	$T_A + (\lceil \log_2 (m - \frac{k}{2}) \rceil + 2)T_X$
$f(x) = x^m + x^{\frac{m}{2}} + 1$			
Wu, Hasan and Blake [6]	m^2	$m^2 - \frac{m}{2}$	$T_A + \left\lceil \log_2 \left(m + 2 \left\lceil \frac{m}{4} \right\rceil \right) \right\rceil T_X$
Sunar and Koc [3]	m^2	$m^2 - \frac{m}{2}$	$T_A + (\lceil \log_2 m \rceil + 1)T_X$
Wu [5]	m^2	$m^2 - \frac{m}{2}$	$T_A + (\lceil \log_2 (m-1) \rceil + 1)T_X$
Presented here	m^2	$m^2 - \frac{m}{2}$	$T_A + (\lceil \log_2 (m-1) \rceil + 1)T_X$

Table 2 gives a comparison of four different implementations of bit-parallel multiplier in the same class of fields. Note that we consider the fields generated

with two irreducible reciprocal trinomials are the same. The bit-parallel multiplier proposed by Wu, Hasan and Blake uses weakly dual basis (WDB) [6].[2] Sunar and Koc presented all trinomial Mastrovito multiplier using polynomial basis. The polynomial basis multiplier proposed in [5] has a different architecture from the Mastrovito multiplier.

It can be seen that all the multipliers achieve the same complexity in terms of the numbers of AND and XOR gates. The time propagation delay incurred with the multiplier presented here comparable to that of the previously proposed multipliers.

Table 3. Comparison of Polynomial Basis Bit-Parallel Squarers.

Proposals	# XOR	Time delay
$f(x) = x^m + x^k + 1$, where $m + k$ odd.		
Wu [5]	$\frac{m+k-1}{2}$	$2T_X$
Presented here	$\left\lceil \frac{m-1}{2} \right\rceil$	T_X
$f(x) = x^m + x^k + 1$, where both m and k are odd.		
Wu [5]	$\frac{m-1}{2}$	T_X
Presented here	$\frac{m-1}{2}$	T_X

It can be seen from Table 3 that Montgomery squarer has both lower complexity and lower time propagation delay for the case that $m+k$ is odd, compared to the regular polynomial basis squarer presented in [5].

References

1. C. K. Koc and T. Acar. Montgomery multiplication in GF(2^k). *Designs, Codes and Cryptography*, 14:57–69, 1998.
2. P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44:519–521, 1985.
3. B. Sunar and C. K. Koc. Mastrovito multiplier for all trinomials. *IEEE Trans. Comput.*, 48(5):522–527, 1999.
4. M. Wang and I. F. Blake. Bit serial multiplication in finite fields. *SIAM Discrete Mathematics*, 3(1):140–148, 1990.
5. H. Wu. Low-complexity arithmetic in finite field using polynomial basis. In *CHES'99*, pages 357–371. Springer-Verlag, 1999.
6. H. Wu, M. A. Hasan, and I. F. Blake. Low complexity weakly dual basis bit-parallel multiplier over finite fields. *IEEE Trans. Comput.*, 47(11):1223–1234, November 1998.

[2] A PB bit-parallel multiplier can be readily made by adding a basis conversion module to both the input and the output ends. By a theorem proposed in [4], when the field is generated with an irreducible trinomial, the coefficients of a field element in WDB is nothing but a permutation of the coefficients of the element in PB. Thus a weakly dual basis bit-parallel multiplier proposed in [4] can be used as a polynomial basis bit-parallel multiplier without additional gates and time delay.

A Scalable and Unified Multiplier Architecture for Finite Fields $GF(p)$ and $GF(2^m)$*

Erkay Savaş, Alexandre F. Tenca, and Çetin K. Koç

Electrical & Computer Engineering
Oregon State University, Corvallis, Oregon 97331
{savas,tenca,koc}@ece.orst.edu

Abstract. We describe a scalable and unified architecture for a Montgomery multiplication module which operates in both types of finite fields $GF(p)$ and $GF(2^m)$. The unified architecture requires only slightly more area than that of the multiplier architecture for the field $GF(p)$. The multiplier is scalable, which means that a fixed-area multiplication module can handle operands of any size, and also, the wordsize can be selected based on the area and performance requirements. We utilize the concurrency in the Montgomery multiplication operation by employing a pipelining design methodology. The upper limit on the precision of the scalable and unified Montgomery multiplier is dictated only by the available memory to store the operands and internal results, and the module is capable of performing infinite-precision Montgomery multiplication in both types of finite fields.
Keywords: Prime fields, binary extension fields, multiplication, Montgomery multiplication, scalability, hardware implementation.

1 Introduction

The basic arithmetic operations (i.e., addition, multiplication, and inversion) in prime and binary extension fields, $GF(p)$ and $GF(2^m)$, have several applications in cryptography, such as decipherment operation of RSA algorithm [17], Diffie-Hellman key exchange algorithm [3], elliptic curve cryptography [7,12], and the Digital Signature Standard including the Elliptic Curve Digital Signature Algorithm [15]. The most important of these three arithmetic operations is the field multiplication operation since it is the core operation in many cryptographic functions.

The Montgomery multiplication algorithm [13] is an efficient method for doing modular multiplication with an odd modulus. The Montgomery multiplication algorithm is very useful for obtaining fast software implementations of the multiplication operation in prime fields $GF(p)$. The algorithm replaces division operation with simple shifts, which are particularly suitable for implementation on both general-purpose computers and application specific hardware.

* Readers should note that Oregon State University filed a patent application containing this work to the US Patent and Trademark Office.

Ç.K. Koç and C. Paar (Eds.): CHES 2000, LNCS 1965, pp. 277–292, 2000.

The Montgomery multiplication operation has been extended to the finite field $GF(2^k)$ in [9]. Efficient software implementations of the multiplication operation in $GF(2^k)$ can be obtained using this algorithm, particularly when the irreducible polynomial generating the field is chosen arbitrarily. The main idea of the architecture proposed in this paper is based on the observation that the Montgomery multiplication algorithm for both fields $GF(p)$ and $GF(2^k)$ are essentially the same algorithm. The proposed unified architecture performs the Montgomery multiplication in the field $GF(p)$ generated by an arbitrary prime p and in the field $GF(2^m)$ generated by an arbitrary irreducible polynomial $p(x)$. We show that a unified multiplier performing the Montgomery multiplication operation in the fields $GF(p)$ and $GF(2^k)$ can be designed at a cost only slightly higher than the multiplier for the field $GF(p)$, providing significant savings when both types of multipliers are needed.

Several variants of the Montgomery multiplication algorithm [16,10,2] have been proposed to obtain more efficient software implementations on specific processors. Various hardware implementations of the Montgomery multiplication algorithm for limited precision operands are also reported [2,16,4]. On the other hand, implementations utilizing high-radix modular multipliers have also been proposed [16,11,18]. Advantages and disadvantages of using high-radix representation have been discussed in [21,20]. Because high-radix Montgomery multiplication designs introduce longer critical paths and more complex circuitry, these designs are less attractive for hardware implementations.

A scalable Montgomery multiplier design methodology for $GF(p)$ was introduced in [20] in order to obtain hardware implementations. This design methodology allows to use a fixed-area modular multiplication circuit for performing multiplication of unlimited precision operands. The design tradeoffs for best performance in a limited chip area were also analyzed in [20]. We use the design approach as in [20] to obtain a scalable hardware module. Furthermore, the scalable multiplier described in this paper is capable of performing multiplication in both types of finite fields $GF(p)$ and $GF(2^k)$, i.e., it is a scalable and unified multiplier.

The main contributions of this paper are summarized below.

- We show that a unified architecture for multiplication module which operates both in $GF(p)$ and $GF(2^m)$ can be designed easily without compromising scalability, time and area efficiency.
- We analyze the design tradeoffs such as the effect of word length, the number of the pipeline stages, and the chip area by supplying implementation results obtained by Mentor graphics synthesis tools.

We start with a short discussion of scalability in §2 and explain the main idea behind the unified multiplier architecture in §3. We then present the methodology to perform the Montgomery multiplication operation in both types of finite fields using the unified architecture. We give the original and modified definitions of Montgomery algorithm for $GF(p)$ and $GF(2^m)$ in §4. We discuss concurrency in the Montgomery multiplication and show the methodology to design a pipeline module utilizing the concurrency in §5. We present the processing unit and the

modifications needed to make the unit operate in prime and binary extension fields in §6. In §7, we discuss the area/time tradeoffs and suitable choices for word lengths, the number of pipeline stages, and typical chip area requirements. Finally, we summarize our conclusions in §8.

2 Scalable Multiplier Architecture

An arithmetic unit is called scalable if it can be reused or replicated in order to generate long-precision results independently of the data path precision for which the unit was originally designed. To speed up the multiplication operation, various dedicated multiplier modules were developed in [18,1,14]. These designs operate over a fixed finite field. For example, the multiplier designed for 155 bits [1] cannot be used for any other field of higher degree. When a need for a multiplication of larger precision arises, a new multiplier must be designed. Another way to avoid redesigning the module is to use software implementations and fixed precision multipliers. However, software implementations are inefficient in utilizing inherent concurrency of the multiplication because of the inconvenient pipeline structure of the microprocessors being used. Furthermore, software implementations on fixed digit multipliers are more complex and require excessive amount of effort in coding. Therefore, a scalable hardware module specifically tailored to take advantage of the concurrency of the Montgomery multiplication algorithm becomes extremely attractive.

3 Unified Multiplier Architecture

Even though prime and binary extension fields, $GF(p)$ and $GF(2^m)$, have dissimilar properties, the elements of either field are represented using almost the same data structures inside the computer. In addition, the algorithms for basic arithmetic operations in both fields have structural similarities allowing a unified module design methodology. For example, the steps of the Montgomery multiplication algorithm for binary extension field $GF(2^m)$ given in [9] only slightly differs from those of the integer Montgomery multiplication algorithm [13,10]. Therefore, a scalable arithmetic module, which can be adjusted to operate in both types of fields, is feasible, provided that this extra functionality does not lead to an excessive increase in area or a dramatic decrease in speed. In addition, designing such a module must require only a small amount of extra effort and no major modification in control logic of the circuit.

Considering the amount of time, money and effort that must be invested in designing a multiplier module or more generally speaking a cryptographic coprocessor, a scalable and unified architecture which can perform arithmetic in two commonly used algebraic fields is definitely beneficial. In this paper, we show the method to design a Montgomery multiplier that can be used for both types of fields following the design methodology presented in [20]. The proposed unified architecture is obtained from the scalable architecture given in [20] after

minor modifications. The propagation time is unaffected and the increase in chip area is insignificant.

4 Montgomery Multiplication

Given two integers A and B, and the prime modulus p, the Montgomery multiplication algorithm computes

$$C = \mathsf{MonMul}(A, B) = A \cdot B \cdot R^{-1} \pmod{p} , \tag{1}$$

where $R = 2^m$ and $A, B < p < R$, and p is an m-bit number. The original algorithm works for any modulus n provided that $\gcd(n, R) = 1$. In this paper, we assume that the modulus is a prime number, thus, we perform multiplication in the field defined by this prime number. This issue is also relevant when the algorithm is defined for the binary extension fields.

The Montgomery multiplication algorithm relies on a different representation of the finite field elements. The field element $A \in GF(p)$ is transformed into another element $\bar{A} \in GF(p)$ using the formula $\bar{A} = A \cdot R \pmod{p}$. The number $\bar{A}$ is called Montgomery image of the element, or $\bar{A}$ is said to be in the Montgomery domain. Given two elements in the Montgomery domain $\bar{A}$ and $\bar{B}$, the Montgomery multiplication computes

$$\bar{C} = \bar{A}\cdot\bar{B}\cdot R^{-1} \pmod{p} = (A\cdot R)\cdot(B\cdot R)\cdot R^{-1} \pmod{p} = C\cdot R \pmod{p} , \tag{2}$$

where $\bar{C}$ is again in the Montgomery domain. The transformation operations between the two domains can also be performed using the MonMul function as

$$\begin{aligned}
\bar{A} &= \mathsf{MonMul}(A, R^2) = A \cdot R^2 \cdot R^{-1} = A \cdot R \pmod{p} , \\
\bar{B} &= \mathsf{MonMul}(B, R^2) = B \cdot R^2 \cdot R^{-1} = B \cdot R \pmod{p} , \\
C &= \mathsf{MonMul}(\bar{C}, 1) = C \cdot R \cdot R^{-1} = C \pmod{p} .
\end{aligned}$$

Provided that $R^2 \pmod{p}$ is precomputed and saved, we need only a single MonMul operation to carry out each of these transformations. However, because of these transformation operations, performing a single modular multiplication using MonMul might not be advantageous. The advantage of the Montgomery multiplication becomes much more apparent in applications requiring multiplication-intensive calculations, e.g., modular exponentiation or elliptic curve point operations. In order to exploit this advantage, all arithmetic operations are performed in the Montgomery domain, including the inversion operation [6,19].

Below, we give bitwise Montgomery multiplication algorithm for obtaining $C := ABR^{-1} \pmod{p}$, where $A = (a_{m-1}, \ldots, a_1, a_0)$ and $C = (c_m, \ldots, c_1, c_0)$.

Input:	$A, B \in GF(p)$ and $m = \lceil \log_2 p \rceil$
Output:	$C \in GF(p)$
Step 1:	$C := 0$

Step 2: for $i = 0$ to $m - 1$
Step 3: $C := C + a_i B$
Step 4: $C := C + c_0 p$
Step 5: $C := C/2$
Step 6: if $C \geq p$ then $C := C - p$
Step 7: return C

In the case of $GF(2^m)$, the definitions and the algorithms are slightly different since we use polynomials of degree at most $m - 1$ with coefficients from the binary field $GF(2)$ to represent the field elements. Given two polynomials

$$A(x) = a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + \cdots + a_1 x + a_0$$
$$B(x) = b_{m-1}x^{m-1} + b_{m-2}x^{m-2} + \cdots + b_1 x + b_0 \ ,$$

and the irreducible monic degree-m polynomial

$$p(x) = x^m + p_{m-1}x^{m-1} + p_{m-2}x^{m-2} + \cdots + p_1 x + p_0$$

generating the field $GF(2^m)$, the Montgomery multiplication of $A(x)$ and $B(x)$ is defined as the field element $C(x)$ which is given as

$$C(x) = A(x) \cdot B(x) \cdot R(x)^{-m} \pmod{p(x)} \ . \tag{3}$$

We note that, as compared to Equation 1, $R(x) = x^m$ replaces $R = 2^m$. The representation of x^m in the computer is exactly the same as the representation of 2^m, i.e., a single 1 followed by 2^m zeros. Furthermore, the elements of $GF(p)$ and $GF(2^m)$ are represented using the same data structures. Only the arithmetic operations acting on the field elements differ. The Montgomery image of a polynomial $A(x)$ is given as $\bar{A}(x) = A(x) \cdot x^m \pmod{p(x)}$. Similarly, before performing Montgomery multiplication, the operands must be transformed into the Montgomery domain and the result must be transformed back. These transformations are accomplished using the precomputed variable $R^2(x) = x^{2m} \pmod{p(x)}$ as follows:

$$\bar{A}(x) = \mathsf{MonMul}(A, R^2) = A(x) \cdot R^2(x) \cdot R^{-1}(x) = A(x) \cdot R(x) \pmod{p(x)} \ ,$$
$$\bar{B}(x) = \mathsf{MonMul}(B, R^2) = B(x) \cdot R^2(x) \cdot R^{-1}(x) = B(x) \cdot R(x) \pmod{p(x)} \ ,$$
$$C(x) = \mathsf{MonMul}(\bar{C}, 1) = C(x) \cdot R(x) \cdot R^{-1}(x) = C(x) \pmod{p(x)} \ .$$

The bit-level Montgomery multiplication algorithm for the field $GF(2^m)$ is given below:

Input: $A(x), B(x) \in GF(2^m)$, $p(x)$, and m
Output: $C(x)$
Step 1: $C(x) := 0$
Step 2: for $i = 0$ to $m - 1$
Step 3: $C(x) := C(x) + a_i B(x)$
Step 4: $C(x) := C(x) + c_0 p(x)$
Step 5: $C(x) := C(x)/x$
Step 6: return $C(x)$

We note that the extra subtraction operation in Step 6 of the previous algorithm is not required in the case of $GF(2^m)$, as proven in [9]. Also, the addition operations are different. While addition in binary field is just bitwise mod 2 addition, the addition in $GF(p)$ requires carry propagation.

Our basic observation is that it is possible to design a unified Montgomery multiplier which can perform multiplication in both types of fields if an adder module, equipped with the property of performing addition with or without carry, is available. The design of an adder with this property is provided in the following sections.

The algorithms presented in this section require that the operations be performed using full precision arithmetic modules, thus, limiting the designs to a fixed degree. In order to design a scalable architecture, we need modules with the scalability property. The scalable algorithms are word-level algorithms, which we give in the following sections.

4.1 The Multiple-Word Montgomery Multiplication Algorithm for $GF(p)$

The use of fixed precision words alleviates the broadcast problem in the circuit implementation. Furthermore, a word-oriented algorithm allows design of a scalable unit. For a modulus of m-bit precision, $e = \lceil m+1/w \rceil$ words (each of which is w bits) are required. Note that one extra bit is used for all the variables in the actual implementation in order to take care of partial sum in the Montgomery algorithm, which can reach $(m+1)$-bit precision. The algorithm proposed in [20] scans the operand B (multiplicand) word-by-word, and the operand A (multiplier) bit-by-bit. The vectors involved in multiplication operations are expressed as

$$\begin{aligned} B &= (B^{(e-1)}, \ldots, B^{(1)}, B^{(0)}) \ , \\ A &= (a_{m-1}, \ldots, a_1, a_0) \ , \\ p &= (p^{(e-1)}, \ldots, p^{(1)}, p^{(0)}) \ , \end{aligned}$$

where the words are marked with superscripts and the bits are marked with subscripts. For example, the ith bit of the kth word of B is represented as $B_i^{(k)}$. A particular range of bits in a vector B from position i to j where $j > i$ is represented as $B_{j..i}$. $(x|y)$ represents the concatenation of two bit sequence. Finally, 0^m stands for an all-zero vector of m bits. The algorithm is given below:

Input: $A, B \in GF(p)$ and p
Output: $C \in GF(p)$
Step 1: $T := 0^{m+1}$
Step 2: for $i = 0$ to $m - 1$
Step 3: $(Carry|T^{(0)}) := a_i \cdot B^{(0)} + T^{(0)}$
Step 4: $Parity := T_0^{(0)}$
Step 5: $(Carry|T^{(0)}) := Parity \cdot p^{(0)} + (Carry|T^{(0)})$
Step 6: for $j = 1$ to $e - 1$

Step 7: $(Carry|T^{(j)}) := a_i \cdot B^{(j)} + Carry + T^{(j)} + Parity * p^{(j)}$
Step 8: $T^{(j-1)} := (T_0^{(j)}|T_{w-1..1}^{(j-1)})$
Step 9: $T^{(e-1)} := (Carry|T_{w-1..1}^{(e-1)})$
Step 10: $C := T$
Step 11: if $C > p$ then $C := C - p$
Step 12: return C

Note that the variable $Carry$ must be capable of accumulating more than one single bit. As suggested in [20], we use the Carry-Save form for the partial sum T, thus $T = (TC, TS)$ where TC and TS are carry and sum part of T, respectively.

4.2 Multiple-Word Montgomery Multiplication Algorithm for $GF(2^m)$

The Montgomery multiplication algorithm for $GF(2^m)$ is given below. Since there is no carry computation in $GF(2^m)$ arithmetic, the intermediate addition operations are replaced by bitwise XOR operations, which are represented below using the symbol $\oplus$.

Input: $A, B \in GF(2^m)$ and $p(x)$
Output: $C \in GF(2^m)$
Step 1: $T := 0^{m+1}$
Step 2: for $i = 0$ to m
Step 3: $T^{(0)} := a_i B^{(0)} \oplus T^{(0)}$
Step 4: $Parity := T_0^{(0)}$
Step 5: $T^{(0)} := Parity \cdot p^{(0)} \oplus T^{(0)}$
Step 6: for $j = 1$ to $e - 1$
Step 7: $T^{(j)} := a_i B^{(j)} \oplus TS^{(j)} \oplus Parity \cdot p^{(j)}$
Step 8: $T^{(j-1)} := (T_0^{(j)}|T_{w-1..1}^{(j-1)})$
Step 9: $T^{(e-1)} := (0|T_{w-1..1}^{(e-1)})$
Step 10: $C := T$
Step 11: return C

Notice that in the outer loop the index i runs from 0 to m. Since $(m + 1)$ bits are required to represent the irreducible polynomial of $GF(2^m)$, we prefer to allocate $(m + 1)$ bits to express the field elements.

5 Concurrency in Montgomery Multiplication

In this section, we analyze the concurrency in Montgomery multiplication algorithms as given in the subsections §4.1 and §4.2. In order to accomplish this task, we need to determine the inherent data dependencies in the algorithm and describe a scheme to allow the Montgomery multiplication to be computed on an array of processing units organized in a pipeline.

We prefer to accomplish concurrent computation of the Montgomery multiplication by exploiting the parallelism among the instructions across the different iterations of i-loop of the algorithms, as proposed in [20]. We scan the multiplier one bit at a time, and after the first words of the intermediate variables (TC, TS) are fully determined, which takes two clock cycles, the computation for the second bit of A can start. In other words, after the inner loop finishes the execution for $j = 0$ and $j = 1$ in ith iteration of the outer loop, the $(i+1)th$ iteration of outer loop starts its execution immediately. The dependency graph shown in Figure 1 illustrates these computations.

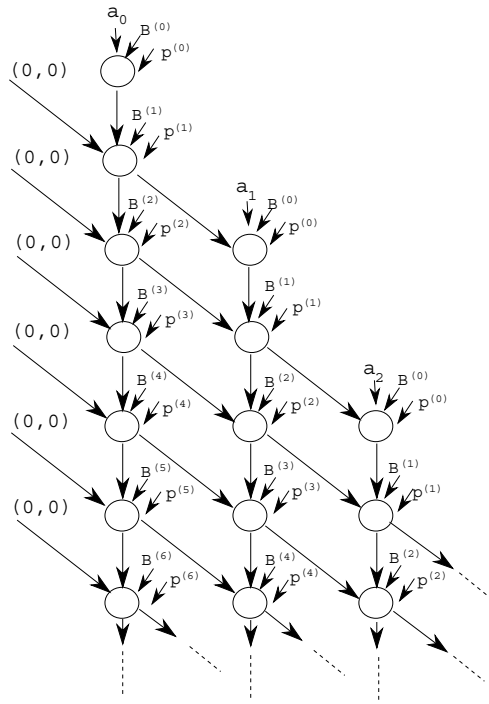


Figure 1: The Dependency Graph of the MonMul Algorithm.

Each circle in the graph represents an elementary computation performed in each iteration of the j-loop. We observe from this graph that these computations are very suitable for pipelining. Each column in the graph represents operations that can be performed by separate processing units (PU) organized as a pipeline. Each PU takes only one bit from multiplier A and operates on each word of multiplicand, B, each cycle. Starting from the second clock cycle, a PU generates one word of partial sum $T = (TC, TS)$ in the Carry-Save form at each cycle, and communicates it to the next PU which adds its contribution to the partial sum, when its turn comes. After $e + 1$ clock cycles, the PU finishes its portion of work, and becomes available for further computation. In case there is no

available PU and there is work to do, the pipeline must stall and wait for the working PUs to finish their jobs. Since the PU at the end of the pipeline has no way of communicating its result to another PU, we need to provide extra buffers for it. In the worst case, which happens when there is only one PU, there must be $2e$ extra buffers of w length to hold these partial sum words. In the last clock cycle of each column, the The PU responsible for this column must receive $p^{(e)} = B^{(e)} = 0$. Elementary computations represented by circles in Figure 1 are performed on the same hardware module. Local control module in the PU must be able to extract $T_0^{(0)}$ and keep this value for the entire operand scanning. Each PU, in other words, has to obtain this value and use it to decide whether to add the modulus p to the partial sum. This value is determined in the first clock cycle of each stage.

An example of the computation for 6-bit operands is shown in Figure 2 for the word size $w = 1$ provided that there are sufficient number of PUs preventing the pipeline to stall. Note that there is a delay of 2 clock cycles between the stage for x_i and the stage for x_{i+1}. The total execution time for the computation takes 20 clock cycles in this example.

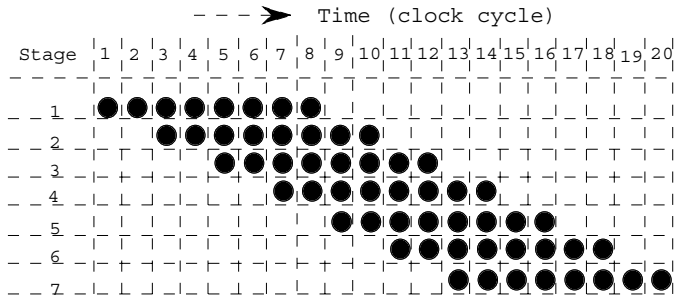


Figure 2: An Example of Pipeline Computation for 6-Bit Operands, where $w = 1$.

If there are at least $\lceil (e+1)/2 \rceil$ PUs in the pipeline organization the pipeline stalls do not take place. The total computation time, CC (clock cycles), is slightly different from the one in [20] and is given as

$$CC = \begin{cases} (\lceil \frac{m+1}{k} \rceil - 1)2k + e + 1 + 2(k-1) & \text{if } (e+1) < 2k \ , \\ (\lceil \frac{m+1}{k} \rceil)(e+1) + 2(k-1) & \text{otherwise} \ , \end{cases}$$

where k is the number of PUs in the pipeline. Notice that the first line of the formula gives the execution time in clock cycles when there are sufficiently many PUs while the second line corresponds to the case when there are stalls in the pipeline.

6 Scalable Architecture

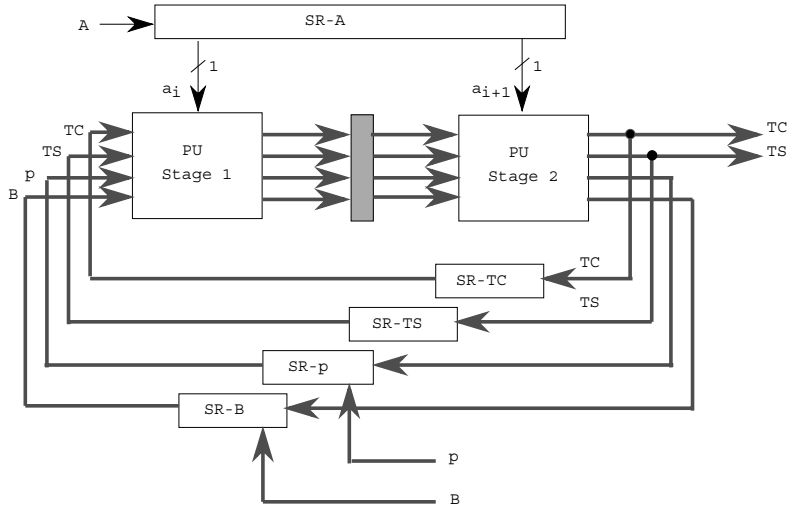


Figure 3: Pipeline Organization with 2 PUs.

An example of pipeline organization with 2 PUs is shown in Figure 3. An important aspect of this organization is the register file design. The bits of multiplier a_i are given serially to the PUs, and are not used again in later stages and can be discarded immediately. Therefore, a simple shift register would be sufficient for the multiplier. The registers for the modulus p and multiplicand B can also be shift registers. When there is no pipeline stall, the latches between PUs forward the modulus and multiplicand to next PU in the pipeline. However, if pipeline stalls occur, the modulus and multiplicand words generated at the end of the pipeline enter the $SR - p$ and $SR - B$ registers. The length of these shift registers are of crucial importance and determined by the number of pipeline stages (k) and the number of words (e) in the modulus. By considering that $SR - p$ and $SR - B$ values require one extra register to store the all-zero word needed for the last clock cycle in every stage (recall that $p^{(e)} = B^{(e)} = 0$) the length of these registers can be given as

$$L_1 = \begin{cases} e - 2 \cdot (k-1) \text{ if } (e+1) > 2k \ , \\ 0 \qquad\qquad\quad \text{otherwise.} \end{cases} \tag{4}$$

The width of the shift registers is equal to w, the wordsize. Once the partial sum (TC, TS) is generated, it is transmitted to the next stage without any delay. However, we need two shift registers, $SR-TC$ and $SR-TS$, to hold the partial sums from the last stage until the job in the first stage is completed. The length (L_2) of the registers TC and TS is equal to L_1.

The registers for TC, TS, B, and p must have loading capability which can complicate the local control circuit by introducing several multiplexers (MUX).

The delay imposed by these MUXes will not create a critical path in the final circuit. The global control block was not mentioned since its function can be inferred from the dependency graph and the algorithms.

6.1 Processing Unit

The processing unit (PU) consists of two layers of adder blocks, which we call *dual-field adders*. A dual-field adder is basically a full adder which is capable of performing addition both with carry and without carry. Addition with carry corresponds to the addition operation in the field $GF(p)$ while addition without carry corresponds to the addition operation in the field $GF(2^m)$. We give the details about the dual-field adder in the next subsection. The block diagram of a processing unit (PU) for $w = 3$ is shown in Figure 4.

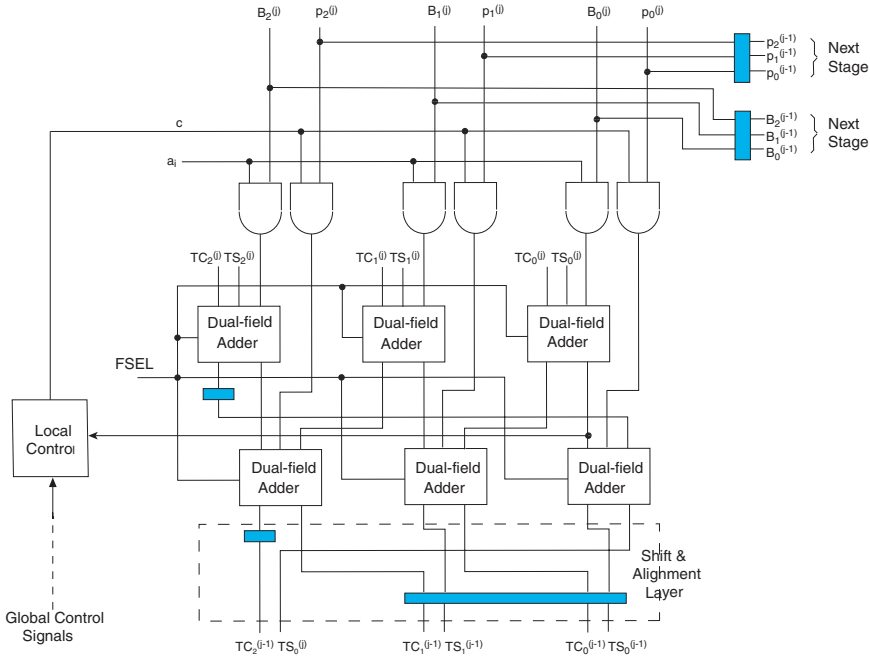


Figure 4: Processing Unit (PU) with $w = 3$.

The unit receives the inputs from the previous stage and/or from the registers $SR - A$, $SR - B$ and $SR - p$, and computes the partial sum words. It delays p and B for the first cycle, then, it transmits them to the next stage along with the first partial sum word (which is ready at the second clock cycle) if there is an available PU. The data path for partial sum $T = (TC, TS)$ (which is expressed in the redundant Carry-Save form) is $2w$ bits long while it is w bits long for p and B and 1 bit long for a_i. At the first cycle, the decision to add the modulus to the partial sum is determined, and this information is kept during the following e clock cycles by the local control. FSEL selects between $GF(p)$ and $GF(2^m)$ fields.

6.2 Dual-Field Adder

The dual-field adder (DFA) shown in Figure 5a, as mentioned before, is basically a full-adder equipped with the capability of doing bit addition both with and without carry. It has an input called $FSEL$ (field select) that enables this functionality. When $FSEL = 1$, the DFA performs the bit-wise addition with carry, which enables the multiplier to do arithmetic in the field $GF(p)$. When $FSEL = 0$, on the other hand, the output $Cout$ is forced to 0 regardless of the values of the inputs. The output S produces the result of bitwise modulo-2 addition of three input values. At most 2 of 3 input values of dual-field adder can have nonzero values while in the $GF(2^m)$ mode.

An important aspect of designing the dual-field adder is not to increase the critical path of the circuit compared to the full-adder, which can have an effect on the clock speed which this would be against our design goal. However, a small amount of extra area can be sacrificed. We show in the following section that this extra area is very insignificant. Figure 5b shows the actual circuit synthesized by Mentor Graphics tools using the $1.2\mu m$ CMOS technology.

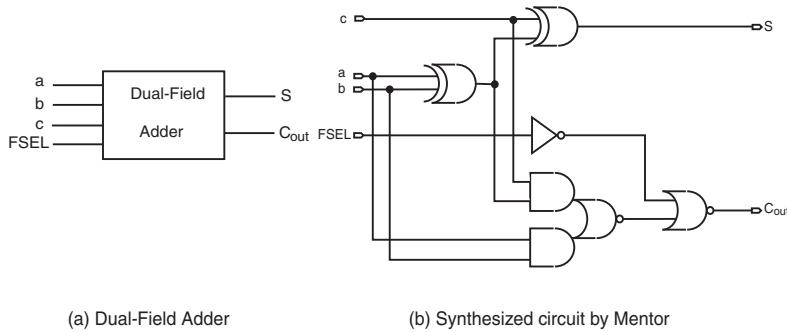


Figure 5: The Dual-Field Adder Circuit.

In the circuit, the two XOR gates are dominant in terms of both area and propagation time. As in the standard full-adder circuit, the dual-field adder has two XOR gates connected serially. Thus, propagation time of the dual-field adder is not larger than that of full adder. Their areas differ slightly.

7 Design Considerations

In [20], an analysis of the area and time tradeoffs is given for the scalable multiplier. The architecture allows designs with different word lengths and different pipeline organizations for varying values of operand precision. In addition, the area can be treated as a design constraint. Thus, one can adjust the design according to the given area, and choose appropriate values for the word length and the number of pipeline stages, in accordance. We give a similar analysis for the scalable and unified architecture. We are targeting two different classes of ranges for operand precision:

- *High precision range* which includes 512, 768 and 1024, is intended for applications requiring the exponentiation operation.
- *Moderate precision range* which includes 160, 192, 224, and 256, is typical for elliptic curve cryptography.

The propagation delay of the PU is independent of the wordsize w when w is relatively small, and thus all comparisons among different designs can be made under the assumption that the clock cycle is the same for all cases. The area consumed by the registers for the partial sum, the operands, and modulus is also the same for all designs, and we are not treating them as parts of the multiplier module.

The proposed scheme yields the worst performance for the case $w = m$ in the high precision range, since some extra cycles are introduced by the PU in order to allow word-serial computation, when compared to other full-precision conventional designs. On the other hand, using many pipeline stages with small wordsize values brings about no advantage after a certain point. Therefore, the performance evaluation reduces into finding an optimum organization for the circuit.

In order to determine the optimum selection for our organization, we obtain implementation results by synthesizing the circuit with Mentor Graphics tools using $1.2\mu m$ CMOS technology. The cell area for a given word size w is obtained as

$$A_{cell}(w) = 48.5w \tag{5}$$

units, and is slightly different from the one found in [20], where the multiplication factor in the formula is the area cost provided by the synthesis tool for a single bit slice. Note that a 2-input NAND gate takes up 0.94 units of area. In the pipelined organization, the area of the inter-stage latches is important, which was measured as

$$A_{latch}(w) = 8.32w \tag{6}$$

units. Thus, the area of a pipeline with k processing elements is given as

$$A_{pipe}(k, w) = (k-1)A_{latch}(w) + kA_{cell}(w) = 56,82kw - 8.32w \tag{7}$$

units. For a given area, we are able to evaluate different organizations and select the most suitable one for our application. The graphs given in Figure 6 allow to make such evaluations for a fixed area of 15,000 gates.

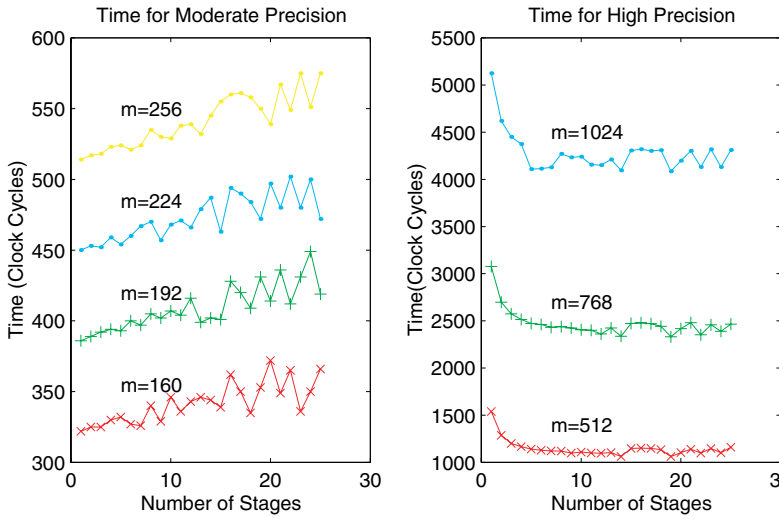


Figure 6: Time Efficiency for Different Configurations with a Fixed Area of 15,000 Gates.

For both moderate and high precision ranges, the number of stages between 5 and 10 are likely to give the best performance. For the high precision cases, fewer than 5 stages yields very poor performance since the fixed area becomes insufficient for large wordsizes and the performance degradation due to pipeline stalls becomes a major problem. The small number of stages with very long word sizes seem to provide a reasonable performance in the moderate range, however, because of the incompatibility issues about using very long word sizes and inefficiency when the precision increases, using fewer than 5 stages is not advised. We avoid using many stages for two reasons:

- high utilization of the PUs will be possible only for very high precision, and
- the execution time may have undesirable oscillations.

The behavior mentioned in the latter category is the result of the facts that

- extra stages at the end of the computations, and
- there is not a good match between the number of words e and the number of stages k, causing a underutilization of stages in the pipeline.

From the synthesis tool we obtained a minimum clock cycle time of 11 nanoseconds, which allows to use a clock frequency of up to 90MHz with $1.2\mu m$ CMOS. Using the CMOS technology with smaller feature size, we can attain much faster clock speeds. It is very important to know how fast this hardware organization really is when comparing it to a software implementation. The answer to this would determine whether it is worth to design a hardware module. In general, it is difficult to compare hardware and software implementations. In order to obtain realistic comparisons, a processor which uses similar clock cycles and technology must be chosen. We selected an ARM microprocessor [5] with 80

MHz clock which has a very simple pipeline. We compare the $GF(p)$ multiplication timing on this processor against that of our hardware module. We use the same clock frequency 80 MHz for the module of the pipeline organization with $w = 32$ and $k = 7$ for the hardware module. On the other hand, the Montgomery multiplication algorithm is written in the ARM assembly language by using all known optimization techniques [8,10]. Table 1 shows the multiplication timings and the speedup.

Table 1: The Execution Times of Hardware and Software Implementations of the $GF(p)$ Multiplication.

precision	Hardware (μs) (80 MHz, $w = 32$, $k = 7$)	Software (μs) (on ARM with Assembly)	speedup
160	4.1	18.3	4.46
192	5.0	25.1	5.02
224	5.9	33.2	5.63
256	6.6	42.3	6.41
1024	61	570	9.34

8 Conclusion

Using the design methodology proposed in [20], we obtained a scalable field multiplier for $GF(p)$ and $GF(2^m)$ in unified hardware module. The methodology can also be used to design separate modules for $GF(p)$ and $GF(2^m)$ which are fast, scalable and area-efficient. The fundamental contribution of this research is to show that it is possible to design a dual-field arithmetic unit without compromising scalability, the time performance and area efficiency. Our analysis shows that a pipeline consisting of several stages is adequate and more efficient than a single unit processing very long words. Working with relatively short words diminishes data paths in the final circuit, reducing the required bandwidth.

The proposed multiplier was synthesized using the Mentor tools, and a circuit capable of working with clock frequencies up to 90 MHz is obtained. Except for the upper limit on the precision which is dictated only by the availability of memory to store the operands and internal results, the module is capable of performing infinite-precision Montgomery multiplication in $GF(2^m)$ and $GF(p)$.

References

1. G. B. Agnew, R. C. Mullin, and S. A. Vanstone. An implementation of elliptic curve cryptosystems over F_{2155}. *IEEE Journal on Selected Areas in Communications*, 11(5):804–813, June 1993.
2. A. Bernal and A. Guyot. Design of a modular multiplier based on Montgomery's algorithm. In *13th Conference on Design of Circuits and Integrated Systems*, pages 680–685, Madrid, Spain, November 17–20 1998.
3. W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22:644–654, November 1976.

4. S. E. Eldridge and C. D. Walter. Hardware implementation of Montgomery's modular multiplication algorithm. *IEEE Transactions on Computers*, 42(6):693–699, June 1993.
5. Steve Furber. *ARM System Architecture*. Addison-Wesley, Reading, MA, 1997.
6. B. S. Kaliski Jr. The Montgomery inverse and its applications. *IEEE Transactions on Computers*, 44(8):1064–1065, August 1995.
7. N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177):203–209, January 1987.
8. Ç. K. Koç. High-Speed RSA Implementation. Technical Report TR 201, RSA Laboratories, 73 pages, November 1994.
9. Ç. K. Koç and T. Acar. Montgomery multiplication in GF(2^k). *Designs, Codes and Cryptography*, 14(1):57–69, April 1998.
10. Ç. K. Koç, T. Acar, and B. S. Kaliski Jr. Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, June 1996.
11. P. Kornerup. High-radix modular multiplication for cryptosystems. In E. Swartzlander, Jr., M. J. Irwin, and G. Jullien, editors, *Proceedings, 11th Symposium on Computer Arithmetic*, pages 277–283, Windsor, Ontario, June 29 – July 2 1993. IEEE Computer Society Press, Los Alamitos, CA.
12. A. J. Menezes. *Elliptic Curve Public Key Cryptosystems*. Kluwer Academic Publishers, Boston, MA, 1993.
13. P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, April 1985.
14. D. Naccache and D. M'Raïhi. Cryptographic smart cards. *IEEE Micro*, 16(3):14–24, June 1996.
15. National Institute for Standards and Technology. Digital Signature Standard (DSS). FIPS PUB 186-2, January 2000.
16. H. Orup. Simplifying quotient determination in high-radix modular multiplication. In S. Knowles and W. H. McAllister, editors, *Proceedings, 12th Symposium on Computer Arithmetic*, pages 193–199, Bath, England, July 19–21 1995. IEEE Computer Society Press, Los Alamitos, CA.
17. J.-J. Quisquater and C. Couvreur. Fast decipherment algorithm for RSA publickey cryptosystem. *Electronics Letters*, 18(21):905–907, October 1982.
18. A. Royo, J. Moran, and J. C. Lopez. Design and implementation of a coprocessor for cryptography applications. In *European Design and Test Conference*, pages 213–217, Paris, France, March 17-20 1997.
19. E. Savaş and Ç. K. Koç. The Montgomery modular inverse - revisited. *IEEE Transactions on Computers*, 49(8), July 2000. To appear.
20. A. F. Tenca and Ç. K. Koç. A scalable architecture for Montgomery multiplication. In Ç. K. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems*, Lecture Notes in Computer Science, No. 1717, pages 94–108. Springer, Berlin, Germany, 1999.
21. C. D. Walter. Space/Time trade-offs for higher radix modular multiplication using repeated addition. *IEEE Transactions on Computers*, 46(2):139–141, February 1997.

Montgomery Exponentiation with no Final Subtractions: Improved Results

Gaël Hachez and Jean-Jacques Quisquater

Université Catholique de Louvain, UCL Crypto Group
Place du Levant, 3, B-1348 Louvain-la-Neuve, Belgium
{hachez,quisquater}@dice.ucl.ac.be

Abstract. The Montgomery multiplication is commonly used as the core algorithm for cryptosystems based on modular arithmetic. With the advent of new classes of attacks (timing attacks, power attacks), the implementation of the algorithm should be carefully studied to thwart those attacks. Recently, Colin D. Walter proposed a constant time implementation of this algorithm [17,18]. In this paper, we propose an improved (*faster*) version of this implementation. We also provide figures about the overhead of these versions relatively to a speed optimised version (theoretically and experimentally).
Keywords: Montgomery multiplication, modular exponentiation, smart cards, timing attacks, power attacks.

1 Introduction

In RSA based crypto-systems, modular exponentiations are often computed with Montgomery multiplications [14].The optimisation of this algorithm is consequently very important. Several fast implementations of this algorithm were proposed both in hardware (e.g. [18]) and software (e.g. [10,6]). These implementations were mainly designed to achieve speed gains.

Recently, a new range of attacks (timing attacks [11] and power attacks [12]) appeared. These attacks are based on side-channel information that are leaked by the hardware device. The tricks used to optimise to the utmost the speed of the algorithm usually amplify this side-channel information. Therefore, new implementations of the algorithm are being created to reduce these threats while almost preserving the speed performance.

In two recent papers [17,18], Colin D. Walter shows that, with a correct implementation, it is possible to make a complete exponentiation based on Montgomery multiplications without any modular reduction (even at the end of the exponentiation) [1]. His implementation is slower than an optimised one although a security gain is achieved against timing attacks and power attacks.

[1] Similar results were already obtained for slower modular multiplication algorithms such as Barrett and Quisquater multiplications (see [6]).

Ç.K. Koç and C. Paar (Eds.): CHES 2000, LNCS 1965, pp. 293–301, 2000.

The author focuses on hardware implementations while neglecting software implementations that are commonly used even in embedded hardware such as smart cards[2].

Here, we will show a tighter bound on the assumptions made by Colin D. Walter that allow us to speed up software implementations. To illustrate this gain, we will show some figures about performance on a 32-bit RISC-based chip for smart card.

In hardware, the situation is more complex. Usually the tighter bound will either speed up a hardware implementation, or reduce the size of the circuitry needed to obtain this implementation of the Montgomery multiplication. In a particular case, if the size of the modulus is smaller than the size of the multiplier, the new implementation is not suitable.

2 Montgomery Multiplication

The Montgomery multiplication is an algorithm used to compute the product of two integers A and B modulo an integer N.

Because A and B are, for security reasons, quite large, the multiplication is computed with A and B decomposed in small blocks. Those blocks usually have a length t of 8, 16, 32, 64 bits and each number can be decomposed in the form $X = \sum_{i=0}^{p-1} x_i 2^{it}$ where p is the number of blocks needed to represent all numbers used in the algorithm.

The Montgomery multiplication algorithm is described in Fig. 1. As Barrett [2,3] and Quisquater [15,16] modular multiplication, this one does not require any division (expensive operation in hardware). Here, the multiplication is done from left (high order bits) to right (low order bits) which is not the classical order used to make a multiplication.

{Pre-condition: N prime to 2^t}
$S = 0$
for $i = 0$ **to** $p - 1$
    $q_i = (s_0 + a_i b_0) n'_0 \bmod 2^t$
    $S = (S + a_i \times B + q_i \times N) \text{ div } 2^t$
    {Invariant: $0 \leq S < N + B$}
endfor
{Post-condition: $S2^{pt} = A \times B + Q \times N$}

Fig. 1. Montgomery Multiplication

The value n'_0 is computed so that $n_0 \times n'_0 \equiv 1 \bmod N$. The integer p must be chosen such that $N < 2^{pt}$. For more details on the algorithm, see [14,6,18].

[2] The latest chip developed by ST Microelectronics, the smartJ 22 contains software implementation of public key primitives.

3 Montgomery-Based Exponentiation

3.1 Description

The Montgomery multiplication is the basic component used to implement a classical square and multiply algorithm that computes an exponentiation. The result of a Montgomery multiplication ($\overline{\times}$) is not $A \times B \bmod N$ but rather $A \times B \times 2^{-pt} \bmod N$. To obtain a correct result at the end of the exponentiation, we need to make a pre-multiplication ($A \ \overline{\times}\ 2^{2pt} \bmod N$) and a post-multiplication ($A^e \ \overline{\times}\ 1 \bmod N$).

With the following assumptions: $A < 2N, t \geq 1$ and $2N < 2^{(p-1)t}$ C. Walter [17,18] proves that the end-result of the exponentiation (E) is lower than the modulus (N) and does not need any further modular reduction. We will rapidly sketch out the proof.

Proof. Because the result of the multiplication is used as input for the next multiplication, the output must have the same bound as the input. At the second last iteration, we have $S' < N + B$. The assumptions $A < 2N$ and $2N < 2^{(p-1)t}$ guarantee that $a_{p-1} = 0$. Therefore at the last iteration, we have $S < N + 2^{-t}B < 2N$.

At the last multiplication of the exponentiation, we have $A^e < 2N$. The post-multiplication by 1 will remove the possible last reduction. We have at the end: $E2^{pt} = A^e + QN$. $Q < 2^{nt}$ and $A^e < 2N$ implies that: $E2^{pt} < (2^{pt} + 1)N$. We obtain $S \leq N$ (S is an integer). The last case $S = N$ is removed because it implies that $A^e \equiv 0 \bmod N$ and therefore $A \equiv 0 \bmod N$. This signifies that either $A = 0$ (no reductions) or $A = N$ (in a classical crypto-system, $A < N$). □

3.2 Shortcomings

The first part of the proof shows the non-growing property of the Montgomery multiplication. With $A, B < 2N, t \geq 1$ and $2N < 2^{(p-1)t}$ the output of the multiplication is bound: $S < 2N$.

While this result is true, we should not forget the pre-multiplication phase. In this pre-multiplication the integer A is multiplied by 2^{2pt} that is obviously greater than $2N$ and thus we have no insurance that S will be bounded by $2N$ after this pre-multiplication. Therefore, we can not be sure that the result at the end of the exponentiation will not require a final reduction.

We have two solutions to avoid that (proposed in [7,8]):

- pre-compute $2^{2pt} \bmod N$
- use a normal modular multiplication algorithm (Barrett or Quisquater) and compute $A \times 2^{pt} \bmod N$.

Besides this little problem, performance is impeded by one assumption. The $2N < 2^{(p-1)t}$ condition can be very annoying. Specially if we take classical sizes for N and t.

Example 1. We have a modulus N (512 bits) and a 32x32 multiplier ($t = 32$), then we need $p = 18$ instead of $p = 16$ which lowers the performance because the number of multiplications is $O(p)$. With non classical sizes of modulus such as 510 bits, we obtain $p = 17$ instead of $p = 16$ which is less annoying.

For the rest of the paper, we will suppose that we are in a typical case where the size of N is equal to 512, 768, 1024, 2048 bits and $t = 32$.

3.3 Bound Optimisation

We can improve this bound and prove that the result ($S < 2N$) still holds even with $N < 2^{(p-1)t}$ and with a tighter constraint on t: that is, $t \geq 2$ which is obviously not a problem in a software implementation.

In hardware, this can be a problem. If the size of N is less than 2^t, this result does not stand. However this situation does not happen very often as, nowadays, the minimum size for N is at least 512 bits.

At each step of the algorithm the following bound is satisfied: $S < N + B$. From $N < 2^{(p-1)t}$ and $A < 2N$, we know that $a_{p-1} \in \{0, 1\}$. If we start from the second last iteration we have that:

$$
\begin{aligned}
S' &= (S + a_{p-1} \times B + q_{p-1} \times N) \text{ div } 2^t \\
S' &\leq (S + B + q_{p-1} \times N) \text{ div } 2^t \\
S' &\leq (S + B + (2^t - 1) \times N) \text{ div } 2^t \\
S' &< (N + B + B + (2^t - 1) \times N) \text{ div } 2^t \\
S' &< (2B + 2^t \times N) \text{ div } 2^t \\
S' &< 2B \text{ div } 2^t + N \\
S' &< 4N \text{ div } 2^t + N \\
S' &< 2N \quad \square
\end{aligned}
$$

The remaining of the proof is the same as Walter's one because he does not require anymore that $2N < 2^{(p-1)t}$. Therefore, we proved that we still avoid a final reduction at the end of the exponentiation with better bounds.

Example 2. In the previous example, this new bound is $p = 17$ which is worse than the classical algorithm but better than Walter's version.

4 Speed Analysis

4.1 Building a Generic Model

We can build an approximative model of the number of operations required for a Montgomery multiplication. Let C_A represent the number of clock cycles for an addition and C_M the number of clock cycles for a multiplication. At each step, we need:

- $(2C_A + 2C_M)p$ clock cycles for computing S
- $C_A + 2C_M$ clock cycles for computing q_i .

We need to make a final subtraction in the case of the original Montgomery multiplication: this final subtraction takes $C_A p$ clock cycles. So we have the following formulae to compute the approximative clock cycles required for a Montgomery multiplication:

- $((2C_A + 2C_M)p + C_A + 2C_M)p$,
- $((2C_A + 2C_M)p + 2C_A + 2C_M)p$ with a final subtraction.

4.2 Adaptation to the ARM7M

We already had a cryptographic library that was designed in the European project CASCADE [4] by J.-F. Dhem. The library runs on an ARM7M CPU (this CPU is used in the GemXpresso 2.0 smart card from Gemplus). Therefore, we used this platform to experimentally compare the performance of the implementations.

The ARM7M is a pure RISC processor. It does not hold any division instructions and there is no support for floating point operations. On the ARM7M, an addition takes 1 clock cycle ($C_A = 1$). The multiplication is a little more complex. The ARM7M possess a dedicated multiply unit that is able to multiply 32x8 bits. Therefore, to multiply 32x32 bits and obtain a 64 bits result, this unit must be used four times. If we add the setup time, a multiplication usually takes 6 clock cycles ($C_M = 6$).

The time taken by the multiplication is not always constant due to optimisations in the ARM7M. If one of the 8 bits blocks of the operand is null, this sub-part of the multiplication is skipped. More details are available in [1]. In particular, if the operand is null then the number of clock cycles decreases from 6 to 2 (the setup time only).

Remembering that the block $a_{p-1} \in \{0, 1\}$, if we take one block more, we need to adapt the above formulae to deal with this non-constant time. So if we take one block more (this paper), we consider that the last block's multiplication for computing S takes only 2 clock cycles[3] and if we take two blocks more (Walter's version), we consider that the last two blocks' multiplication takes only two clock cycles. We obtain thus the following estimations in Table 1.

4.3 Speed Comparison

The library we use has been protected against timing attacks. The original version of the Montgomery algorithm always makes a subtraction after the multiplication and chooses to take the result of the subtraction if it is greater than zero, otherwise the result remains unchanged. A modification was made to avoid timing attacks by adding cycles to have the same timing when the result of the subtraction must be discarded. See [5,9] for timing attacks on this library.

[3] This is a valid approximation because most of the time $a_{p-1} = 0$

Table 1. Formulae (based on a simple model of the ARM7) used to predict the number of clock cycles required for the different versions of the algorithm.

Value	This paper	Walter's version
q_i	$C_A + 2C_M$	$C_A + 2C_M$
S	$(2C_A + 2C_M)p + 2C_A + 2C_{M'}$	$(2C_A + 2C_M)p + 2(2C_A + 2C_{M'})$

Table 2. Predicted time increase for a multiplication ($C_A = 1$, $C_M = 6$) relatively to the standard version with an ending modular reduction ($(14p + 14)p$).

Size of N	This paper $(14p + 6 + 13)(p + 1)$	Walter's version $(14p + 12 + 13)(p + 2)$
512 bits ($p = 16$)	8.5 %	17.7 %
768 bits ($p = 24$)	5.6 %	11.7 %
1024 bits ($p = 32$)	4.2 %	8.8 %
2048 bits ($p = 64$)	2.1 %	4.4 %

However because those added cycles come from an empty loop, this is not a protection against power attacks [13,12].

If we compare predicted results in Table 2 and real results in Table 3, we can see some divergence. This is normal due to the following facts:

- The prediction is made on one multiplication and we get the results on a complete exponentiation without taking the added time into account.
- There is a 3-stage pipeline in the ARM7.
- This is a basic model (no memory operations are taken into account).

It is crucial to note the improvement will be far higher if we take a CPU architecture where the multiplication takes a constant time whatever the value of the operands. Suppose that the time of a multiplication is the same as the time of the addition and equals one clock cycle, we obtain the following results in Table 4.

5 Security Considerations

Today, in smart cards, absolute performance is not the only objective for algorithms anymore. New kinds of side channels based attacks (like the time [11], the power [12]) appeared and security algorithms must be protected against them. This is usually done at the expense of the performance of algorithms. We will see how this algorithm theoretically performs against timing and power attacks.

Table 3. Average time increase for an exponentiation relatively to the standard version with an ending modular reduction.

Size of N	This paper	Walter's version
512 bits	6.3 %	17.6 %
768 bits	4.3 %	11.9 %
1024 bits	3.3 %	9 %
2048 bits	1.6 %	4.5 %

Table 4. Predicted time increase for a multiplication ($C_A, C_M = 1$) relatively to the standard version with an ending modular reduction ($(4p + 4)p$).

Size of N	This paper $(4(p+1)+3)(p+1)$	Walter's version $(4(p+2)+3)(p+2)$
512 bits ($p = 16$)	10.9 %	24 %
768 bits ($p = 24$)	7.3 %	15.9 %
1024 bits ($p = 32$)	5.5 %	11.9 %
2048 bits ($p = 64$)	2.7 %	5.9 %

5.1 Timing Attacks

The original speed optimised algorithm is already protected against timing attacks. Against such attacks our version does not add more security. However this is a cleaner design than always perform a subtraction and add an empty loop (if needed) at the end of the exponentiation.

5.2 Power Attacks

In the original speed optimised version, after the always performed final subtraction, a conditional instruction must decide whether the result of the final subtraction must discarded. Because the result is returned by value and not by address, if the result must be kept, it must be copied. To avoid timing attacks, in the other case (no copy), an empty loop is executed to simulate the time taken by the copy. This method can be easily detected in a power attack. In our new version, a security gain is achieved because no conditional instructions exist anymore.

At first sight, it can only be considered as a security gain because it will not be sufficient to protect against power attacks. Indeed, attacks can be mounted on the exponentiation algorithm independently of the multiplication algorithm as, here, a conditional Montgomery multiplication is executed within the exponentiation algorithm depending on the value of each key bit. This is unrelated to the multiplication algorithm used, it depends on the exponentiation algorithm (attacks of this type were done in [13]).

6 Conclusion

We notice an important improvement of the performance with this version of the Montgomery multiplication but it remains slower than the speed optimised version. With a more generic platform than the ARM7, we should obtain even better improvements as shown in Table 4.

The security gain is related to power attacks [12] against smart cards as there are no more conditional reductions. However, this is not sufficient because the exponentiation algorithm itself is not protected against power attacks.

References

1. ARM. *ARM 7TDMI Data Sheet*, August 1995. Document number: ARM DDI 0029E.
2. P. Barrett. Communications, Authentication and Security Using Public Key Encryption - A Design for Implementation. Master's thesis, Oxford University, September 1984.
3. P. Barrett. Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor. In A. M. Odlyzko, editor, *Advances in Cryptology - CRYPTO '86*, volume 263 of *LNCS*, pages 311–323. Springer-Verlag, 1987.
4. CASCADE (Chip Architecture for Smart CArds and portable intelligent DEvices). `http://www.dice.ucl.ac.be/crypto/cascade/`, 1997.
5. J.-F. Dhem, F. Koeune, P.-A. Leroux, P. Mestré, J.-J. Quisquater, and J.-L. Willems. A Practical Implementation of the Timing Attack. In *CARDIS '98*, LNCS. Springer-Verlag, 1998. to appear.
6. Jean-François Dhem. *Design of an Efficient Public-key Cryptographic Library for RISC-based Smart Cards*. Ph.D. Thesis, Université Catholique de Louvain, May 1998.
7. Stephen E. Eldridge. A Faster Modular Multiplication Algorithm. *Inter. J. Comput. Math.*, 40:63–68, 1991.
8. Stephen E. Eldridge and Colin D. Walter. Hardware Implementation of Montgomery's Modular Multiplication Algorithm. *IEEE Transactions on Computers*, 42(6):693–699, June 1993.
9. Gael Hachez, François Koeune, and Jean-Jacques Quisquater. Timing Attack: What Can Be Achieved by a Powerful Adversary? In A. Barbé, E.C. van der Meulen, and P. Vanroose, editors, *The 20th symposium on Information Theory in the Benelux*, pages 63–70, May 1999.
10. Kouichi Itoh, Masahiko Takenaka, Naoya Torii, Syouji Temma, and Yasushi Kurihara. Fast Implementation of Public-Key Cryptography on a DSP TMS320C6201. In Çetin K. Koç and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES '99*, volume 1717 of *LNCS*, pages 61–72. Springer-Verlag, August 1999.
11. Paul Kocher. Timing Attack on Implementations of Diffie-Hellman, RSA, DSS and other systems. In Neil Kobliz, editor, *Advances in Cryptology - CRYPTO '96*, volume 1109 of *LNCS*, pages 104–113. Springer-Verlag, August 1996.
12. Paul Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In M. Wiener, editor, *Advances in Cryptology - CRYPTO '99*, volume 1666 of *LNCS*, pages 388–397. Springer-Verlag, August 1999.

13. Thomas S. Messerges, Ezzy A. Dabbish, and Robert H. Sloan. Power analysis Attack of Modular Exponentiation in Smartcards. In Çetin K. Koç and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES '99*, volume 1717 of *LNCS*, pages 144–157. Springer-Verlag, August 1999.
14. Peter L. Montgomery. Modular Multiplication Without Trial Division. *Mathematics of Computation*, 44(170):519–521, April 1985.
15. Jean-Jacques Quisquater. Procédé de Codage selon la Méthode dite RSA, par un Microcontrôleur et Dispositifs Utilisant ce Procédé. Demande de brevet français. (Dépôt numéro: 90 02274), February 1990.
16. Jean-Jacques Quisquater. Encoding System According to the So-called RSA Method, by Means of a Microcontroller and Arrangement Implementing this System. U.S. Patent 5,166,978, November 1992.
17. Colin D. Walter. Montgomery Exponentiation Needs no Final Subtractions. *Electronics Letters*, 35(21):1831–1832, October 1999.
18. Colin D. Walter. Montgomery's Multiplication Technique: How to Make It Smaller and Faster. In Çetin K. Koç and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES '99*, volume 1717 of *LNCS*, pages 80–93. Springer-Verlag, August 1999.

Physical Security Devices for Computer Subsystems: A Survey of Attacks and Defenses

Steve H. Weingart

Secure Systems and Smart Card Group
IBM Thomas J. Watson Research Center, Hawthorne, NY
weingart@us.ibm.com

Abstract. As the value of data on computing systems increases and operating systems become more secure, physical attacks on computing systems to steal or modify assets become more likely. This technology requires constant review and improvement, just as other competitive technologies need review to stay at the leading edge.
This paper describes known physical attacks, ranging from simple attacks that require little skill or resource, to complex attacks that require trained, technical people and considerable resources. Physical security methods to deter or prevent these attacks are presented. The intent is to match protection methods with the attack methods in terms of complexity and cost. In this way cost effective protection can be produced across a wide range of systems and needs.
Specific technical mechanisms now in use are shown, as well as mechanisms proposed for future use. Common design problems and solutions are discussed with consideration for manufacturing.

1 Introduction

Traditionally the term 'physical security' has been used to describe protection of material assets from fire, water damage, theft, or similar perils. However, recent concerns in computer security have caused physical security to take on a new meaning: Technologies used to safeguard information against physical attack.

In this new sense, physical security is a barrier placed around a computing system to deter unauthorized physical access to the computing system itself. This concept is complementary to logical security, the mechanisms by which operating systems and other software prevent unauthorized access to data. Both physical and logical security are complementary to environmental security. Environmental security is the protection the system receives by virtue of location such as guards, cameras, badge readers, access policies, etc. The reason for separating physical and environmental security is partly due to the change in the nature of the assets being protected. In the past the assets to be protected were nominally physical items: cash, jewelry, bonds, etc. Now the assets are often information, which can be stolen without being physically removed from where they are kept. If information can be seen, it can simply be copied. This information can be anything from a spreadsheet work file to cryptographic keys. It may

Ç.K. Koç and C. Paar (Eds.): CHES 2000, LNCS 1965, pp. 302–317, 2000.

be reasonable for an individual to have access to a location (environmental security) and not to have access to the information stored on a computing system in that environment (physical security).

Physical security is also becoming more important because computing systems are moving out of environmentally secure computer rooms and into less environmentally secure offices and homes. At the same time, the value of the data on these computing systems is increasing. Logical security has also been improved so that a physical attack may become more easily performed than a logical attack [1]. We can see that the motivation to attack computing systems is increasing because the rewards for doing so are increasing.

For physical security to be effective the following criteria must be met: in the event of an attack, there should be a low probability of success and a high probability of detection either during the attack, or subsequent to penetration [17].

It is possible to build physical security systems to protect sensitive data [12,5,6,15] These systems can make unauthorized access to the data difficult, as a bank vault makes stealing cash a daunting task (tamper resistant). They can trigger mechanisms to thwart the attack, much like an alarm system (tamper responding). They can make an attempted attack apparent so that subsequent inspection will show an attack had been attempted (tamper evident).

Classification systems have been proposed that evaluate computing systems according to criteria that measure the difficulty of mounting a successful attack [16,8]. Requiring additional documentation, testing, and quality assurance further ensures increasing degrees of security. Continued work has lead to the advancement of standards [9], these standards are becoming accepted since trying to do one's own evaluation is a daunting task and the standards are being rigorously and publicly evaluated.

Physical security technology is a relatively recent addition to computing system design. This paper attempts to describe and catalog the currently known design and implementation techniques. Effort is made to differentiate between simple methods, which are applicable in areas of low criticality vs. the sophisticated methods required for protecting very critical data.

2 Kinds of Physical Security

A number of physical security methods are currently in use. This is a new field in the commercial market and is still being developed. The US government has been working on this problem for over 25 years but the results remain classified. The ways and means described here are not an exhaustive list, nor are they represented as ultimate methods. Development is continuing in protection methods and it is proceeding in attack methods. Any evaluation of appropriateness of a physical security system is time dependent and must be repeated periodically. For example, the FIPS 140 standard [9] is to be re-evaluated at five-year intervals.

2.1 Tamper Resistant

Tamper resistant systems take the bank vault approach. This type of system is typified by the outer case design of an automated teller machine. Thick steel or other robust materials are utilized to slow down the attack by requiring tools and great effort to breach the system. This type of system can be used in many environments and sometimes has the advantage of being so physically heavy (as in automated teller machines), that it resists theft by sheer weight. However recent thefts of automated teller machines by thieves using towing chains and four-wheel drive vehicles may indicate that ATMs are no longer sufficiently tamper resistant. A system that is only tamper resistant has the disadvantage that the owner may not be aware of the loss until the break-in is discovered. That may be never, if the attacker did a 'neat' job and replaced any material that had been removed.

Tamper resistant physical security is usually the easiest to apply. Steel cases and locks are well-known technology and are easily manufactured. Weight and bulk can be a problem or benefit, depending on the application.

Complexity or size can be another variety of tamper resistance. Single chip implementations of secure devices have a certain level of physical security due to the small size of the features and complexity involved in the determination of which part of a circuit performs which function. However this advantage is rapidly being lost as the equipment and skills needed to work with semiconductor devices at the microscopic level are becoming commonly available at many universities and technology centers.

Tamper responding systems use the burglar alarm approach. The defense is the detection of the intrusion, followed by a response to protect the asset. In the case of attended systems, the response may consist of sounding an alarm. Erasure or destruction of secret data is sometimes employed to prevent theft in the case of isolated systems which cannot depend on outside response. Tamper responding systems do not depend on robust construction or weight to guard an asset. Therefore, they are good for portable systems or other systems where size and bulk are a disadvantage.

Tamper evident systems are designed to ensure that if a break-in occurs, evidence of the break-in is left behind. This is usually accomplished by chemical or chemical/mechanical means, such as a white paint that 'bleeds' red when cut or scratched, or tape or seals that show evidence of removal. This approach can be very sensitive to even the smallest of penetrations. Frangible (brittle, breakable) covers or seals are other methods available using current technology.

These systems are not designed to prevent an attack or to respond to the indication that one is in progress. Their job is to ensure that the fact of a break-in will remain known and can be ascertained at a later time. An audit policy must exist, and be adhered to, for a tamper evident system to be effective. Otherwise it may not be known if, or when, the system was breached. If no one looks for the evidence of tampering, that evidence will never be found.

Some Additional Physical Security Considerations: Some of the properties of specific methods of physical security were discussed with the introduction of each type. Here, some additional points are considered. One must examine each system to determine the correct protection.

Size and Weight: The size and weight implications of a potential physical security design must be considered in the light of the application. Thick steel would not be a good idea for a portable system. A lightweight system would not be effective for an automated teller machine, as it would allow the system to be carried away more easily.

Mixed and Layered Systems: In many cases a security system can be made substantially more secure by using more than one layer and more than one kind of system. For example, a typical safety deposit vault has steel walls, an alarm system, and a high quality vault lock. These methods might seem sufficient, but the individual safety deposit boxes have significant locks as well. The individual locks serve two purposes. They provide a second layer of general security by requiring an attacker to break into each box individually after breaking into the vault. The locks on the individual boxes also serve as an additional authorization/authentication process which requires an individual to possess the correct key to open the box.

Similarly, a layer of tamper evident security placed over a layer of tamper resistance or tamper response can prevent an attack, which might be attempted over a period of days. A regular audit may turn up indications of tampering before the system is fully breached and allow additional measures to be taken before the attack is completed.

Multiple layers of security also make the attack more difficult in general. The requirement for two different kinds of tools, skills, etc., may not make the two-layered system twice as difficult to attack, but it does increase the difficulty.

3 Physical Security Methods and Mechanisms

The following sections describe different methods of physical attack that may be attempted upon computing systems, as well as the defense mechanisms that can be useful in deterring or detecting such attacks.

Physical security can be broadly divided into two categories: high technology and low technology. Low technology concepts such as inserting desktop systems into external steel cases and using floppy drive cover locks are fairly well known and will not be discussed here. The high technology examples will explore existing and contemplated attack mechanisms, and the corresponding defense mechanisms that are being brought into commercial use now, or are being considered for the near future.

3.1 High Technology Attacks

This section deals with mechanisms that used to be considered unusual. The attacks described in this section, and the defenses described in the following section, far exceed the typical levels of skills and resources available to the common attacker. However, the skill level of the common attacker is increasing. These attacks and defenses are presented to meet the requirements of markets such as banking. However as data value increases, as is occurring now with the rise of Internet commerce, these defensive techniques should become a standard part of common business practice. These techniques have are now required to meet certain government requirements [9]. The business community is also beginning to embrace these standards as a means of assurance.

Probe Attacks: The purpose of a probe attack is to directly attach conductors to the circuit(s) being protected so that information can be obtained from, and/or changes injected into, the system under attack.

Passive Probes: These are common oscilloscope or logic analyzer probes. They may be used to watch and record information contained in circuits. When used with a logic analyzer, a trigger condition may be set such that the attacker waits for a predetermined event and then begins recording.

The term passive probe is somewhat of a misnomer in that so-called passive probes may be terminated in active circuitry, which gives them very high input impedance. This may prevent their detection by, or interference with, the circuit being attacked.

Active or Injector Probes: Active probes are generally used in conjunction with passive probes. Using a pattern generator or similar device, these probes can inject signals or information into an active system.

Pico-Probes: Pico-probes can be used in either of the capacities described above. Pico-probes are very tiny and are used to directly probe the surfaces of integrated circuits.

Energy Probes: Energy probes can be electron beams, ion beams, or focused beams of light. Depending on the technology being attacked, energy probes can read or write the contents of semiconductor storage, or change control signals. Ion beam deposition has been used to successfully reconnect fuse links, to return product level smart cards to their debug-state where the output of key registers, etc., was permitted.

Machining Methods: The purpose of machining is to cut or remove material. In this context, a cover or potting material is machined to access circuitry

beneath the potting or cover. Once the covering is removed, a probe attack as described above can proceed.

If the system is protected by physical security, the intent is to perform the machining operation without tripping a sensor or leaving evidence[1]. After the covering material is removed, the sensor is then disabled or bypassed so that a probing attack may proceed. If the system is protected by a tamper evident system, there may be an attempt to cover the evidence after the attack is complete.

The list of machining methods include chemical and energy methods of material removal, as well as traditional machining methods.

Manual Material Removal: Manual material removal is commonly referred to as the 'brain surgery' attack. In this scenario an attacker using a knife, or other tool, attempts to remove material from a potted or sealed container while stopping short of tripping a sensor. This attack is much more effective than might be thought. If the attacker is dexterous and has good hand-eye coordination, extremely delicate work can be accomplished.

Mechanical Machining: This method removes much material, very precisely, in the shortest time. Its disadvantages lie in the fact that there is little or no feedback. This frequently causes cuts that are too deep. If the cutter is conductive, it may be detected by the tamper detector.

Water Machining: Water machining is a very precise method for material removal. The 'cutter' can be non-conductive (if the water is pure), does not dull, and is very effective for all but very soft materials. Its chief disadvantage is that water machining equipment is typically very large. However, in situations where cost and size are a concern, but time is not, a directed slow, steady, drip of water will effectively cut through many materials given sufficient time.

Laser Machining: This technique has many of the same advantages as water. One disadvantage of laser machining is that the process may generate a great deal of heat. The laser must be tuned for the material of interest, e.g. EXCIMER (U.V.) lasers are excellent for ablating organic materials (such as epoxy).

Chemical Machining: Almost any material can be dissolved. Jet Etch[2] and similar commercial tools are very good for removing coatings and potting material cleanly. These techniques work by using a high-pressure, very precise spray of a solvent or acid to dissolve away the material. The solvent or acid may be heated to increase effectiveness. The main disadvantage is the potentially high conductivity of highly ionic cutting liquids, which may cause short circuits.

[1] If the data has an extremely short duration of value, or the audit period is excessively long, there may be no effort to cover the evidence.

[2] Jet Etch is a commercial product commonly used for removal of semiconductor surface coatings for analysis.

Shaped Charge Technology: Shaped charge technology has become commonly available to the degree where that charge precision welding and cutting sample kits are available to universities to promote the technology. These techniques have the advantages of being very accurate and being extremely fast. The penetration speed can approach 25,000 ft/sec. At these hypersonic speeds, a package can be penetrated and circuits disabled before they can respond. For example, a memory zeroing circuit can be disabled before the energy can be removed from the memory. This could give the attacker from a few seconds to a minute to finish entering a package and to reapply power to the memory before its contents decay.

TEMPEST: This is a passive attack. Electromagnetic emanations from a computer, or other electronic device, can be detected at a distance and decoded to determine contents or behavior. The distance can be many hundreds to a thousand feet or more. Power supply current profiles can also be measured to determine circuit activity.

Most information on TEMPEST is government classified in the interests of national security. However it is well known, and has been demonstrated, that a video display or serial communication line can be tapped at distances of hundreds of feet. Recently more aspects of TEMPEST technology have been independently invented/discovered in the commercial sector. Smart cards have been successfully attacked by means of studying their power supply current [10,4], and others [11] have developed new approaches to using this method.

Energy Attacks: These attacks are both of the contact and non-contact variety. However even the non-contact attacks usually require close access to the system.

Radiation Imprinting: By irradiating CMOS RAM in the X-Ray band (and possibly other bands), the contents can be 'burned in' such that power down or over-write will fail to erase the contents.

The basic imprinting attack uses radiation to imprint the CMOS RAM used to store cryptographic keys or other secret data, then the unit is physically breached without regard for power down or rewrite mechanisms. The RAM may then be read at leisure.

Temperature Imprinting: CMOS RAM will retain its contents with the power removed for seconds to hours when the temperature of the RAM is lowered. This effect starts at just below freezing. Over-writing will erase the contents.

High Voltage Imprinting: By 'spiking' CMOS RAM with short duration, high-voltage pulses, it may be possible to imprint the contents in a manner similar to radiation imprinting. This technique has not been verified by the author.

High or Low Voltage: By changing Vcc to abnormally high or low values, erratic behavior may be induced in many circuits. The erratic behaviour may include the processor misinterpreting instructions, erase or over-write circuitry failing, or memory retaining its data when not desired.

Clock Glitching: By lengthening or shortening the clock pulses to a clocked circuit such as a microprocessor, it's operation can be subverted. Instructions or tests can be skipped or generally erratic operation can be induced [2].

Circuit Disruption: This area has not yet been studied in depth by the author, however it is known that strong electromagnetic interference may cause disruption in noise-diode type random number generators and computing circuits.

Electron Beam Read/Write: The electron beam of a conventional scanning electron microscope can be used to read, and possibly write, individual bits in an EPROM, EEPROM, or RAM. To do this the surface of the chip must be exposed first, usually via chemical machining. This is a very powerful attack once the chip is exposed since buried, normally non-readable, keys and secrets can possibly be stolen and/or modified.

IR LASER Read/Write: Silicon is transparent at IR frequencies. Because of this, it is possible to read and write storage cells in a computing device by using an IR LASER directed through the bulk silicon side of the chip. By going through the bulk side there is no need to jet etch or otherwise remove the device's passivation.

Imaging Technologies: Any of the current imaging technologies including X-Ray, tomography, ultrasound, etc. can all be used to visualize the contents of a sealed or potted package. This can assist the attacker by pinpointing areas of vulnerability, identifying printed circuit card layout, showing part placement, and possibly identifying specific parts.

3.2 High Technology Defenses

The detection methods below fall into three categories: preventing intrusion, detecting intrusion, detection of noninvasive energy attacks (cold, radiation, etc.). After detection, there are various methods of response. Each method must be examined when choosing the design point. For example, a design that calls for a low temperature sensor must take into account the temperatures which the unit could be exposed to while in transport.

Tamper Resistant: This is basic bank vault technology. For example, an automated teller machine required a one inch thick mild steel case which enclosed another one-inch thick cash box [3]. These types of systems also resist theft by means of bulk. Another approach is to attach the device to the tamper barrier so firmly that the attempt to separate the layers, or to penetrate the protection, results in the destruction of the protected device.

Hard Barriers: Steel, brick, ceramics, etc., can all be used as effective barriers. As noted above, this may also help to inhibit theft.

Single Chip Coatings: This technique is used to prevent attack on the single chip level (e.g. pico-probing). The surface of the chip may not be probed with the coating in place and these coatings are applied so that removal will damage the chip beyond reclamation. This is a very complex topic as new chemistry is constantly being developed.

Insulator Based Substrates: To prevent an attacker avoiding a protective coating by using an IR LASER technique, the bulk silicon must be replaced with a material that is not transparent at useful frequencies. Silicon/Metal Oxide (SiMOX), Silicon-on- Sapphire (SOS), or other silicon-on-insulator technologies, combined with advanced passivation represent the highest level of passive, single chip, protection. One must still carefully evaluate the possibility of using surface grinding techniques to thin the substrate to the point of transparency.

Special Semiconductor Topographies: To prevent scanning electron microscope or pico-probing attacks, even in the presence of chemical machining or other techniques that can remove coatings, a chip can be designed so as not to expose critical structures without removing active layers of the device.

Tamper Evident: Tamper evident systems are not designed to prevent attack or entry into the protected area. They are designed such that entry *will* leave evidence to be discovered during physical audit.

Brittle Packages: The device is sealed in a package that is made of ceramic, glass, or another frangible material. If an attempt is made to enter the package, it cracks or shatters, leaving evidence.

Crazed Aluminum: The package is made from aluminum or other similar material, which has been heated (usually above 1000 degrees F.) and quenched. This heat treating causes a myriad of shallow, web-like cracks to appear on the surface. These cracks, like a fingerprint, are unique to each piece. The case can be photographed and subsequently audited using the photograph and optical comparison devices.

Polished Packages: Similar to crazed aluminum the package is inspected for changes in surface appearance. In this case any mark at all represents an attempted breach.

Bleeding Paint: Again, the surface quality is the auditable characteristic. Paint of one color is mixed with micro-balloons containing paint of a contrasting color. If the surface is marred, the other color "bleeds' onto the surface.

Holographic Tape: The surface of tape, with a very firm adhesive, is printed with a holographic image similar to the kind used on credit cards. This kind of tape is moderately difficult to forge, and it is constructed so that attempts to remove it will damage it (the tape may be scored to promote tearing when removal is attempted). This is good for checking to see if doors or covers have been illicitly opened. Recently there have been several incidents of holograhic seals being counterfeited.

Tamper Responding Sensor Technology: Tamper sensors cover a wide variety of devices, like the tamper evident devices above. Each type of sensor is designed to detect a particular type of intrusion. Like the example above of the automated teller machine and its steel case, certain designs are better suited for particular environments than others.

Voltage Sensors: Voltage sensors are useful in almost any design that requires proper power delivery for correct operation. Both high and low voltage can be a deliberate or accidental attack. To guarantee correct operation of circuits all power supplies should be monitored. Any excursion outside of nominal operating range should be considered an attack, and response should be engaged. References for monitors should be independent of power supply variations.

Probe Sensors: Probe sensors form a large family of active tamper barriers. Individual designs may feature tamper resistance or evidence, as well as tamper detection for additional security. Some designs are more or less costly, or heavy, or manufacturable, than others.

Wire Sensors: Thin wire wrapped around the package to be protected and then potted forms the intrusion sensor. Ideally the wire should have a high resistance so the wire can be used as a distributed resistance, so small changes can be detected as well as opens and shorts. If the wire is folded back over itself, or wound as multiple parallel strands, the sensitivity is increased because two adjacent wires may be electrically distant. So shorting two wires gives a larger signal than would two adjacent strands on a continuous wrap. The insulation on the wire should be as similar to the potting material as possible in both appearance and chemistry. This makes machining more difficult because no hints

as to the whereabouts of the wire are given. Chemical attacks are made more difficult because of the difficulty of dissolving the potting without dissolving the insulation and causing shorts. It is also an advantage if the wire is made from a material which is difficult to attach to.

Printed Circuit Board Sensors: A sensor similar to the wire sensor above can be made for a much lower cost by printing the wiring onto a printed circuit board. However, the regular spacing of the lines and the usual copper conducting material give somewhat less security. This is due to the ease with which the conductors may be isolated, owing to the regularity of a rigid printed circuit board. Once a conductor is located, it is very easy to attach another wire to it for the purpose of giving the tamper detection circuitry false information. However, with good potting material and small lines, this design gives moderate security.

Flexible Printed Circuit Sensors: This design incorporates the best features of the previous two. The flexible surface helps break up the regularity of the surface planes. The lines can be made of silk-screened conductive paste, which allows high resistance. It is even better to use lines made from a conductively doped version of the same material used for final potting. The realm of package shapes is wider because the package can be "gift wrapped' with the material, then potted. Also, the narrow screened lines will be much more difficult to find without breaking. Multiple layers can be used for additional security.

Stressed Glass Printed Circuit Sensors: Metal, or metal oxide, lines can be printed on glass, in a manner similar to a printed circuit board sensor. Contacts to the glass can be made using elastomeric 'Zebra' connectors. Stressed glass can be obtained that is virtually impenetrable without the glass breaking. This method is very good for large flat surfaces, or possibly, for secure doors.

Stressed Glass with Piezo-Electric Sensor: Using the same glass as in the previous example, this sensor uses a piezo-electric element to signal the breakage of the glass. The force of stressed glass breaking is enough to induce a large signal from a piezo-electric device attached to the inside of the glass.

Piezo-Electric Sheet: Plastic piezo-electric sheets can be used as probe barriers. If an area protected by a piezo-electric sheet is probed or punctured, an electric charge is generated proportional to the force applied. This charge can be measured and used to activate tamper response circuitry. There are problems with this application because of sensitivity to pressure and vibration, both making the design too sensitive to environmental conditions, and potentially insensitive to slow puncture attacks.

Bulk Multiple Scattering: This sensor uses the scattering properties of coherent light through bulk materials to create a very sensitive probe sensor based on measuring the optical speckle pattern.

Motion Sensors: These sensors are typically used to sense motion in an area or box. They are often need to be used in pairs because each type can sometimes cause a false positive or can miss under unusual conditions. An infrared sensor can trip falsely when the first rays of the sun fall on the protected package through a window.

Ultra-sonic: Ultra-sonic sensors average a picture of the protected space via ultra-sonic projection and reflection. They can be very effective, but can have false positives due to air currents, etc.

Microwave: Similar to ultra-sonic, with the same strengths and weaknesses, but at a higher frequency. The material of the walls of the protected area have to be taken into account with this type of system since some non-metallic materials can be transparent at these frequencies. This can cause false positives due to activity outside of the protected region.

Infra-red: This sensor is not typically sensitive to air currents or the like, but these systems have been known to trip due to light (and heat) changes due to sunrise through windows when the averaging is too sensitive. They are most useful for detecting warm bodies, people, animals, etc. A tool at ambient temperature will probably not be noticed unless it was moved to suddenly block an infrared radiating source that the sensor already 'sees.'

Acceleration Sensors: These sensors are used to detect movement or vibration. Their primary uses are to prevent theft, and to detect drilling or hammering.

Solid State: This sensor detects a beam of light reflected by mirrors that are attached to flexible mounts, or a piezo-electric device and a small mass. They are quite sensitive and reliable.

Micro-switches: Micro-switch motion sensors use mercury or pendulums to detect motion. They are lower in cost than solid state devices, but are less sensitive, and are more prone to failures. However, a liquid mercury switch can be reliable and virtually without wear.

Radiation Sensors: Radiation sensors are used to detect attempts at radiation imprinting. These sensors are most important for remotely located systems which could be taken into a laboratory and attacked.

Flux Sensors: Flux sensors sense the real-time radiation intensity. The advantage of this type of circuit is that it can be very low cost. The disadvantage is that this sensor has no cumulative memory (total dose measurement). If the data is invariant over a long period of time, low levels of radiation (below the sense point) can imprint the data. Given the power and cost budget for typical physical security systems, integrating the flux reading is too costly. So a compromise must be struck as to flux level trip point vs. minimum time to imprinting.

Phototransistors can be very effective radiation flux sensors. The circuit is the same as is used for light measurement, however a higher gain is typically required. The typical problems with this circuit are that the sensitivity in the radiation band of interest is usually not specified by the manufacturer and must be determined by testing, and that the sensors tend to degrade with time and exposure to radiation.

Dosage Sensors: These sensors store the total radiation dose over time. Total dose is the best indicator of imprinting in CMOS SRAM. Unfortunately, at this time there are no available dosage sensors which are small, low cost, low power, and directly readable.

Temperature Sensors: Temperature sensors are well-known and readily available at all cost performance points.

Tamper Responding - Response Technology: The methods of tamper response technology discussed here are means of removing data from RAM circuits which presumably contain secret information. This is currently the most common method of storing such information because the retention is reliable and the erasure is reasonably so. If one were to use the highest level of technology available to attempt recovery of data that had been stored and then erased on almost *any* known media, there is little, outside of physical destruction, that can prevent recovery.

RAM Power Drop: This is the most straightforward method of data erasure. If aided by a crowbar circuit that supplies a very low impedance path from Vcc to ground, it is reliable if imprinting protection (temperature sensing and radiation sensing) has been employed. Since there is a tendency for RAM contents to imprint over time, any information that is to be stored in RAM for long periods should be regularly scrambled, inverted, or otherwise changed to prevent imprinting.

RAM Overwrite: This method has had the widest acceptance in government specifications [17], however in a catastrophic condition it is difficult to guarantee that reliable power will be available to operate the over-write circuit. The common method is to over-write some number of times with all 0's, then all 1's.

It would seem random or pseudo-random data would be more effective, but this has not been shown. It would also take even longer to complete the overwrite since the data would have to be generated.

Physical Destruction: This is the only method of data erasure that is completely reliable. Destruction can be accomplished with a minimum of overt violence. The occurrence would barely be detectable at the surface of a metal hybrid package. Nonetheless, this method is typically reserved for the most sensitive circumstances.

3.3 Operating Envelope Concept

One of the main problems encountered while implementing physically secure systems is the prevention of the class of attacks that cause erratic operation. This can occur when the operating point is pushed to the boundaries of the operating range. For example, running the circuit at either marginally high or low supply voltages may cause erratic operation of the circuit such that secret information could be leaked. If one considers the possibility of adjusting both temperature and voltage, the problem can become even more complex.

Manufacturers define the operating range of the components that they make, but often the specification is incomplete. It can be incomplete because no one ever intended the part to be used in some particular way, and the manufacturer, justifiably, doesn't want to deal with the problem. In general, designers can design circuits that stay within prescribed limits and the circuit functions properly. For example, if the circuit is run at too high a temperature while at too low a supply voltage, the condition is undefined. This may open the system to attack.

It is the physical security designer's responsibility to determine the safe operating envelope of the circuit under all conditions, and to provide safeguards to detect conditions outside of the acceptable operating envelope. If these conditions are detected the response circuitry must protect the secret data. This is the basic idea behind the environmental failure protection requirement in FIPS 140-1 [9]. If conditions leave the safe operating envelope in a non-catastrophic manner (e.g. Vcc drop during power down), the system should be stopped (or held reset). If conditions leave the safe operating envelope in a catastrophic manner (e.g. ambient temperature exceeding safe operating range), the critical data should be erased and the system should be prevented from operating.

Secure designs should also employ good engineering practice to prevent improper clock signals from reaching sensitive circuits by use of phased lock loops (PLLs), or similar techniques. Power analysis attacks should be prevented by designing in adequate power filtering to reduce information leakage.

4 A High Technology Physical Security Design Example

The following design began as a concept and has now been developed.

A small printed circuit board contains the microcomputer, cryptographic, and tamper detection/response circuitry. The circuitry on the card includes voltage, temperature and radiation sensors to protect the battery backed-up CMOS SRAM from becoming imprinted as well as circuitry to erase the SRAM by power down with a crowbar to ground on the SRAM power pin. Circuitry also guarantees that the contents of the rest of the system (key registers, microprocessor contents, etc.) are lost on tamper. Additional circuitry monitors the tamper detection screen which surrounds the entire assembly. The tamper detection screen is constructed of conductive organic lines on a polyester substrate. These lines are arranged in a configuration so that changes in the resistance of the lines caused by shorting, breaking, or otherwise damaging the lines are detectable. The assembly is then potted using an organic material similar in composition to the conductors, in a metal case which serves as an electrcal shield [8].

If the design is examined it can be seen that a number of attacks have been anticipated and guarded against. The voltage, temperature and radiation sensors ensure that cold and radiation attacks will not succeed in causing imprinting. The voltage sensors protect from imprinting and disruption. The SRAM power down and crowbar circuit will reliably erase the SRAM in the event of an attack. The SRAM devices used in the design have been tested to assure erasure when the power down circuit is activated.

The probe sensor is sensitive to very small probes and the potting makes machining very difficult because the uneven surface of the polyester under the hard potting would make cutting too deep quite likely. Even if the screen were reached successfully, the lines are very difficult to manipulate and would most likely be damaged in the attack attempt, triggering the power down. The metal case protects additionally from machining as well as acting as an electrical interference barrier (Faraday cage).

This design has also been tested, and has been found not susceptable, to power analysis attacks.

This design is representative of the commercial state of the art in physical security design, and has been validated at FIPS 140-1 level 4 overall [13,14].

5 Conclusions

Physical security devices like those described here are becoming desirable in areas where a technical means for ensuring data secrecy is required. As data values climb, the motivation for using physical means to extract data from computing systems is steadily increasing. System design must meet this growing need for protection.

As with any developing technology, the design point and performance must be constantly reviewed. The technology of potential adversaries, as well as the value of the data which motivates these individuals, is increasing. So the technology and quality of the protection must keep up with the skills of the attackers.

Acknowledgements

The author would like to thank several people for their contributions to this work. Years ago, Gus Simmons gave some critical pointers that led me to deeper work in physical security, I thank him for that! Steve R. White and Bill Arnold at the IBM T. J. Watson Research Center, and Glen Double from IBM Charlotte, were central to this work for many years. The other members of the IBM 4758 team at Watson, Elaine Palmer, Sean Smith, Ron Perez, Mark Lindemann and Joan Dyer have all helped and shared ideas. Huge thanks to my wife Pat, for patience, support and copy editing.

References

1. R. Anderson, M. Kuhn, 'Tamper Resistance - A Cautionary Note', The Second USENIX Workshop on Electronic Commerce Proceedings, Oakland, California, November 18-21, 1996, pp 1-11, ISBN 1-880446-83-96.
2. R. Anderson, M. Kuhn, 'Low Cost Attacks on Tamper Resistant Devices'
3. R. E. Anderson, 'Bank Security', Butterworth Publishers 1981, pp. 9l-93.
4. S. Chari, C.S. Jutla. J.R. Rao, and P.Rohatgi. 'A Cautionary note regarding evaluation of AES candidates on smart cards'. Proceedings of Second AES Conference, Rome, Mar 1999.
5. David Chaum, 'Concepts for Design of Tamper Responding Systems', Advances in Cryptology, Proceedings of Crypto '83 , Plenum Press 1984, pp.387-392.
6. Andrew 1. Clark, 'Physical Protection of Cryptographic Devices', presented at Eurocrypt '87, Amsterdam.
7. 'Department of Defense Trusted Computer System Evaluation Criteria', U. S. Department of Defense, 5200.28 STD
8. G. P. Double, 'Physical Security for Transaction Systems: A Design Methodology', IBM Technical Report, TR 83.227 IBM 1990.
9. 'Federal Information Processing Standard 140-1: General Security Requirements for Equipment Using the Data Encryption Standard', National Institute for Standards and Technology
10. P. Kocher, J. Jaffe and B. Jun. 'Introduction to Differential Power Analysis and Related Attacks.' Manuscript, Cryptography Research, Inc. 1998.
11. M. Kuhn and R. Anderson, 'Soft Tempest: Hidden Data Transmission Using Electromagnetic Emanations', Information Hiding 1998, LNCS 1525, pp. 124-142, 1998.
12. W. L. Price, 'Physical Security of Transaction Devices', NPL Technical Memo DITC 4/86, National Physical Laboratory, Jan, 1986.
13. S.W. Smith, S.H. Weingart, 'Building a High Performance, Programmable Secure Coprocessor.' Computer Networks (Special Issue on Computing Network Security). 31: 831-860. April 1999.
14. S.W. Smith, V. Austel, R. Perez, S. Weingart. 'Validating a High-Performance, Programmable Secure Coprocessor or, the World's First FIPS 140-1 Level 4.' 22nd National Information Systems Security Cconerence, October 199.
15. S. H. Weingart, 'Physical Security for the uABYSS System', Proceedings of IEEE Symposium on Security and Privacy 1987, IEEE Publications, pp. 52-58.
16. S. H. Weingart, S. White, W. Arnold, and G. Double, 'An Evaluation System for the Physical Security of Computing Systems', Proceedings of the Sixth Annual Computer Security Applications Conference 1990, IEEE Publications, pp. 232-243.
17. U. S. Federal Standard 1027 Telecommunications: General Security Requirements for Equipment Using the Data Encryption Standard.

Software-Hardware Trade-Offs: Application to A5/1 Cryptanalysis

Thomas Pornin and Jacques Stern

Département d'Informatique, École Normale Supérieure
45 rue d'Ulm, 75005 Paris, France
{thomas.pornin,jacques.stern}@ens.fr

Abstract. This paper shows how a well-balanced trade-off between a generic workstation and dumb but fast reconfigurable hardware can lead to a more efficient implementation of a cryptanalysis than a full hardware or a full software implementation. A realistic cryptanalysis of the A5/1 GSM stream cipher is presented as an illustration of such trade-off. We mention that our cryptanalysis requires only a minimal amount of cipher output and cannot be compared to the attack recently announced by Alex Biryukov, Adi Shamir and David Wagner[2].
Keywords: A5/1, GSM, stream cipher, FPGA, cryptanalysis, trade-off.

1 Introduction

There are two main species of computer devices that are used by cryptanalysts: generic all-purposes workstations, and specialized hardware devices. Among the latters, Field Programmable Gate Arrays are more and more used, since they give a good performance/cost ratio and a fast and cheap development cycle.

Operations that are easy to implement on a FPGA include all bit permutations, shifts, bitwise logical operations, and small lookup tables. This makes them especially well-suited for implementing block ciphers such as DES, and all stream ciphers and random generators using Linear Feedback Shift Registers (LFSR). However, the low-level structure of such devices makes it almost impossible to implement of a high-level algorithm whose behaviour is dependant on the input data. A branching process or a recursive search in a tree are definitely out of reach.

Workstations, on the contrary, are good at running complex algorithms, since conditional execution, function calls and stack memory structures are natural on these platforms. They are also especially optimized at performing complex mathematic operations such as integer multiplications or floating point calculations. Yet, they are ineffective at more simple operations, in proportion to their cost: an expensive 21264 Alpha processor will perform only four bitwise logical operations per cycle on 64-bit registers, despite its over 15 millions transistors.

We present here a study on a trade-off between these two technologies. The chosen algorithm is A5/1; this stream cipher is used in GSM mobile phones to ensure confidentiality of "over the air" communication. A5/1 was published in

Ç.K. Koç and C. Paar (Eds.): CHES 2000, LNCS 1965, pp. 318–327, 2000.

[1] unofficially, but Alex Biryukov, Adi Shamir and David Wagner claim in [2] that they received confirmation from the GSM organization that this design is the true A5/1 as used in GSM phones. Therefore we will assume that the A5/1 described here is indeed the correct algorithm; anyway, this algorithm is merely used as an illustration of our technique.

Recently, Alex Biryukov, Adi Shamir and David Wagner presented in [2] an impressive attack against the A5/1 cipher; this attack is a time-memory trade-off that requires a non-negligeable amount of known plaintext (about 25000 bits). Since the internal state of A5/1 is only 64-bit, it should be recoverable with only 64-bit of known plaintext; we therefore consider this framework, where only 64 consecutive bits or so of plaintext (and the corresponding ciphertext) have been intercepted. Moreover, the time-memory trade-off of [2] is made very effective due to many features of A5/1 that allow some smart optimizations. We do not use such features, and our work should be applicable to other similar ciphers.

The hardware used is a Compaq XP-1000 workstation (21264 Alpha processor at 500 MHz) and Compaq (formerly Digital) Pamette cards; a Pamette is a PCI card that includes five Xilinx 4010E FPGA. One of these FPGA is used to handle the PCI bus; there is room for some SDRAM connected to two of the FPGA. At the time of writing this paper, an XP-1000 is a 3000$ workstation, and a Pamette costs about 1000$.

2 Description of A5/1

A5/1 is a neat design that uses a very small amount of silicium when implemented in hardware. It includes three LFSR, with a clocking sequence depending on the internal state of the three registers. It outputs a stream of bits that is combined (by mean of an exclusive or) with the data to encipher.

The three LFSR are of length 19, 22 and 23 bits. At each clock cycle, a majority bit is calculated, from the three middle bits of the registers; those registers which middle bit agrees with the majority are shifted. Then the output bit is the exclusive or of the three final bits of the registers. Figure 1 illustrates this mechanism. A full description of the algorithm may be found in [1].

The majority clocking implies that, at each clock cycle, there are four possible moves:

- register 1 and 2 are shifted
- register 1 and 3 are shifted
- register 2 and 3 are shifted
- all registers are shifted

The internal state is loaded with a 64-bit session key and a 22-bit known counter; the cipher is then ran for 100 cycles and the corresponding output bits discarded, and then 228 bits are produced for enciphering the data. Then the cipher is reset, with the same key and the next counter value. The key can easily be recovered from the internal state at any moment with a critical branching process exposed in [3]; therefore, once one internal state of A5/1 has

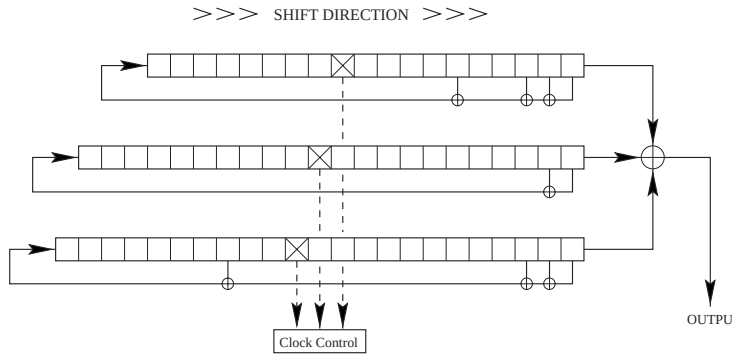


Figure 1: The A5/1 Stream Cipher

been revealed, the cryptanalysis is to be considered complete, since the same session key is used throughout the entire phone conversation.

In GSM phones, the session keys are produced with another algorithm, which might depend on the operator. Marc Briceno, Ian Goldberg and David Wagner, who published in May 1999 in [1] the first complete description of A5/1, claim that all the implementations they checked used 54-bit session keys (that is, 64-bit keys with 10 fixed bits set to 0). Although other operators could decide otherwise, it is probable that this convention will be maintained by operators in the name of backward compatibility. Still, our work does not make use of this feature.

3 Software Cryptanalysis of A5/1

A first cryptanalysis of A5/1 was first informally presented by Ross Anderson, who published in 1994 [4] an alleged description of A5/1 (which turned out to be mostly correct, except for the position of bits for clocking and linear feedback). The idea is to guess the two first registers, and half of the third register, which is basically enough to know the clocking sequence and deduce the second half of the third register by solving a system of linear equations. This attack is applicable to the real A5/1, with a workload of about 2^{52} guesses (each implying the resolution of a system of a dozen linear equations).

Then Jovan Golić presented, in 1997 [3], a complex cryptanalysis with an average complexity of slightly above 2^{40} operations; however, each operation is a resolution of a 64×64 linear system, and some of the assumptions used to get the claimed complexity are somehow irrealistic since they lead to an overly complex and slow implementation.

The Golić attack is, basically, guessing the clock sequence for a given number of clock cycles, adjusting it if necessary by adding some more guesses; knowing the clock sequence, each output bit is a linear equation of known internal state bits. The guess also gives other linear equations, which describe the majority function. When enough equations are obtained, the system is reversed, and the

potential initial state is recovered and tested against the remaining known output bits.

We implemented a simplified version of the Golić attack. This is done by backtrack in a tree representing at depth n the different internal states after n clock cycles. So each node has four subtrees, since there are four possible moves at each cycle. Each guess in the backtrack process is taking one of the four branches; this gives us three equations:

- two equations represent the clock control calculation
- one equation is the calculation of the output bit

These equations are linear in $\mathbf{Z}_2$, with 64 unknown values. These values are the 64-bit initial state, and, at each step of the algorithm, each bit of one LFSR is a linear combination of several bits of the initial sate of the same LFSR; this combination depends only on the number of times the LFSR has been clocked since the initial state. So, if we call c_1, c_2 and c_3 the clocking bits of the respective three LFSR at one step, and guess that registers 1 and 2 move, and register 3 does not, we get the following two equations:

$$c_1 + c_2 = 0$$
$$c_1 + c_3 = 1$$

The third equation is similar: if, after clocking, the end bits of the three LFSR are respectively e_1, e_2 and e_3, and the output bit is v, then we have the following:

$$e_1 + e_2 + e_3 = v$$

We maintain, during the backtrack, a system of such equations describing the previous steps of the algorithm starting from the initial state; this system is triangular, which means the following: for each equation n, there exists one of the unknowns such that its coefficient in equation n is 1, and such that in all following equations (equations $n + 1$, $n + 2$, ...) its coefficient is 0. When the system is complete (64 equations), equation 64 is:

$$x = k_1$$

where x is one of the unknowns, and k_1 is a constant value (0 or 1). Equation 63 is:

$$y + k_2 x = k_3$$

where y is another unknown value, and k_2 and k_3 are constants. So, once x value is known, y is known too. We can go on with this process up to equation 1, and therefore simply recover the whole 64 unknown values. This is the standard, well-known method of linear system solving, due to Gauss.

So, when we add one equation to the yet incomplete system, we need to perform the Gaussian elimination of this equation relatively to the preceedings. If we call u_i the unknown value whose coefficient is 1 in equation i and 0 in all equations j for $j > i$, we apply the following algorithm when we add equation n to the system:

1. call X the equation to add
2. for i from 1 to $n-1$
3. if u_i has a coefficient 1 in X, add equation i to X
4. next i
5. append X to the system
6. find the first non-zero coefficient in X, call u_n the corresponding unknown

The last action of this algorithm may fail, if all coefficients of X are set to 0 by the elimination process. Then X is either $0 = 0$ or $0 = 1$. If we get $0 = 0$, this means that the new equation can de deduced linearly from the preceedings, so we just throw it away and keep on with the backtrack. However, if we get $0 = 1$, we are lucky: we know that the path in the tree of possible clocking sequences, up to the point that has been reached, is wrong. If this happens at clocking step 19, we go back to clocking step 18, and assume that the last guess was wrong. So we forget the equations added by that last move, an go on with another guess for that move. This is where we optimize the Golić attack: we can keep all equations corresponding to step 1 to 18, and we do not have to perform the Gaussian elimination on them again.

This calculation can be implemented effectively on modern workstations: since each coefficient is 0 or 1, it can be stored as one bit. Each equation is a 65-bit word (64 bits for the 64 coefficients, one bit for the constant on the right hand side of the equation). An addition of two equations is a bitwise exclusive or, a native operation on modern processors. Finding the first bit set to one in a 64-bit word may be performed by a dichotomic process, which gives the result in 6 masking/compare/shift group of operations.

Once we have 64 linearly independant equations, in a triangular representation, we might solve the system, recover the initial state, and run A5/1 with this initial state to see if it matches the known output; however, it is more efficient to keep on with the elimination. At each step, since the system is complete, all added equations will be reduced to either $0 = 0$ or $0 = 1$. Only one of the four possible clocking steps will produce two $0 = 0$ equations (since the system is complete, it contains the whole information on the execution of A5/1, and the clocking behaviour is deterministic given this information), and the third equation, depending on the ouput bit, will yield $0 = 0$ with probability 0.5, and $0 = 1$ otherwise. So, on average, we must go two steps further in the backtrack process to check the correctness of the guessed clocking sequence (this means six more equations reduced).

Experiments show that total eliminations (equation X has a left hand side equal to 0) are very rare before step 21; this is coherent with the intuitive idea that we cannot find anything on the internal state of A5/1 before the registers have wrapped around. The complexity of the backtrack is therefore the expected value $4^{64/3} \times 6$, which is about $2^{45.3}$. Each operation is the Gaussian elimination of one equation according to an average of about 64 preceedings linear equations (most of the computation time is spent in the leaves of the tree, where the linear system is complete, or almost). This is the complexity for the whole search; on

the average, we find the correct clocking sequence after exploring half of the tree, so the complexity is $2^{44.3}$. We claim that this is the same complexity as the Golić one, expressed in a more realistic unit.

Our implementation takes 400 days on a Compaq-XP1000 (21264 Alpha processor at 500 MHz) to explore the full tree; this yields an average software-only cryptanalysis time of 200 days on one workstation.

4 Hardware Cryptanalysis of A5/1

4.1 Description of the FPGA Pamette Card

The Pamette card includes five Xilinx 4010E FPGA chips; one of them is dedicated to the handling of the PCI bus. Each 4010E is a matrix of reconfigurable units called Configurable Logic Blocks (CLB).

Each CLB includes:

- two $4 \rightarrow 1$ reconfigurable lookup tables
- one $3 \rightarrow 1$ reconfigurable lookup table, two entries of which are the outputs of the two preceeding lookup tables
- two one-bit registers

Figure 2 gives an insight of a CLB. The two $4 \rightarrow 1$ and the $3 \rightarrow 1$ functions are fully configurable (they are implemented as lookup tables). There are four outputs, two of them corresponding to two one-bit registers; each register is controlled by an "enable" input, that can be set either to 1 (the register always updates) or connected to one output of a CLB (possibly the same). The initial value of each register is either 0 and 1, and this is configurable.

There are $24 \times 24 = 576$ CLB in a 4010E chip; the interconnecting matrix is also highly configurable; up to eight parallel signals can be carried between two rows of CLB.

The Xilinx chips are connected with each other through 16-bit and 8-bit busses; two of the four available chips are connected to optional static RAM, and to the fifth chip (the PCI-handler) with a 32-bit wide bus. The whole card may be clocked up to 66 MHz, depending on the design (to run at 66 MHz, there must be only one CLB and no long routing between two registers).

A more complete description of a Xilinx chip is available from Xilinx (see [5] for details). The Pamette itself is from Compaq (formerly Digital) and is described in [6].

4.2 Implementation of A5/1 on a Pamette

It is possible to implement A5/1 on a Xilinx 4010E with the following characteristics:

- At each cycle, one step of A5/1 is performed.
- It is possible to reload the LFSRs with new values in one cycle.

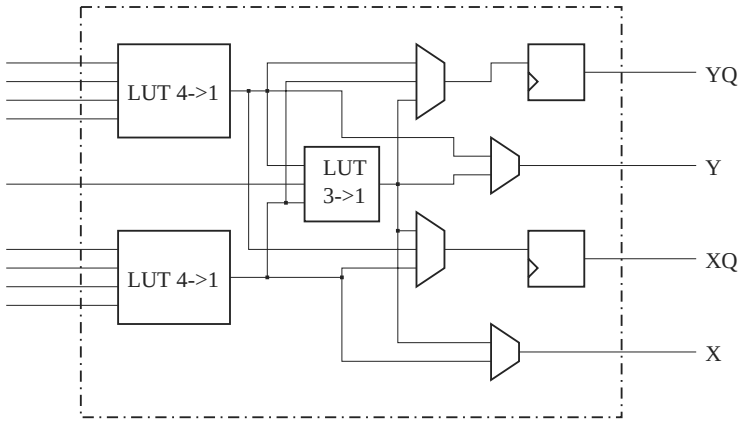


Figure 2: A Configurable Logic Bloc

– The resulting design runs at 50 MHz.
– 12 parallel instances of A5/1 may be put into each Xilinx chip.

The details of the implementation are fairly straightforward. The clocking bit is calculated from the three clocking bits of the registers, with one extra bit indicating that a new initial state must be loaded into the registers. This clocking bit is used as the "enable" input for the registers. The clocking bit calculation requires only half a CLB; we also use half a CLB for each bit in each LFSR (one bit register to store the value, one lookup table to feed the register with either the preceeding value, or a new value, when another initial state must be loaded). With the feedback computation (one half-CLB) and the comparison with the reference value (1.5 to 3.5 CLB), a whole instance of A5/1 requires only 36 or 38 CLBs (depending whether we want to compute 32, 64 or more output bits). So we may store twelve instances of A5/1 on each Xilinx chip and still have room for the synchronization clocks and the shared counter, which gives the successive initial states to try (the twelve instances try the same initial states except for some bits, so they share the counter).

Since the design is really compact (all critical data exchanges are local to one small area of the chip) and the computational depth (maximal number of CLB between two registers) is small (only 2 CLB at most must be gone through at each cycle), we can run the whole design at 50 MHz.

Therefore, we have 48 parallel implentations of A5/1 in one Pamette, that may try 64 A5/1 steps in 64 cycles. One more cycle will be needed for the reload of the state, so we can try up to 37 millions initial states per second per Pamette. This is faster than the best known software implementation of A5/1, described in [2], which can treat up to 8 steps at a time, but runs at the speed of a workstation's RAM. Pamette cards give a high degree of intrinsic parallelism.

So, if we want to perform an exhaustive search on the 64-bit internal state, we need about 15800 Pamette-years, that is 15800 years with one Pamette, or 1 year with 15800 Pamettes. We might want to take advantage of the alleged

fact that session keys are only 54-bit, not 64-bit. Then we must, for each guess, perform the 100 discarding states, which drops the Pamette efficiency to about 14.5 millions key tests per second; however, the exhaustive search workload is divided by a factor of $2^{10} = 1024$, which leads to a total effort of about 39 Pamette-years. This is reachable by many agencies and businesses around the world, although not quite efficient for daily cryptanalysis.

For completeness, we must add that the infrastructure needed is small: actually, there is no real problem of data bandwidth. Each instance of A5/1 runs isolated from the others, and the only data that has to be exchanged is the initial setting of the FPGA (but loading a Xilinx 4010E with a given design is a matter of milliseconds), and one bit from one instance of A5/1 to indicate that a matching initial state has been found. The controlling PC has a very simple job: it waits for the bit to be set, and measures the time taken from the beginning of the search. From this measure, the matching state can be narrowed to a set of only a few millions candidates, which can be precisely tried in a few seconds on the cheapest of nowadays PCs.

5 The Software-Hardware Trade-Off

An intuitive, information theory oriented point of view is that the minimal workload to cryptanalyse A5/1 is something like $4^{64/3}$. In this approach, the clocking sequence is considered as intractable; it cannot be controlled except with an exhaustive search. Since the initial state is 64-bit, we need 64 binary data in order to cryptanalyse A5/1. From each step, we get one bit from the ouput, and two bits from the clocking sequence, since four clocking steps are possible. We will not have our 64 equations until we have considered at least 64/3 steps, and then the exhaustive search will have cost us $4^{64/3}$ operations.

The software cryptanalysis presented in section 3 sticks as close as possible to this workload. On the contrary, the hardware exhaustive search is way above it, but may be ran on a really dumb but fast device. Indeed, any conditionnal code (and there is many in a backtrack) is a pain to implement on a FPGA; it usually ends up in reimplementing a complete cpu, which is a misuse of the hardware, since a real cpu of comparable cost will be much more optimized for this task.

The main idea of the trade-off is to make part of the job with a software implementation, but to jump over the "complexity barrier" with an hardware implementation. This is a trade-off between an increased workload and the possibility to perform part of this workload on an efficient hardware device.

The software part is the beginning of the software cryptanalysis. We perform an exhaustive search on the clocking sequence on the first n steps of A5/1 (for instance, with $n = 17$). Each guess will give us $3n$ linear equations in the initial state; the workstation will then solve each system, that is exhibit the $(64 - 3n + 1)$ 64-bit vectors that represent a basis for the affine subspace in $\mathbf{Z}_2^{64}$ which holds all the solutions to this $3n$ equations system ($(64 - 3n)$ vectors for the basis of the corresponding linear subspace, and one more vector as origin of the

affine subspace). Then these vectors are sent to the Pamette card, which then exhaustively tries all elements of this subspace as initial states; this is a matter of 2^{64-3n} executions of A5/1.

For instance, for $n = 17$, the software part will have to generate 2^{34} systems; solving each system is about twice the cost of performing the Gauss elimination on each of them. For each system, the affine subspace of solutions contains 2^{13} elements, so the workload for the Pamette Card is $2^{34} \times 2^{13} = 2^{47}$ executions of A5/1, at Pamette speed. With a correct balancing between the software and the hardware part (that is, an appropriate choice of n), we can achieve a much lower cryptanalysis time than a full-hardware or full-software solution.

There are two possible optimizations in this method:

- Most of the time, the $3n$ equations are linearly independant. It is therefore not necessary to handle the rare case where the reversing of the system gives either an impossibility or a wider subspace of solutions: we just discard these occurrences. Therefore, 5% or so of cryptanalysis are not successful; we believe this is acceptable. This does not actually improve performances but greatly simplifies the implementation.
- It is not needed to test in the Pamette each initial state against the whole 64-bit output. All we need to do is test against enough bits so that the average case is that no initial state matches, and so that it is very unlikely that two or more initial states match. For instance, if $n = 17$, me might try only 32 bits of output; one subspace every 32768 (on average) will contain a match, and this can be handled easily in software. The hardware testing will be twice faster than if all 64 bits had to be matched for in hardware.

The optimal choice of n heavily depends on the number of workstations and the number of Pamettes available (in fact, on the ratio between these two numbers). We give here numbers for one XP-1000 Alpha station, and two 4010E Pamettes; these are durations for execution of the workload:

n	**soft. load**	**soft. time**	**hard. load**	**hard. time**
16	2^{32}	0.09	2^{48}	22
17	2^{34}	0.39	2^{47}	11
18	2^{36}	1.7	2^{46}	5
19	2^{38}	7.1	2^{45}	2.5
20	2^{40}	30	2^{44}	1.3

The time figures are expressed in days; this corresponds to the full cryptanalysis, that is the worst case. The average cryptanalysis will take up half of the time given. We consider that the Pamette will try to match against 32 bits of output.

We see that, when there are two Pamettes for each workstation, the optimal n is 18, which allows cryptanalysis in 2.5 days on average. By comparison, with the same investment, we could have two workstations, that would perform the full software cryptanalysis in 200 days (average time: 100 days). So we have a factor of forty in performance, and still have some computing power available

on the workstation (some of which is used to check the about 2^{14} subspaces that contain an initial state that gives 32 correct output bits; this means 2^{24} software checks, that is only a few seconds on the workstation). A full hardware exhaustive search is definitely out of the question, even if we take benefit of the reduced session key size.

6 Conclusion

We showed how a right balancing between a tricky software and a dumb hardware implementations can dramatically speed up a cryptanalysis of A5/1. With a small investment (less than 20000$), it is quite possible to uncover an intercepted GSM communication in a realistic interception scenario: although we do not have a complete specification of the GSM protocol, we believe that it is easy to guess 64 bits of communication. This is, in our point of view, a much more applicable attack in the real world, than the (although impressive) attack from Biryukov, Shamir and Wagner, since this latter requires an average of two seconds of exact plaintext.

Morevover, we did not use any specific characteristic of A5/1 such as the position of clocking bits or the feedback function, so this study has a much wider impact than GSM privacy. All LFSR-based pseudo-random generators, with a data-controlled clock sequence, might be affected by this technique. We strongly suggest that such generators be given an internal state of 128 bits at least.

References

1. Marc Briceno, Ian Goldberg, David Wagner, *A pedagogical implementation of A5/1*, web publication, `http://www.scard.org/gsm/body.html`, 1999.
2. Alex Biryukov, Adi Shamir, David Wagner, *Real Time Cryptanalysis of A5/1 on a PC*, presented at FSE2000.
3. Jovan Dj. Golić, *Cryptanalysis of Alleged A5 Stream Cipher* Lecture Notes in Computer Science, Advances in Cryptology, proceedings of EUROCRYPT'97, pp. 239–255, 1997.
4. Ross Anderson, *A5 (Was: HACKING DIGITAL PHONES)* Usenet communication on sci.crypt, alt.security and uk.telecom, June 17th 1994.
5. The Xilinx web site, `http://www.xilinx.com/`
6. The Pamette main web site, `http://www.research.digital.com/SRC/pamette/`

MiniPASS: Authentication and Digital Signatures in a Constrained Environment

Jeffrey Hoffstein and Joseph H. Silverman

NTRU Cryptosystems, Inc., 5 Burlington Woods
Burlington, MA 01803, USA
jhoff@ntru.com,jhs@ntru.com

Abstract. We describe an implementation of the PASS polynomial authentication and signature scheme [5,6] that is suitable for use in highly constrained environments such as smart cards and wireless applications. The algorithm underlying the PASS scheme, as described in [5,6], already features high speed and a small footprint, and these are further enhanced by transferring computational overhead to the server to the extent possible. We also describe timing and footprint results from a prototype implementation.

Introduction

Secure public key authentication and digital signatures are increasingly important for electronic communications and commerce, and they are required not only on high powered desktop computers, but also on smart cards and wireless devices with severely constrained memory and processing capabilities. An authentication/digital signature scheme called PASS (Polynomial Authentication and Signature Scheme) was introduced in [5], and a slightly modified version with even better operating characteristics was described in [6]. It was asserted in [5,6] that PASS is ideal for constrained environments due to its high speed and small footprint. In this article we substantiate those claims by giving a detailed description of how to implement PASS on a small memory/low speed device such as a smart card. We also give the results of experiments using a preliminary implementation of these ideas.

The importance of public key authentication and digital signatures is amply demonstrated by the large literature devoted to both theoretical and practical aspects of the problem, see for example [2,3,8,9,11,13,14,16]. The widespread need for such applications makes the introduction of new schemes of interest to both the academic and financial communities, especially schemes which are based on well studied hard mathematical problems and which offer significant practical advantages in terms of speed and key size over existing methods.

1 A Brief Description of PASS

The PASS Polynomial Authentication and Signature Scheme is based on the hard mathematical problem of finding a binary polynomial $f(X)$ that takes on pre-

Ç.K. Koç and C. Paar (Eds.): CHES 2000, LNCS 1965, pp. 328–339, 2000.

scribed values $f(\alpha) \bmod q$ at a given collection of numbers $\alpha = \alpha_1, \alpha_2, \ldots, \alpha_n$. (This problem is equivalent to solving the closest vector problem in a certain lattice, see [5,6] for details.) Briefly, the Prover publishes the set of values $f(\alpha_i)$ as her public key, and she proves her identity by demonstrating that she possesses a binary polynomial taking those values.

One version of PASS was presented at CrypTEC '99 [5] and a modification based on the same principles, but with better operating characteristics, is described in [6]. Since our goal in this paper is to fit a fast and secure authentication scheme into a highly constrained environment, we will use the version of PASS from [6], but virtually all of our remarks apply also to the original version in [5]. (See also Remark 8.) In this section we will briefly review how the PASS scheme works. Further information and a detailed security analysis may be found in the cited papers.

A PASS scheme depends on the choice of a prime number q, and we set $N = q - 1$. A typical choice yielding a security level approximately equivalent to an RSA 1024 bit key (see [5,6] for a detailed security analysis) is

$$q = 769 \qquad \text{and} \qquad N = 768. \tag{1}$$

For higher security, one might take $q = 1053$ and $N = 1052$. For the remainder of this article, unless we specify otherwise, all computations with numbers are performed modulo q.

The basic objects used by PASS are polynomials of degree $N - 1$,

$$f(X) = a_0 + a_1 X + a_2 X^2 + \cdots + a_{N-1} X^{N-1},$$

taken with coefficients modulo q. Multiplication is accomplished using the rule $X^N = 1$, which leads to the multiplication formula

$$\left(\sum_{i=0}^{N-1} a_i X^i\right)\left(\sum_{i=0}^{N-1} b_i X^i\right) = \sum_{k=0}^{N-1}\left(\sum_{\substack{i+j\equiv k \pmod N \\ 0\le i,j<N}} a_i b_j\right) X^k. \tag{2}$$

Another way to view this multiplication is to write the coefficients of a polynomial as a vector

$$[a_0, a_1, \ldots, a_{N-1}],$$

and then the product of two vectors is the usual convolution product.

The other public parameter for PASS is a set

$$S = \{\alpha_1, \alpha_2, \ldots, \alpha_{N/2}\}$$

of distinct nonzero numbers modulo q with the property that if $\alpha \in S$, then also $\alpha^{-1} \in S$. For concreteness, we will fix a generator w modulo q (i.e., w is a primitive root modulo q) and take

$$S = \{w^i : N/4 \le i \le 3N/4\}. \tag{3}$$

In particular, there is no need to actually store the set S.

If $f(X)$ is a polynomial of degree $N-1$ with mod q coefficients as above, then its *Discrete Fourier Transform* (DFT) is a polynomial $\hat{f}(X)$ whose coefficients are the values of f. More precisely, we fix a generator w modulo q, and then

$$\hat{f}(X) = \sum_{j=0}^{N-1} f(w^j)X^j,$$

where remember that all numbers are computed modulo q. A well-known formula says that the coefficients of the original polynomial $f(X) = \sum a_i X^i$ can be recovered from the values of f via the equation

$$a_i = \frac{1}{N}\hat{f}(w^{-i}) = \frac{1}{N}\sum_{j=0}^{N-1} f(w^j)w^{-ij}. \tag{4}$$

(Here w^{-1} is the inverse of w modulo q, and w^{-i} is the i^{th} power of w^{-1}.)

Now suppose that we are only given some of the values of f, for example the set of values

$$f(S) = \{f(\alpha) : \alpha \in S\}.$$

There are a large number of polynomials which take these prescribed values (precisely, there will be $q^{N/2}$ of them). However, the PASS polynomial forming the private key will have the additional property that it is a *binary polynomial*, that is, all of its coefficients are 0 or 1. It is then a difficult problem to find the target binary polynomial $f(X)$ among the $q^{N/2}$ polynomials taking the correct values.

Remark 1. The security of PASS is based on the fact that it is difficult to simultaneously control both the values and the coefficients of a polynomial over a finite field. As indicated above, the values and the coefficients of a polynomial are (discrete) Fourier transforms of one another. Thus underlying PASS is the mathematical principle that it is difficult to simultaneously control the values of a function and the values of its Fourier transform. This principle is the discrete analogue (for finite fields) of the Heisenberg Uncertainty Principle. (A mathematical formulation of the Heisenberg Uncertainty Principle says that for suitably normalized functions f, the product $\|f\| \cdot \|\hat{f}\|$ cannot be made arbitrarily small.) As described in [5,6,12], this problem of finding a small polynomial taking some given values can be solved using lattice reduction methods (just as RSA can be broken using the number field sieve), but if N is sufficiently large, then the underlying lattice problem is too difficult to solve using current techniques.

Outline of the PASS Authentication and Signature Scheme

Public Parameters. All users agree on a prime number q and a set of distinct numbers $S = \{\alpha_1, \ldots, \alpha_{N/2}\}$ modulo q, and they let $N = q - 1$. All polynomials are of degree $N - 1$, polynomial multiplication uses the convolution

rule (2) given above, and all computations (except for verification step A) are performed modulo q. Appropriate quantities A_h, B_h are also chosen to be used in the verification process.

Key Creation. The Prover selects a binary polynomial $f(X)$. This polynomial is her private key. She publishes the set of values $f(S) = \{f(\alpha) : \alpha \in S\}$. This set of values is her public key.

Commitment. In the commitment step, the Prover selects another binary polynomial $g_1(X) \in R_q$. She computes and sends to the Verifier the set of values $g_1(S)$.

Challenge. In the challenge step, the Verifier selects two extremely small polynomials $c_1(X)$ and $c_2(X)$ (say with between two and eight nonzero coefficients) and sends them to the Prover. For security reasons, it is also important that $c_1(X)$ have no nonzero roots modulo q for values of X not in S. (There is at least a 50% chance that this will be true for a randomly chosen c_1.)

Response. In the response step, the Prover selects a third binary polynomial $g_2(X)$. She computes and sends to the Verifier the polynomial

$$h(X) = \big(f(X) + c_1(X)g_1(X) + c_2(X)g_2(X)\big)g_2(X). \tag{5}$$

Verification. The Verifier performs the following two steps to verify the Prover's identity:

(A) The Verifier checks that the polynomial $h(X)$ is moderately small by writing it as $h(X) = \sum a_i X^i$ and verifying the bound

$$\sum_{i=0}^{N-1} (a_i - A_h)^2 < B_h,$$

where A_h and B_h are public quantities. Note that this computation is not done modulo q, but is simply a sum of integers.

(B) For each $\alpha \in S$, the Verifier computes the quantity

$$\big(f(\alpha) + c_1(\alpha)g_1(\alpha)\big)^2 + 4c_2(\alpha)h(\alpha) \pmod q \tag{6}$$

and checks that it is a square modulo q.

If the polynomial h passes tests (A) and (B), then the Verifier accepts the Prover's identity.

Why It Works. First we note that the definition (5) says that the polynomial $h(X)$ is a simple combination of the polynomials f, g_1, g_2, c_1, c_2, all of which are binary, so it is clear that the coefficients of $h(X)$ will also be moderately small. It is a simple matter to take A_h to be the average expected value of the coefficients of h and to experimentally find a bound B_h so that if h has the correct form (5), then it will almost certainly pass verification step A. Next we observe that the verifier is able to compute the quantity (6), since he knows the polynomials $h(X), c_1(X), c_2(X)$ and he knows the values

of $f(X)$ and $g_1(X)$ for all $\alpha \in S$. To see why (6) is a square, we use the definition (5) of $h(X)$ to compute

$$\begin{aligned}(f+c_1g_1)^2+4c_2h &= (f+c_1g_1)^2+4c_2\big((f+c_1g_1+c_2g_2)g_2\big)\\ &= f^2+2c_1g_1+c_1^2g_1^2+4c_2g_2f+4c_1c_2g_1g_2+4c_2^2g_2^2\\ &= (f+c_1g_1+2c_2g_2)^2. \end{aligned} \tag{7}$$

Thus $\big(f(X)+c_1(X)g_1(X)\big)^2+4c_2(X)h(X)$ is actually the square of a polynomial, so it certainly gives a square modulo q when evaluated at any α.

Remark 2.

- The best way to create the challenge polynomials c_1 and c_2 is for the Verifier to choose a random string (of 80 to 160 bits) and send it to the Prover. They then both apply a common hash function to the random string in order to create c_1 and c_2. This has the advantage of cutting down the number of bits transmitted, as well as making it more difficult for the Verifier to mount any sort of attack based on choosing c_1 and c_2 to have a particular form.
- We have presented PASS as an authentication scheme, but any authentication scheme that includes a challenge step can be combined with a hash function to create a signature scheme. Thus if a digital document D is to be signed, D and the set of values $g_1(S)$ are sent through a standard hash function to obtain a small bit string (say 80 to 160 bits), and this bit string is used to create the challenge polynomials c_1, c_2. The Signer then publishes the values $g_1(S)$, $h(X)$, and D. (We assume, of course, that the Signer's public key $f(S)$ is already in the public domain.) Anyone wishing to verify the signature can use D, $g_1(S)$, and the hash function to recreate c_1 and c_2, and then he has enough information to perform the verification step described above.
- The computation of the sets of values $f(S)$, $g_1(S)$, and $h(S)$ required during the PASS process can be performed extremely rapidly. Even a direct computation takes only $O(N^2)$ steps, but it is even more efficient to use FFT, especially if N is highly divisible by 2 (as are the suggested values $N = 768$, $N = 928$, and $N = 1052$), in which case the computation is reduced to $O(N \log N)$ steps. Note that the field $\mathbb{F}_q$ contains a primitive N^{th}-root of unity, so in this setting one can compute Fast Fourier Transforms using only integer arithmetic; there is no need to use complex numbers or floating point numbers.
- In verification step (B), the Verifier needs to check if certain quantities are squares. This can be done rapidly using either quadratic reciprocity or the powering map, but if a little extra storage is available, it's even quicker to precompute a table of squares.

2 MiniPASS

Our goal is to fit PASS into a minimal amount of space while still retaining desirable operating characteristics. We begin by analyzing each step of the PASS algorithm to see how much computation needs to be done. In particular, we will assume that a smart card with constrained operating resources is communicating with a server that has access to a more robust operating environment. Thus we will analyze PASS twice, first with the smart card as the Prover, and second with the smart card as the Verifier. In both cases we will shift as much of the computation as possible onto the server. We will assume that the smart card already includes a (pseudo)random number generator and a hash function.

2.1 The Smart Card as Prover

We will assume that the smart card's public key $f(S)$ is publicly available, and that her private key $f(X)$ is stored in ROM. Note that since the private key $f(X)$ is a binary polynomial, it requires only N bits to store.

The SmartCard/Prover begins with the Commitment step. She chooses a random binary polynomial $g(X)$. She needs to compute and send to the server the values $g(\alpha)$ for every $\alpha \in S$. If $g(X) = \sum b_i X^i$, she can compute these values one at a time via the formula

$$g(\alpha) = (\cdots((a_{N-1} * \alpha + a_{N-2}) * \alpha + a_{N-3}) * \alpha + \cdots + a_1) * \alpha + a_0. \qquad (8)$$

This method requires N multiplications modulo q and N additions. There are much faster ways to compute the $g(\alpha)$ values (see remark 6 below), but they require somewhat more storage, which is what we are trying to minimize. Note that the SmartCard/Prover does not need to store the values, so she can simply compute one $g(\alpha)$, send that value to the Server/Verifier, and then go on to the next value of α.

The Server/Verifier then selects challenge polynomials c_1 and c_2, or more likely, selects a bit string that is hashed to form c_1 and c_2. See Remark 4 below for further details on the selection of c_1 and c_2.

In the response step, the SmartCard/Prover is supposed to select another binary polynomial $g_2(X)$ and send the quantity

$$h = (f + c_1 g_1 + c_2 g_2) g_2 = f g_2 + c_1 g_1 g_2 + c_2 g_2^2$$

to the Server/Verifier. However, it is probably more efficient for the SmartCard/Prover to compute and transmit all of the values of this polynomial h, and then the Server/Verifier can reconstruct h itself using the inversion formula (4). As in the commitment step, the SmartCard/Prover only needs to compute one value of $h(\alpha)$ at a time. Further, the challenge polynomials c_1 and c_2 are extremely sparse, so evaluating them can be done very rapidly. Thus the time consuming part of the response step is computation of the values $f(\alpha)$, $g_1(\alpha)$, $g_2(\alpha)$ for all $0 \le \alpha < q$, but even this is not a tremendously onerous task.

The SmartCard/Prover has now fulfilled her tasks, and it remains for the Server/Verifier to perform the final verification step.

2.2 The Smart Card as Verifier

We will assume that the SmartCard/Verifier contains the Server/Prover's public key $f(S)$ in ROM. In principle, this requires $\frac{N}{2}\log_2(q)$ bits; but in practice for (say) $q = 769$, each of the $N/2$ numbers modulo q would be stored in 16 bits, so the public key requires N bytes of storage.

The first thing that the SmartCard/Verifier does is receive from the Server/Prover the set of values $g_1(\alpha)$ for $\alpha \in S$, and it appears that the SmartCard/Verifier needs to store all of those values for later use. This seems necessary because in the final verification step, the SmartCard/Verifier is asked to check that the quantity

$$\big(f(\alpha) + c_1(\alpha)g_1(\alpha)\big)^2 + 4c_2(\alpha)h(\alpha) \pmod q \tag{9}$$

is a square modulo q for every $\alpha \in S$. However, suppose that instead the SmartCard/Verifier only checks the condition (9) for a random selection of $\alpha \in S$. More precisely, suppose that she checks (say) 60 values of α and that all of them pass the test (9). The probability of this happening at random is 2^{-60}, so she can be fairly confident that in fact every $\alpha \in S$ will pass the test (9). This simple observation will greatly increase operating speed while decreasing memory requirements.

This means that prior to the commitment step, the SmartCard/Verifier randomly selects a set of 60 numbers

$$T = \{\alpha_1, \alpha_2, \ldots, \alpha_{60}\}$$

in S. (For added efficiency, but slightly reduced security, she could instead select 40 numbers.) Then, during the commitment step, the SmartCard/Verifier will receive from the Server/Prover the values $g_1(\alpha)$ for every $\alpha \in S$, but she will only store the 60 values of $g_1(\alpha)$ for $\alpha \in T$.

Remark 3. The choice of the number of values to check clearly has an impact an security. One of the nice features of PASS is that the Verifier can choose what level of security she feels is appropriate. For most applications it will probably be acceptable that an imposter has a 1 in 2^{60} chance of success; indeed, even 1 in 2^{40} is likely to be enough. In general, if the SmartCard/Verifier checks t randomly chosen values in S, then the probability of fraud is 1 in 2^t, while the amount of computation is proportional to t. This probabilistic component of PASS is one of its attractive features, since it lets the Verifier balance exponential security against linear computational load.

For the challenge step, the SmartCard/Verifier creates the polynomials $c_1(X)$ and $c_2(X)$ and sends them to the Server/Prover. More precisely, she chooses a random string, and c_1 and c_2 are created using a hash function, see Remark 4 below for details.

The Server/Prover's response is to create a certain polynomial $h(X)$ and send $h(X)$ to the SmartCard/Verifier. If $h(X)$ is the polynomial

$$h(X) = a_0 + a_1X + a_2X^2 + \cdots + a_{N-1}X^{N-1},$$

we will require the Server/Prover to transmit $h(X)$ to the SmartCard/Verifier as the list of coefficients $a_{N-1}, a_{N-2}, \ldots, a_1, a_0$. As the SmartCard/Verifier receives each coefficient, she does two things. First, she keeps a running total of the quantities

$$(a_i - A_h)^2, \quad i = 0, 1, 2, \ldots, N-1. \tag{10}$$

Note that these numbers are not reduced modulo q, so the sum should be stored as a 32 bit number. Second, she computes the values of $h(\alpha)$ modulo q one coefficient at a time, but only for the 60 values of α in T. Note that she does not need to store the coefficients of h, so the only storage requirements are the running total (10) and the 60 values of h, for a total of $4 + 60 \cdot 2 = 124$ bytes.

After receiving and storing this information, it remains to complete the verification process. If the running total (10) is larger than B_h, then the Server/Prover's identity is rejected. Otherwise, the SmartCard/Verifier computes the quantity

$$\big(f(\alpha) + c_1(\alpha)g_1(\alpha)\big)^2 + 4h(\alpha)c_2(\alpha) \tag{11}$$

for each $\alpha \in T$ and checks if it is a square modulo q. Note that the SmartCard/Verifier stored the values of $g_1(\alpha)$ for $\alpha \in T$ during the commitment step, she stored the values of $h(\alpha)$ for $\alpha \in T$ during the response step, and she knows f, c_1, c_2, so she can compute their values for any α. It is thus easy (and fast) for her to compute the 60 values of (11) for the numbers $\alpha \in T$.

There remains the question of how she verifies that they are squares modulo q. The fastest method is to store a table of values, or more efficiently, store a bit string of length $q-1$ so that the i^{th} bit equals 1 if i is a square modulo q. For $q = 769$, this requires an additional 96 bytes of ROM. An alternative method for checking if a number n is a square modulo q is to compute

$$n^{(q-1)/2} \bmod q.$$

This value will be 1 if n is a square, and -1 if it is not a square. This powering operation is fast, and will probably already be used for computing the values of the sparse polynomials $c_1(X)$ and $c_2(X)$, so it will not require additional routines. (It is, of course, also possible to use quadratic reciprocity for this step.)

We stress again that since the SmartCard/Verifier is only checking 60 values, the time required to perform a verification is extremely small. Based on the experiments described in Section 3, the smart card does verifications 25 to 30 times faster than proving identity.

2.3 Additional Implementation Considerations

We briefly mention additional items to consider during implementation.

Remark 4. In order to avoid possible attacks, the challenge polynomials c_1 and c_2 should be generated as follows. The Verifier selects a random 80 bit string B. A

hash function is evaluated at B, and the result is fed into a simple function that generates two very sparse binary polynomials $c_1(X)$ and $c_2(X)$. For $N = 768$, it suffices to have a total of eight nonzero coefficients, and for ease of implementation, we will assume that $c_1(X)$ has two nonzero coefficients and that $c_2(X)$ has six nonzero coefficients. Thus $c_1(X)$ looks like

$$c_1(X) = X^{n_1} + X^{n_2}, \tag{12}$$

and $c_2(X)$ looks similar, but with six terms. Note that c_1 and c_2 should be stored as a list of exponents (e.g., $c_1 = (n_1, n_2)$), so they require only 16 bytes of storage.

For security reasons described in [6], it is important that $c_1(X)$ have no nonzero roots α modulo q with $\alpha \notin S$. An easy way to guarantee this is to take c_1 as above (12) with the condition that the exponents satisfy $\gcd(N, n_1 - n_2) = 1$. Further, for $N = 768$, this gcd condition is equivalent to

$$n_1 - n_2 \equiv \pm 1 \pmod 6, \tag{13}$$

so it isn't even necessary to compute a gcd.

Thus a simple protocol for choosing c_1 is to use the hash function as above to produce a possible candidate (12) for c_1. If it satisfies the security condition (13), stop, otherwise increment n_1 until condition (13) is satisfied. This will take at most 3 iterations.

Remark 5. In order to save space, binary polynomials should be stored as 1 bit per coefficient. Evaluation of binary polynomials via the formula (8) is then moderately inefficient, since individual bits need to be pulled out one at a time. One way to speed up this process is to precompute a small table of (say) 16 values. Thus to compute $f(\alpha)$, first make a table of values:

Bits	Value	Bits	Value	Bits	Value	Bits	Value
0000	0	0100	α^2	1000	α^3	1100	$\alpha^3 + \alpha^2$
0001	1	0101	$\alpha^2 + 1$	1001	$\alpha^3 + 1$	1101	$\alpha^3 + \alpha^2 + \alpha$
0010	α	0110	$\alpha^2 + \alpha$	1010	$\alpha^3 + \alpha$	1110	$\alpha^3 + \alpha^2 + \alpha$
0011	$\alpha + 1$	0111	$\alpha^2 + \alpha + 1$	1011	$\alpha^3 + \alpha + 1$	1111	$\alpha^3 + \alpha^2 + \alpha + 1$

Then read the coefficient bits of $f(X)$ off four bits at a time and use the table to compute a partial value. We illustrate with a polynomial of low degree. If $f(X)$ is the polynomial

$$f(X) = x^{15} + x^{14} + x^{12} + x^{11} + x^9 + x^8 + x^7 + x^5 + x^4 + x^2 + 1,$$

then we can evaluate $f(\alpha)$ as

$$(([\alpha^3 + \alpha^2 + 1] * \alpha^4 + [\alpha^3 + \alpha + 1]) * \alpha^4 + [\alpha^3 + \alpha + 1]) * \alpha^4 + [\alpha^2 + 1].$$

The quantities in square brackets (and the precomputed value of α^4) can be read from the table, significantly increasing efficiency. Since the table only takes 32 bytes, the space is negligible. If additional space is available, one could use 512 bytes to make a table of 256 values; but beyond that size it would make more sense to use Fast Fourier Transforms, which give an even greater speedup.

Remark 6. As indicated in the previous remark, there are various ways to compute the values $f(\alpha)$ of a polynomials that trade space for time. In our situation, since N is divisible by a large power of 2 and since a generator modulo q is an N^{th} root of unity, the fastest way to compute the full set of values $\{f(\alpha) : 1 \leq \alpha < q\}$ is using Fast Fourier Transforms (FFT). This is probably not a good method for use by the smart card, since it requires more storage; but the server will certainly want to use FFT. An FFT polynomial evaluation routine for use by PASS easily fits into 10K (of which only about 4K need be RAM), and at the cost of some efficiency, can be made to fit into 5 or 6K.

3 Sample Implementation of MiniPASS

In this section we describe the results of implementing MiniPASS on a desktop computer using C. We implemented the routines to be used by the smart card. We did not implement the server routines, which would be considerably faster, but would also require more memory.

For ease of implementation, we used the standard C utility function `rand()` to generate random numbers. This is not cryptographically secure. In practice the smart card would probably have its own (pseudo)random number generator and hash function.

Table 1 gives the operating characteristics of our implementation of MiniPASS. We make the following remarks concerning the information in Table 1.

Remark 7.

- ROM includes storage for the smart card private key (96 bytes) and for the server public key (768 bytes).
- The smart card never simultaneously acts as Prover and Verifier, so it suffices to have 564 bytes of RAM. At the cost of only checking 40 values (with somewhat reduced security), this may be reduced to 404 bytes.
- The MacOS figures were obtained on a Macintosh G3 300 MHz running MacOS 8.5 and compiled with Metroworks Codewarrior. The Linux figures were obtained on a Celeron 400 MHz running RedHat Linux 6.0 and compiled with egcs.
- We used a table of length 16 as described in Remark 5 to speed evaluation of polynomials. The requisite 32 bytes of RAM is included in the table.

The timing estimates in Table 1 are for computations only. They do not include time for communication between the smart card and the server. The amount of data that needs to be exchanged is listed in Table 2. (We have listed the Challenge as the 20 bytes needed to send the actual challenge polynomials c_1 and c_2, but in practice the challenge would consist of 80 bits that is hashed to produce the challenge polynomials.)

Table 1. MiniPASS Operating Characteristics

	Card/Prover	Card/Verifier
RAM	350 bytes	564 bytes
ROM (MacOS)	3076 bytes	
ROM (Linux)	3088 bytes	
Time (MacOS)	60.5 ms	2.6 ms
Time (Linux)	71.3 ms	2.3 ms

Table 2. MiniPASS Communication Requirements

Commitment	768 bytes
Challenge	20 bytes
Response	1536 bytes
Total	2324 bytes

Remark 8.

- It would be relatively easy to pack the transmitted material more efficiently and save approximately 37%. This is because the smart card and server are exchanging lists of numbers, with each number lying between 0 and 768. For simplicity, we have assumed that these numbers are stored and transmitted as 16 bit numbers, but they will actually each fit into 10 bits.
- Further savings of both speed and bytes transmitted may be achieved by other PASS-type authentication/signature schemes, i.e., by schemes that depend for their security on the difficulty of reconstructing a small polynomial from a partial set of its values. These PASS-type schemes are similar to the schemes described in [5] and [6] (and in this paper), but use polynomial combinations different from the quantities

$$c_1fg + c_2fg' + c_3f'g + c_4f'g' \quad \text{and} \quad (f + c_1g_1 + c_2g_2)g_2$$

used in [5] and [6], respectively. However, since these new schemes are still undergoing security analyses, we have opted to feature the PASS scheme from [6] in this paper.

References

1. M. Ajtai, C. Dwork, *A public-key cryptosystem with worst case/average case equivalence*, in Proc. 29th ACM Symposium on Theory of Computing, 1997, 284–293.
2. E.F. Brickell and K.S. McCurley. *Interactive Identification and Digital Signatures*, AT&T Technical Journal, November/December, 1991, 73–86.
3. L.C. Guillou and J.-J. Quisquater. *A practical zero-knowledge protocol fitted to security microprocessor minimizing both transmission and memory*, Advances in Cryptology—Eurocrypt '88, Lecture Notes in Computer Science 330 (C.G. Günther, ed.), Springer-Verlag, 1988, 123–128.

4. J. Hoffstein, J. Pipher, J.H. Silverman, *NTRU: A new high speed public key cryptosystem*, in Algorithmic Number Theory (ANTS III), Portland, OR, June 1998, Lecture Notes in Computer Science 1423 (J.P. Buhler, ed.), Springer-Verlag, Berlin, 1998, 267–288.
5. J. Hoffstein, D. Lieman, J.H. Silverman, *Polynomial Rings and Efficient Public Key Authentication*, in Proceeding of the International Workshop on Cryptographic Techniques and E-Commerce (CrypTEC '99), Hong Kong, (M. Blum and C.H. Lee, eds.), City University of Hong Kong Press.
6. J. Hoffstein, J.H. Silverman, *Polynomial Rings and Efficient Public Key Authentication II*, CCNT '99 Proceedings, to appear.
7. O. Goldreich, S. Goldwasser, S. Halevi, *Public-key cryptography from lattice reduction problems*, in Proceedings of CRYPTO 97, Lect. Notes in Comp. Sci. 1294, 1997, 112–131.
8. A.J. Menezes and P.C. van Oorschot and S.A. Vanstone. *Handbook of Applied Cryptography*, CRC Press, 1996.
9. T. Okamoto. *Provably secure and practical identification schemes and corresponding signature schemes*, Advances in Cryptology—Crypto '92, Lecture Notes in Computer Science 740 (E.F. Brickell, ed.) Springer-Verlag, 1993, 31–53.
10. Rosing, M.: Implementing Elliptic Curve Cryptography. Manning Publications Greenwich, CT (1999).
11. C.-P. Schnorr. Efficient identification and signatures for smart cards, Advances in Cryptology—Crypto '89, Lecture Notes in Computer Science 435 (G. Brassard, ed), Springer-Verlag, 1990, 239–251.
12. J.H. Silverman, *Lattices, Cryptography, and the NTRU Cryptosystem*, Proceedings of a DIMACS Conference, January 10–14, 2000, American Mathematical Society, to appear.
13. J. Stern. A new identification scheme based on syndrome decoding, Advances in Cryptology—Crypto '93, Lecture Notes in Computer Science 773 (D. Stinson, ed.), Springer-Verlag, 1994, 13–21.
14. J. Stern. Designing identification schemes with keys of short size, Advances in Cryptology—Crypto '94, Lecture Notes in Computer Science 839 (Y.G. Desmedt, ed), Springer-Verlag,1994, 164–173.
15. J. Stern, *Lattices and Cryptography: An Overview* in Public Key Cryptography (PKC '98), Lecture Notes in Computer Science 1431, Springer-Verlag, 1998, 50–54.
16. D. Stinson, *Cryptography: Theory and Practice.* CRC Press, 1997.

Efficient Generation of Prime Numbers*

Marc Joye[1], Pascal Paillier[1], and Serge Vaudenay[2]

[1] Gemplus Card International, France
{Marc.Joye,Pascal.Paillier}@gemplus.com
[2] École Polytechnique Fédérale de Lausanne, Switzerland
Serge.Vaudenay@epfl.ch

Abstract. The generation of prime numbers underlies the use of most public-key schemes, essentially as a major primitive needed for the creation of key pairs or as a computation stage appearing during various cryptographic setups. Surprisingly, despite decades of intense mathematical studies on primality testing and an observed progressive intensification of cryptographic usages, prime number generation algorithms remain scarcely investigated and most real-life implementations are of rather poor performance. Common generators typically output a n-bit prime in heuristic average complexity $O(n^4)$ or $O(n^4/\log n)$ and these figures, according to experience, seem impossible to improve significantly: this paper rather shows a simple way to substantially reduce the value of hidden constants to provide much more efficient prime generation algorithms. We apply our techniques to various contexts (DSA primes, safe primes, ANSI X9.31-compliant primes, strong primes, etc.) and show how to build fast implementations on appropriately equipped smart-cards, thus allowing on-board key generation.
Keywords: Prime number generation, key generation, RSA, DSA, fast implementations, crypto-processors, smart-cards.

1 Introduction

Traditional prime number generation algorithms asymptotically require $O(n^4)$ or $O(n^4/\log n)$ bit operations where n is the bit-length of the expected prime number. This complexity may even become of the order of $O(n^5/(\log n)^2)$ in the case of *constrained* primes, such as safe or quasi-safe primes for instance. These asymptotic behaviors,[1] according to experience, seem impossible to improve significantly. In this paper, we rather propose simple algebraic methods which substantially reduce the value of the hidden constants, thus providing much more efficient prime generation algorithms.

We apply our techniques to various contexts such as DSA primes [9], strong primes [14] and ANSI X9.31-compliant primes [1], that is, real-life scenarios of

* Some parts presented in this paper are patent pending.

[1] assuming that multiplications modulo q are in $O(|q|^2)$. Theoretically, one could decrease this complexity by using multiplication algorithms such as Karatsuba in $O((|q|^{\log_2 3})$ or Schönhage-Strassen in $O(|q| \log |q| \log\log |q|)$.

Ç.K. Koç and C. Paar (Eds.): CHES 2000, LNCS 1965, pp. 340–354, 2000.

well-recognized utility. As an illustration, we also reduce the number of rounds of Boneh and Franklin's [3] shared RSA keys protocol by a factor of nearly 10.

Finally, our techniques allow fast implementations on cryptographic smart-cards for on-board RSA [15] (or other schemes) key generation. Our motivation here is to help transferring this task from terminals to smart-cards themselves in the near future for more confidence, security, and compliance with network-scaled distributed protocols that include smart-cards, such as electronic cash or mobile commerce.

Notations. Throughout this paper, the following notations are used. Other notations will be introduced when needed.

Symbol	Signification
$\#\mathcal{A}$	cardinal of a set $\mathcal{A}$
$\lvert x\rvert$	bit-length of number x
$\{0,1\}^t$	set of t-bit numbers
$\mathbb{Z}_{\Pi}$	ring of integers modulo Π
$\mathbb{Z}_{\Pi}^*$	multiplicative group of $\mathbb{Z}_{\Pi}$
$\phi(\cdot)$	Euler's totient function
$\lambda(\cdot)$	Carmichael's function
$\mathrm{O}(\cdot)$	asymptotic bound
$\Omega(\cdot)$	asymptotic equivalence
$a \approx b$	a is approximatively equal to b
$a \gtrsim b$	$a \approx b$ and $a \geq b$
$a \lesssim b$	$a \approx b$ and $a \leq b$

2 Primality and Compositeness Tests

A lot of studies on primality testing have been carried out for years, and can be found in the literature devoted to the subject (e.g., see [7]). Computationally, we may distinguish *true primes* and *probable primes*: the difference being the way these are generated. A probable prime is usually obtained through a *compositeness test*. Such a test declares that a number is composite with probability 1 or prime with some probability < 1. Hence repeatedly running the test gives more and more confidence in the generated (probable) prime. Typical examples of compositeness tests include Fermat test, Solovay-Strassen test [16], and Miller-Rabin test [10, p. 379].

There also exist (true) *primality tests*, which declare a number prime with probability 1 (e.g., Pocklington's test [12] and its elliptic curve analogue [2], the Jacobi sum test [4]). However, these tests are generally more expensive or intricate.

To motivate further analysis, we hereafter assume that we are given some compositeness test T provided as a primality oracle of complexity $\tau(n) = \mathrm{O}(n^3)$ and of negligible error probability. Designing an efficient prime generation algorithm then reduces to the problem of knowing how to use T in order to produce a n-bit prime with a minimal number of calls to the oracle.

3 Generating Primes: Prior Art

3.1 Naive Generators

We refer to the naive prime number generator as the following:

1. pick a random n-bit odd number q
2. if $\mathsf{T}(q) = \texttt{false}$ then goto 1
3. output q

Fig. 1. Naive Prime Number Generator.

Neglecting calls to the random number generator, the expected number of trials here is asymptotically equal to $(\ln 2^n)/2 \approx 0.347\, n$. Generating a 256-bit prime thus requires 89 trials in average.

The previous algorithm has an incremental variant, which is given below on Fig. 2.

1. pick a random n-bit odd number q
2. while $\mathsf{T}(q) = \texttt{false}$ do $q \leftarrow q + 2$
3. output q

Fig. 2. Naive Incremental Prime Number Generator.

It should be outlined that this second algorithm has not the same proven complexity [5]. A proper analysis actually has to exploit the properties of the distribution of prime numbers, in connection with Riemann's Hypothesis. The incremental generator is however commonly used and we recall that it was shown to fail with probability $\mathrm{O}(t^3\, 2^{-\sqrt{t}})$ after $\Omega(t)$ trials (see [11, p. 148]).

3.2 Classical Generation Algorithms

The naive incremental generator can be made more efficient by choosing the initial candidate q already co-prime to small primes. Usually, one defines $\Pi = 2 \cdot 3 \cdots 29$ and randomly chooses a n-bit number q satisfying $\gcd(q, \Pi) = 1$. If $\mathsf{T}(q) = \texttt{false}$ then q is updated as $q \leftarrow q + \Pi$ (note that the naive generator corresponds to the special case $\Pi = 2$). If Π is a constant independent from n and contains k distinct primes, we denote this probabilistic algorithm by $\mathsf{H}[n, k]$.

The next section presents two new algorithms. The first one, making use of look-up tables, produces random numbers *constructively* co-prime to small

primes. The second algorithm, slightly slower, is space-optimized and particularly suited for smart-card implementations. Based on this, we construct a new prime generation algorithm in Section 5 and give timing results in Section 6. Finally, in Section 7, we apply these new techniques to particular contexts.

4 Generating Invertible Numbers modulo a Product of Primes

Common prime number generators generally include a stage of trial divisions by small primes. We investigate in this section a way of avoiding this stage by efficiently constructing candidates that already satisfy co-primality properties. We base our constructions on simple algebraic techniques.

4.1 A Table-Based Method

Let $\Pi = \prod_{i=1}^{k} {p_i}^{\delta_i}$ be a n-bit product of the first k primes with some small exponents δ_i. Let $\Delta = \max_i \delta_i$. We denote by $x = (x_1, \ldots, x_k)_{\equiv}$ the modular representation of $x \in \mathbb{Z}_\Pi$, i.e., $x_i = x \bmod p_i^{\delta_i}$. For $i = 1, \ldots, k$, one then defines $\theta_i = (0, \ldots, 1, \ldots, 0)_{\equiv}$ where the "1" stands in i^{th} position. It is obvious to see that we always have

$$\forall x \in \mathbb{Z}_\Pi \qquad x = \sum_{i=1}^{k} x_i \, \theta_i \bmod \Pi \ ,$$

that is, the function $x \mapsto (x_i)$ is a bijection[2] from $\mathbb{Z}_{{p_1}^{\delta_1}} \times \cdots \times \mathbb{Z}_{{p_k}^{\delta_k}}$ into $\mathbb{Z}_\Pi$. This function also defines a bijection from $\mathbb{Z}^*_{{p_1}^{\delta_1}} \times \cdots \times \mathbb{Z}^*_{{p_k}^{\delta_k}}$ to $\mathbb{Z}^*_\Pi$, and it follows that

$$\begin{aligned} \forall x \in \mathbb{Z}_\Pi \qquad x \in \mathbb{Z}^*_\Pi &\iff x_i \in \mathbb{Z}^*_{{p_i}^{\delta_i}} \\ &\iff x_i^{\delta_i} \not\equiv 0 \pmod{p_i^{\delta_i}} \\ &\iff x_i^{\delta_i} \theta_i \not\equiv 0 \pmod{\Pi} \quad \text{for } i = 1, \ldots, k \ . \end{aligned} \tag{1}$$

As a consequence, it appears that $x \in \mathbb{Z}^*_\Pi$ can be built-up from numbers x_i as long as they verify Eq. (1) above. We then define $\mathcal{A}$ as a set of random sequences $\alpha = (\alpha_1, \alpha_2, \ldots)$ with $\alpha_i \in \{0,1\}^t$. Equation (1) gives a natural way of surjectively transforming any $\alpha \in \mathcal{A}$ into an invertible number $\mathtt{g}(\alpha) \in \mathbb{Z}^*_\Pi$. The corresponding algorithm, $\mathtt{g}$, is depicted on Fig. 3.

Since we use t-bit numbers α_i and reduce them (implicitly) modulo $p_i^{\delta_i}$, there exists a bias (lying around 2^{-t}) leading to a non-uniform output distribution.[3] But this underlying bias may easily be made arbitrarily small by increasing t (which negatively affects the average complexity as well). We therefore suggest

[2] This is the usual Chinese Remainder Theorem correspondence [8].

[3] It nevertheless seems a hard task to exploit it in some way for *a posteriori* secret key retrieval.

Precomputations: $\Pi = \prod_{i=1}^{k} p_i^{\delta_i}$, $t = C \cdot \max_i |p_i^{\delta_i}|$ $(C = 2)$, $\{\theta_i\}$
Input: a random sequence α
Output: an invertible number c modulo Π

1. $c = 0$
2. for $i = 1$ to k
   - 2.1 pick a random t-bit number α_i from α
   - 2.2 if $\alpha_i^{\delta_i}\theta_i \bmod \Pi = 0$ goto 2.1
   - 2.3 $c \leftarrow c + \alpha_i\,\theta_i \bmod \Pi$
3. output c

Fig. 3. Generator g of Invertible Numbers modulo Π.

$t = C \cdot \max_i |p_i^{\delta_i}|$ as a good compromise, where the ratio C may be fixed to 2 for practical implementations. Further, we claim (for a negligible bias) that the function $\mathsf{g} : \mathcal{A} \to \mathbb{Z}_\Pi^*$ verifies

(*i*) to be surjective;
(*ii*) that for each element x of $\mathbb{Z}_\Pi^*$, the number of x's pre-images is about $\#\mathcal{A}/\#\mathbb{Z}_\Pi^* = \#\mathcal{A}/\,\phi(\Pi)$, and this guarantees the uniformity of g's outputs from its inputs;
(*iii*) g has a low (time) complexity $\gamma(n) = \mathrm{O}(n^2)$.

4.2 Modular Search Method

As aforementioned, the algorithm g generates uniformly distributed elements of $\mathbb{Z}_\Pi^*$. Although the execution time of g happens to be excellent when using an arithmetic processor, the memory space needed to store the numbers $\{\theta_i\}$ may appear dissuasive, in particular on a smart-card where memory may be subject to strong size constraints. We propose here a simple alternative method based on Carmichael's theorem[4]

$$\forall c \in \mathbb{Z}_\Pi^* \qquad c^{\lambda(\Pi)} \equiv 1 \pmod{\Pi}\,,$$

or more exactly on its converse:

Proposition 1. *$\forall c \in \mathbb{Z}_\Pi$, if $c^{\lambda(\Pi)} \equiv 1 \pmod{\Pi}$ then $c \in \mathbb{Z}_\Pi^*$.*

Proof. A number $0 \le c < \Pi$ is in $\mathbb{Z}_\Pi^*$ if and only if, for all primes p dividing Π, we have $\gcd(c, p) = 1 \iff c^{p-1} \equiv 1 \pmod{p}$ so that $c^{\lambda(\Pi)} \equiv 1 \pmod{\Pi}$ by Chinese remaindering. □

This provides an easy co-primality test that requires a single modular exponentiation with exponent $\lambda(\Pi)$. Note that this technique only needs the storage of Π and $\lambda(\Pi)$, and is also particularly suitable for crypto-processors. In addition, since Π is smooth, $\lambda(\Pi)$ is optimally small. The obtained procedure is depicted below.

[4] If $\Pi = \prod_{i=1}^{k} p_i^{\delta_i}$ then $\lambda(\Pi) = \mathrm{lcm}[\lambda(p_i^{\delta_i})]_{i=1,\ldots,k}$, and $\lambda(p_i^{\delta_i}) = \phi(p_i^{\delta_i}) = p_i^{\delta_i - 1}(p_i - 1)$ for an odd prime p_i, $\lambda(2) = 1$, $\lambda(4) = 2$ and $\lambda(2^{\delta_i}) = \frac{1}{2}\,\phi(2^{\delta_i}) = 2^{\delta_i - 2}$ for $\delta_i \ge 3$.

Precomputations: $\Pi = \prod_{i=1}^{k} p_i^{\delta_i}$ and $\lambda(\Pi)$
Output: an invertible number c modulo Π
1. pick an n-bit random number $c < \Pi$
2. while $c^{\lambda(\Pi)} \bmod \Pi \neq 1$ do $c \leftarrow c + 1$
3. output c

Fig. 4. Generator g' of Invertible Numbers modulo Π.

The previous algorithm can be improved via Chinese remaindering. Instead of testing the co-primality of c to Π, one checks the co-primality to some factor of Π, say π_1. If $\gcd(c, \pi_1) \neq 1$ (i.e., if $c^{\lambda(\pi_1)} \bmod \pi_1 \neq 1$) then we already know that $\gcd(c, \Pi) \neq 1$. Otherwise, we test the co-primality of c with another factor π_2 of Π, and so on for several factors π_i until $\prod_i \pi_i = \Pi$ (or is a multiple of Π).

Although the complexity of g' may appear greater than $\gamma(n)$, the comparison must take into account the computational features of the underlying crypto-processor (see Section 6). Of course, the implementer shall choose between generators g and g' (or a variant) according to the necessity of saving time (using g) or space (using g'). We consider in the following that this choice has been done once for all, and that a black-box generator (hereafter referred to as g) of elements of $\mathbb{Z}_{\Pi}^{*}$ is at disposal: we now have to deal with how to design a prime generation algorithm in which primitives T and g get optimally exploited.

5 An Efficient Prime Generation Algorithm

Generated primes are expected to lie in some target window $\mathcal{F} = [w_{\min}, w_{\max}]$, where $w_{\max} = 2^n - 1$ in most contexts, and $w_{\min}$ is equal to $2^{n-1} + 1$ when generating n-bit primes, or $\lceil \sqrt{2^{2n-1} + 1} \rceil$ if the context imposes to obtain a strict $2n$-bit number when multiplying two so-generated primes (RSA moduli for instance).

The basic idea consists in utilizing g to produce a sequence of candidates that will be tested one by one until a prime is found. We now describe how we choose parameter Π. First, we find an integer η containing a maximum number of (different) primes (or more precisely minimizing the ratio $\phi(\eta)/\eta$) and such that there exist small integers $\varepsilon_{\min}$ and $\varepsilon_{\max}$ satisfying

$$\varepsilon_{\min}\eta \gtrapprox w_{\min} \quad \text{and} \quad \varepsilon_{\max}\eta \lessapprox w_{\max} - w_{\min} \ .$$

We then set

$$\Pi = \varepsilon_{\max}\,\eta \quad \text{and} \quad \rho = \varepsilon_{\min}\,\eta \ . \tag{2}$$

Once an invertible element $c^{(1)} \in \mathbb{Z}_{\Pi}^{*}$ is generated (using g), the first prime candidate is defined as

$$q^{(1)} = c^{(1)} + \rho \ .$$

Output: a n-bit prime q
1. $c = \mathsf{g}()$
2. $q = c + \rho$
3. if $\mathsf{T}(q) = \texttt{false}$ then $c \leftarrow f_a(c)$ and goto 2.
4. output q

Fig. 5. $\mathsf{G}[n]$ – A basic Prime Number Generator Gased on g.

Note that $\gcd(q^{(1)}, \eta) = \gcd(c^{(1)} + \rho, \eta) = \gcd(c^{(1)}, \eta) = 1$ since $c^{(1)} \in \mathbb{Z}^*_{\Pi}$; note also that $q^{(1)} \in \mathcal{F}$. We let $\mathcal{P}_0$ denote the set $(\mathbb{Z}^*_{\Pi} + \rho) \subseteq \mathcal{F}$, and $\mathcal{P}_c$ the set of primes belonging to $\mathcal{P}_0$. For avoiding systematic use of g, rejected candidates should optimally be transformed and re-used in order to continue the search. In this setting, the transition step $c^{(i+1)} = f_a(c^{(i)})$ uses the stability of $\mathbb{Z}^*_{\Pi}$ under multiplication by setting

$$c^{(i+1)} = f_a(c^{(i)}) = a\, c^{(i)} \bmod \Pi \quad \text{and} \quad q^{(i+1)} = c^{(i+1)} + \rho\,, \tag{3}$$

where a is a constant appropriately chosen in $\mathbb{Z}^*_{\Pi}$. We call $\mathsf{G}[n]$ the corresponding algorithm as illustrated in Fig. 5.

The produced *search sequence* $\{q^{(1)}, q^{(2)}, \ldots, q^{(d)}\}$ ends when $q^{(d)} \in \mathcal{P}_c$. Naturally, one has to make sure that the order of f_a (seen as a permutation over $\mathbb{Z}^*_{\Pi}$) is large enough, that is, a's order in $\mathbb{Z}^*_{\Pi}$ must be sufficiently large (since $c^{(i+1)} = a^i\, c^{(1)} \bmod \Pi$): otherwise the search sequence could possibly reach a cyclic set of values without ending.

By denoting $\sigma(n, a)$ and $\tau(n)$ the complexity of f_a and T respectively, and $\mathsf{Comp}(n)$ the average time complexity of $\mathsf{G}[n]$, it can be shown that

$$\mathsf{Comp}(n) = \gamma(n) + (\bar{d} - 1)\, \sigma(n, a) + \bar{d}\, \tau(n)\,, \tag{4}$$

where $\bar{d}$ denotes the average sequence length over many trials. Making the heuristic approximation that the random variables induced by the $q^{(i)}$s are independent and uniformly distributed, we get

$$\bar{d} = \frac{\#\mathcal{P}_0}{\#\mathcal{P}_c} \propto \frac{\phi(\eta)}{\eta}\,. \tag{5}$$

It can be shown that our heuristic algorithm $\mathsf{G}[n]$ outputs random n-bit primes in average time complexity $\mathrm{O}(n^4/\log n)$, although we do not give a proof of this fact here due to the lack of space.

From a practical viewpoint, since g and T are given, the only remaining degree of freedom resides in f_a. Note that $\sigma(n, a)$ is multiplied by a potentially big factor, $\#\mathcal{P}_0/\#\mathcal{P}_c$ in (4), so that decreasing $\sigma(n, a)$ leads to a proportional gain in $\mathsf{Comp}(n)$.

We now specialize Eq. (3) so that the transition step is very fast: the best possible value is $a = 2$. In this respect, we exclude $p_1 = 2$ from Π's factorization.

The benefit is immediate due to the fact that for all $c \in \mathbb{Z}_{\Pi}$, $f_2(c) = 2c \bmod \Pi$ only requires *non-modular* additions: $f_2(c) = 2c$ or $2c - \Pi$. Note here that, since Π is odd, $f_2(c)$ can be odd or even. Hence from Eq. (3), our candidate for primality $q = c + \rho$ can be even! So, in order to avoid useless tests, we suggest the following modification: we define Π as $\Pi = (\varepsilon_{\max} - 1)\eta$ and ρ as in Eq. (2). Next, $q = c + \rho$ is optionally added to η according to its parity so that the resulting q is always odd. Here is the final algorithm. We give practical values for η, Π and ρ to generate 512-bit primes in the next section.

Precomputations: parameters η, $\Pi = (\varepsilon_{\max} - 1)\eta$, and $\rho = \varepsilon_{\min}\,\eta$
Output: a n-bit prime q

1. $c = \mathtt{g}()$
2. $q = c + \rho$
3. if q is even then $q \leftarrow q + \eta$
4. if $\mathsf{T}(q) = \mathtt{false}$ then $c \leftarrow 2c \bmod \Pi$ and goto 2
5. output q

Fig. 6. GPrime$[n]$ – An Optimized Prime Number Generator.

Alternatively, one may use an even Π and fix a to some particular invertible value modulo Π so that multiplying by a requires very few operations (e.g., $a = 2^{16} + 1$).

6 Implementation Results

After having implemented g on Infineon's SLE66CX160S smart-card platform (8-bit CPU and 1100-bit arithmetic crypto-processor) for $n = 512$ and

$$\begin{aligned}\eta &= \mathtt{b16bd1e084af628fe5089e6dabd16b5b80f60681d6a092fc}\\ &\quad\ \mathtt{b1e86d82876ed71921000bcfdd063fb90f81dfd}\\ &\quad\ \mathtt{07a021af23c735d52e63bd1cb59c93cbb398afd}_{16}\,,\\ \Pi &= 1729 \cdot \eta\,,\\ \rho &= 4180 \cdot \eta\,,\end{aligned}$$

we compute a uniformly distributed[5] random invertible number modulo Π in less than 40 ms at 3.57 MHz. Algorithm on Fig. 5 with $a = 2$ runs in about 3.150 seconds in average to generate a 512-bit prime, which is in high accordance with Eq. (4) (T is a basic Fermat test with base 2 running in $\tau(512) \approx 90$ ms). As a direct consequence, this particularly fast smart-card implementation allows 1024-bit RSA keys on-board generation in less than 8 seconds in average. The

[5] Again, we consider the bias of Section 4 to be negligible.

generation of an invertible number using g requires about 2.7 KB of code memory (due to the storage of the θ_i). Such a large memory consumption can be avoided by replacing g with g', which only implies the storage of Π and

$$\lambda(\Pi) = \texttt{1dc6c203d4cc780033f9c5d8d97aa2468a54e3700}_{16} \ .$$

This implementation choice has a little impact on performances since the whole RSA key generation process still runs in less than 10 seconds in average. As a comparison with classical methods, we give on Fig. 5 the (heuristic) expected number of calls to T needed by $\mathsf{G}[n]$ and $\mathsf{H}[n, 10]$.

n	256	384	512	640	768	896	1024
$\mathsf{G}[n]$	18.72	26.12	33.29	40.25	46.90	53.56	59.98
$\mathsf{H}[n, 10]$	28.03	42.04	56.05	70.07	84.08	98.1	112.1

Fig. 7. $\mathsf{G}[n]$ *vs* $\mathsf{H}[n, 10]$ – Heuristic Expected Number of Calls to the Primality Oracle T.

7 Applications

We now apply the previously analyzed tools to some particular contexts. We believe that these techniques constitute a serious improvement on current prime number generators in almost every circumstances, including while implementing ANSI X9.31 recommendations.

7.1 Generation of DSA Primes

Here we focus on the problem of generating a uniformly distributed random n-bit prime $p = 1 + qr$ for a given 160-bit prime q. Trial divisions are intended to check that the candidate p has no prime factor p_i for $i = 1, \ldots, k$. As before, we can advantageously generate r so that p automatically fulfills this condition. It suffices that

$$p \not\equiv 0 \pmod{p_i} \iff r \not\equiv -\frac{1}{q} \pmod{p_i} \qquad \text{for } i = 1, \ldots, k \ . \tag{6}$$

Choosing $\Pi = p_1^{\delta_1} \cdots p_k^{\delta_k}$ with $|\Pi| = |r| = n - 160$, Eq. (6) can be rewritten as

$$r = -\frac{1}{q} + c \bmod \Pi \tag{7}$$

where $c \in \mathbb{Z}_\Pi^*$.

Based on Fig. 5, we therefore propose algorithm $\mathsf{GDSA}[n, q]$. Again, as in Section 5, $\mathsf{g}()$ generates elements of $\mathbb{Z}_\Pi^*$ and $f_a(c) = ac \bmod \Pi$ for some $a \in \mathbb{Z}_\Pi^*$.

As a comparison with classical methods, we also give benchmarks for $\mathsf{GDSA}[n, q]$ and $\mathsf{H}[n, 10]$.

Input: a 160-bit prime q
Output: a n-bit prime satisfying $p = 1 + rq$

1. compute $1/q = q^{\lambda(\Pi)-1} \bmod \Pi$
2. $c = \mathsf{g}()$
3. $r = (-1/q + c) \bmod \Pi$
4. $p = 1 + qr$
5. if $\mathsf{T}(p) = \texttt{false}$ then $c \leftarrow f_a(c)$ and goto 3
6. output p

Fig. 8. $\mathsf{GDSA}[n, q]$ – DSA Prime Generation Algorithm Based on g.

n	256	384	512	640	768	896	1024
$\mathsf{GDSA}[n, q]$	22.37	28.71	35.34	42.06	48.62	55.12	61.6
$\mathsf{H}[n, 10]$	28.03	42.04	56.05	70.07	84.08	98.1	112.1

Fig. 9. $\mathsf{GDSA}[n, q]$ – Heuristic Expected Number of Calls to T.

7.2 Generation of Safe Primes

A prime p is said to be safe if $p = 1 + 2q$ where q is also prime. In order to generate a safe n-bit prime $p = 1 + 2q$, we have to produce a search sequence of pairs $(p^{(i)}, q^{(i)})$ in which $p^{(i)} = 1 + 2q^{(i)}$ and $p^{(i)}$, $q^{(i)}$ are both invertible modulo Π. This can be done by finding for $\Pi = p_1^{\delta_1} \cdots p_k^{\delta_k}$ a value close to 2^{n-2} with maximum k. As we know how to generate an element c of $\mathbb{Z}_\Pi^*$, we propose to test $q^{(1)} = c + \Pi$ and $p^{(1)} = 1 + 2c + 2\Pi$ for primality. By construction, since $c \in \mathbb{Z}_\Pi^*$, $q^{(1)}$ is indeed co-prime to Π and thus makes a good candidate for being a prime: this is however not the case for $p^{(1)}$. For solving this drawback, we propose to modify g into g_s as given in Fig. 10.

Precomputations: $\Pi = \prod_1^k p_i^{\delta_i}$, $t = C \cdot \max_i |p_i^{\delta_i}|$ $(C = 2)$, $\{\theta_i\}$
Input: a random sequence α
Output: a uniformly distributed invertible number $c \in \mathbb{Z}_\Pi^*$ with $1 + 2c \in \mathbb{Z}_\Pi^*$

1. $c = 0$
2. for $i = 1$ to k
   - 2.1 pick a t-bit random number α_i
   - 2.2 if $\alpha_i^{\delta_i} \theta_i \bmod \Pi = 0$ goto 2.1
   - 2.3 if $(1 + 2\alpha_i)^{\delta_i} \theta_i \bmod \Pi = 0$ goto 2.1
   - 2.4 $c \leftarrow c + \alpha_i \theta_i \bmod \Pi$
3. output c

Fig. 10. Generator g_s for Safe Prime Generation.

From this modified generator, we naturally define an algorithm $\mathsf{Gsafe}[n]$ generating safe primes as shown on Fig. 11. Since it appears uneasy to find a low-cost

Ouput: a safe n-bit prime $p = 1 + 2q$ with q prime
1. $c = \mathsf{g}_s()$
2. $q = c + \Pi$
3. $p = 1 + 2q$
4. if $\mathsf{T}(p) = \texttt{false}$ or $\mathsf{T}(q) = \texttt{false}$ goto 1
5. output p

Fig. 11. $\mathsf{Gsafe}[n]$ – Safe Prime Generation Algorithm.

transformation $(p^{(i+1)}, q^{(i+1)}) = f(p^{(i)}, q^{(i)})$ that respects co-primality to Π, g_s is recalled as many times as necessary.

7.3 Application to ANSI X9.31

In order to thwart certain classes of attacks on RSA, the ANSI recommends the use of prime factors satisfying particular properties as exposed in the specifications of X9.31. According to the standard, each prime factor q must be chosen such that

$$\begin{cases} q-1 \text{ has a large prime divisor } u, \\ q+1 \text{ has a large prime divisor } s, \end{cases}$$

where the respective sizes of u and s are chosen close to 100 bits. Primes numbers featuring this property will be called X9.31-compliant primes. We first note that, after having chosen parameters $\eta \approx 2^{99}$ and $\Pi = \rho = \eta$, our algorithm $\mathsf{G}[100]$ outputs two 100-bit prime numbers u and s with an expected complexity of 8.73 primality tests. We still have to generate a n-bit prime q such that

$$q = 1 + r_1 \cdot u = -1 + r_2 \cdot s \ ,$$

where r_1 and r_2 are integers of bit-size $n - 100$. Hence $r_1 \equiv -\frac{2}{u} \pmod{s}$ and there must be an integer r_3 such that

$$q = 1 + u \cdot (-\frac{2}{u} \bmod s + r_3 \cdot s) \ .$$

By a reasoning similar to the one of Section 7.1, we are driven to produce candidates q of the preceding form with

$$r_3 = -\frac{1}{su} - \frac{-2u^{-1} \mod s}{s} + c \mod \Pi \ ,$$

where $c \in \mathbb{Z}_{\Pi}^{*}$ and Π is a product of small primes of total size close to $n - 200$. Note also that the intermediate computations

$$\kappa = 1 + u(-2u^{-1} \mod s)$$

and

$$\mu = -\kappa(su)^{-1} \mod \Pi$$

of respective bit-sizes 200 and $n - 200$ can be done easily in two exponentiations

$$u^{-1} = u^{s-2} \mod s$$

and

$$(su)^{-1} = (su)^{\lambda(\Pi)-1} \mod \Pi\ .$$

This motivates algorithm $\mathsf{Gx9.31}[n]$ illustrated on Fig. 12. As before, a is a constant chosen in $\mathbb{Z}_{\Pi}^{*}$ and $f_a(c) = ac \mod \Pi$. We also give the expected number of calls to the primality oracle T as a function of n on Fig. 13.

Precomputations: Π and $\lambda(\Pi)$
Output: a X9.31-compliant n-bit prime q

1. generate u and s using $\mathsf{G}[100]$
2. compute $\kappa \leftarrow 1 + u \cdot (-2u^{-1} \mod s)$ and $\mu \leftarrow -\kappa(su)^{-1} \mod \Pi$
3. $c \leftarrow \mathsf{g}()$
4. $r \leftarrow \mu + c \mod \Pi$ and $q \leftarrow \kappa + su \cdot r$
5. if $\mathsf{T}(q) = \texttt{false}$ then $c \leftarrow f_a(c)$ and goto 4
6. output q

Fig. 12. $\mathsf{Gx9.31}[n]$ – X9.31-Compliant Prime Generation Algorithm.

n	256	384	512	640	768	896	1024
$\mathsf{Gx9.31}[n]$	25.15	29.64	36.05	42.68	49.18	55.54	61.90

Fig. 13. $\mathsf{Gx9.31}[n]$ – Heuristic Expected Number of Calls to the Primality Oracle T.

7.4 Generation of Strong Primes

A prime number q is said to be *strong* when

$$\begin{cases} q-1 \text{ has a large prime divisor } u, \\ q+1 \text{ has a large prime divisor } s, \\ u-1 \text{ has a large prime divisor } t. \end{cases}$$

The property of being strong therefore implies $X9.31$-compliance. Usually, the bit-sizes of u, s and t are chosen fixed to constant values and hence do not depend on the bit-size of q, n. We will take $|s| = |t| = 100$ and $|u| = 130$ here as an illustrative example, despite the fact that our technique remains fully generic towards these parameters.

Clearly, we can take advantage of the algorithm $\mathsf{Gx9.31}[n]$ of the preceding section and include the additional stage $u = \mathsf{GDSA}[130, t]$ before the search sequence takes place. This can be done by setting $\Pi \approx 2^{29}$ in $\mathsf{GDSA}[130, t]$. This gives the algorithm Gstrong described on Fig. 14.

Precomputations: Π and $\lambda(\Pi)$
Output: a n-bit strong prime q

1. generate s and t using $\mathsf{G}[100]$
   generate u using $\mathsf{GDSA}[130, t]$
2. compute $\kappa \leftarrow 1 + u \cdot (-2u^{-1} \mod s)$ and
   $\mu \leftarrow -\kappa(su)^{-1} \mod \Pi$
3. $c \leftarrow \mathsf{g}()$
4. $r \leftarrow \mu + c \mod \Pi$ and $q \leftarrow \kappa + su \cdot r$
5. if $\mathsf{T}(q) = \texttt{false}$ then $c \leftarrow f_a(c)$ and goto 4
6. output q

Fig. 14. $\mathsf{Gstrong}[n]$ – A Strong Prime Generation Algorithm.

We stress the fact that our technique features a dramatic performance improvement compared to classical methods. To illustrate this, we give a comparison of the average number of calls to T executed by Gstrong and the classical method, Gordon's algorithm.

n	256	384	512	640	768	896	1024
$\mathsf{Gstrong}[n]$	30.34	30.82	36.7	43.1	49.55	56	62.21
Gordon	88.73	133.1	177.45	221.8	266.17	310.53	354.9

Fig. 15. $\mathsf{Gstrong}[n]$ *vs* Gordon – Heuristic Expected Number of Calls to the Primality Oracle T.

7.5 Application in a Shared RSA Protocol

In [3], Boneh and Franklin proposed a shared RSA protocol which enables two parties with the help of a third party to generate a shared RSA key $N = pq$ and $de \equiv 1 \pmod{\phi(N)}$. In this protocol, N and e are public but p, q and d are shared through a secret sharing algorithm.

The Boneh-Franklin protocol enables the two parties to decrypt. One crucial step in this protocol resides in the protocol generating N. Basically, both parties A and B choose (p_A, q_A) and (p_B, q_B) respectively, proceed to a protocol such that they share $N = (p_A + p_B)(q_A + q_B)$, and check that $p = p_A + p_B$ and $q = q_A + q_B$ are simultaneously prime. Prior to this protocol, A and B check whether p and q are not divisible by small primes. In other words, they first generate some shared p and q which have no small prime factors $p_1, \ldots, p_m$ and start again until p and q are both prime. This leads to an expected number of $\left(\frac{e^{-\gamma}\log 2n}{\log p_m}\right)^2$ joint primality tests. As an example, Boneh and Franklin proposed for $n = 512$ that m should be close to 1024 which leads to a number of trial division steps of 32. Alternatively we propose to generate p and q as

$$p = (p'_A p'_B + (p''_A + p''_B)\Pi) \bmod \varepsilon\Pi \quad \text{and} \quad q = (q'_A q'_B + (q''_A + q''_B)\Pi) \bmod \varepsilon\Pi$$

where $\Pi = \prod_{i=1}^{k} p_i$, $\varepsilon\Pi \approx 2^n$ and p'_A, p'_B, q'_A and q'_B are random numbers co-prime to Π generated by generator g, and then to perform trial divisions by $p_{k+1}, \ldots, p_m$. For $m = 1024$ and $k = 74$, letting $\chi = \prod_{i=75}^{1024} p_i$, the number of trials is then $\chi/\phi(\chi)$ which is approximately 3 instead of 32. This drastically reduces the number of exchanged values in the protocol.

8 Conclusion

We introduced new algorithms for generating pseudo-random numbers with no small factors, and showed how to use them in designing prime number generation algorithms to improve related problems. We gave a sketchy expression of our main algorithm's complexity in heuristic terms: this complexity relates to the distribution of prime numbers in the arithmetic progression $a^i c \mod \Pi + \rho$ with $i \geq 0$ and $a, c \in \mathbb{Z}_\pi^*$. Therefore, an open question would be to provide a more formal investigation on the distribution of those primes, the same way Brandt and Damgård [5] characterized the naive incremental generator.

Acknowledgements

We are grateful to the referees for their useful comments.

References

1. ANSI X9.31. Public-key cryptography using RSA for the financial services industry. American National Standard for Financial Services, draft, 1995.
2. A.O.L. Atkin and F. Morain. Elliptic curves and primality proving. *Mathematics of Computation*, vol. 61, pp. 29–68, 1993.
3. D. Boneh and M. Franklin. Efficient generation of shared RSA keys. In *Advances in Cryptology – CRYPTO '97*, vol. 1294 of Lecture Notes in Computer Science, pp. 425–439, Springer-Verlag, 1997.
4. W. Bosma and M.-P. van der Hulst. Faster primality testing. In *Advances in Cryptology – CRYPTO '89*, vol. 435 of Lecture Notes in Computer Science, pp. 652–656, Springer-Verlag, 1990.
5. J. Brandt and I. Damgård. On generation of probable primes by incremental search. In *Advances in Cryptology – CRYPTO '92*, vol. 740 of Lecture Notes in Computer Science, pp. 358–370, Springer-Verlag, 1993.
6. J. Brandt, I. Damgård, and P. Landrock. Speeding up prime number generation. In *Advances in Cryptology – ASIACRYPT '91*, vol. 739 of Lecture Notes in Computer Science, pp. 440–449, Springer-Verlag, 1991.
7. C. Couvreur and J.-J. Quisquater. An introduction to fast generation of large prime numbers. *Philips Journal of Research*, vol. 37, pp. 231–264, 1982.
8. C. Ding, D. Pei, and A. Salomaa. *Chinese Remainder Theorem*, Word Scientific, 1996.
9. FIPS 186. Digital signature standard. Federal Information Processing Standards Publication 186, US Department of Commerce/N.I.S.T., 1994.
10. D.E. Knuth. *The Art of Computer Programming - Seminumerical Algorithms*, vol. 2, Addison-Wesley, 2nd ed., 1981.
11. A.J. Menezes, P.C. van Oorschot, and S.A. Vanstone. *Handbook of Applied Cryptography*, CRC Press, 1997.
12. H.C. Pocklington. The determination of the prime or composite nature of large numbers by Fermat's theorem. *Proc. of the Cambridge Philosophical Society*, vol. 18, pp. 29–30, 1914.
13. H. Riesel. *Prime Numbers and Computer Methods for Factorization*, Birkhäuser, 1985.
14. R.L. Rivest. Remarks on a proposed cryptanalytic attack on the M.I.T. public-key cryptosystem. *Cryptologia*, vol. 2, pp. 62–65, 1978.
15. R.L. Rivest, A. Shamir, and L.M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, vol. 21, pp. 120–126, 1978.
16. R. Solovay and V. Strassen. A fast Monte-Carlo test for primality. *SIAM Journal on Computing*, vol. 6, pp. 84–85, 1977.

Author Index