SPEED

Software Performance Enhancement for Encryption and Decryption

11–12 June 2007

Amsterdam, the Netherlands

Organized by

Vampire
Virtual Application and Implementation Research Lab

within

ECRYPT European Network of Excellence in Cryptography

Program committee:

Local organization:

Invited speakers:

Daniel J. Bernstein, University of Illinois at Chicago, USA
Torbjörn Granlund, Swox, Sweden
Dag Arne Osvik, Ecole Polytechnique Fédérale de Lausanne, Switzerland
Daniel Page, University of Bristol, UK
Matt Robshaw, France Telecom R&D, France

Contributors:

Ashraf Abusharekh, George Mason University, USA
Kazumaro Aoki, NTT Corporation, Japan
Neil Costigan, Dublin City University, Ireland
Thomas Eisenbarth, Horst Görtz Institute for IT Security, Ruhr-Universität Bochum, Germany
Junfeng Fan, Katholieke Universiteit Leuven, Belgium
Krzysztof Gaj, George Mason University, USA
Pierrick Gaudry, Laboratoire Lorrain de Recherche en Informatique et ses Applications, France
Satoshi Oda, NTT Corporation, Japan
Christof Paar, Horst Görtz Institute for IT Security, Ruhr-Universität Bochum, Germany
Axel Poschmann, Horst Görtz Institute for IT Security, Ruhr-Universität Bochum, Germany
Sören Rinne, Horst Görtz Institute for IT Security, Ruhr-Universität Bochum, Germany
Kazuo Sakiyama, Katholieke Universiteit Leuven, Belgium
Michael Scott, Dublin City University, Ireland
Emmanuel Thomé, Laboratoire Lorrain de Recherche en Informatique et ses Applications, France
Leif Uhsadel, Horst Görtz Institute for IT Security, Ruhr-Universität Bochum, Germany
Ingrid Verbauwhede, Katholieke Universiteit Leuven, Belgium
Go Yamamoto, NTT Corporation, Japan

Program and table of contents:

GMP small operands optimization
or
Is arithmetic assembler automatically optimal?

Torbjörn Granlund

Swox, Sweden

GMP as a general purpose low-level arithmetic library has not been specifically tailored for cryptographic applications. Yet it performs well compared to special-purpose crypto libraries. The techniques and algorithms used for crypto-relevant low-precision arithmetic will be explained, and possible future improvments will be examined.

Comparative Analysis of Software Libraries for Public Key Cryptography

Ashraf Abusharekh[1] and Krzysztof Gaj[2]

[1] George Mason University, Fairfax VA 22030, USA,
aabushar@gmu.edu,
[2] George Mason University, Fairfax VA 22030, USA,
kgaj@gmu.edu

Abstract. Software implementations of public key cryptosystems require efficient realization of operations on large integers and elements of the Galois Field. Multiple libraries implementing such operations exist both commercially and in the public domain, in this paper, we perform comparison of eight libraries: CLN, CryptoPP, GNU MP, LiDIA, MIRACL, NTL, OpenSSL and PIOLOGIE, using performance and support of public key primitive operations. The performance of all libraries is ranked based on the measurements performed according to a methodology that takes into account the performance and relative use of primitive cryptographic operations. The performance results shows that GNU MP has the best performance for operations on large integers, OpenSSL has the best performance for operations on elliptic curves over prime fields and LiDIA and MIRACL have the best performance for operations on elliptic curves over binary fields. CryptoPP leads in terms of support for cryptographic primitives and schemes, but is the slowest of all investigated libraries.

1 Introduction

In order to assure the required level of cryptographic strength, mathematical functions used in public key schemes require operations on large integers of size varying between 768 to 2048 bits, as well as elliptic curve operations over fields with element size in the range of 140 to 240 bits. Software implementation of such arithmetic operations is difficult since currently available processors have a limited word-size of up to 64 bit.

Many algorithms have been developed to perform these multi-precision arithmetic operations efficiently, and several libraries implementing such algorithms exist both commercially and in public domain. Nevertheless to our best knowledge, no systematic study has been done to compare and contrast these libraries against each other.

In the study described in this paper, eight libraries have been chosen from the public domain to perform the comparison: CLN[3], CryptoPP[5], GNU MP[6], LiDIA[14], MIRACL[20], NTL[17], OpenSSL[18] and PIOLOGIE[7]. These libraries were chosen based on the authors personal experience and usage .The aim of this study is to evaluate the suitability of using the aforementioned software libraries for implementation on a wide range of public key cryptosystems by using the performance of primitive operations as the main evaluation criterion, then further introducing other secondary criteria such as support for public key primitives and schemes, documentation, ease of use and portability.

Evaluation of software performance is not only considered difficult but also complex due to the increasing number of variables such as, operating system, processor, available memory and the choice of compiler and its optimization options. In order to achieve the performance evaluation, a methodology for ranking the entire libraries is developed based on the performance of their primitive cryptography related operations.

This study is intended to provide the developers of public key software implementations with knowledge needed to make better choices regarding the use of available libraries in their products based on the analysis of existing trade offs.

2 Libraries and Test Platforms

The libraries used in the comparison are listed in Table 1; the majority can be described as multi-precision libraries or number theoretical libraries. The only exceptions are CryptoPP, MIRACL and OpenSSL which specifically target cryptographic schemes.

Table 1. Libraries

Library	Category	License	Version used
CLN	Number theoretic	GNU GPL	1.1.5
CryptoPP	Cryptographic	Copyrighted as a compilation	5.1
GMP	Multi-precision, Number theoretic	GNU GPL	4.1.2
LiDIA	Number theoretic	LiDIA group	2.1pre7
MIRACL	Cryptographic	Shamus Software Ltd.	4.82
NTL	Number theoretic	GNU GPL	5.3.1
OpenSSL	Cryptographic	Apache-style license	0.9.7c
PIOLOGIE	Multi-precision, Number theoretic	www.hipilib.de	1.3.2

CLN, LiDIA and NTL were compiled using GMP as an underlying multi-precision library as recommended by the library developers to achieve maximum speed. The structure of GMP has six function categories; two of them are used by the aforementioned libraries. These two are: mpz, high-level functions for signed/unsigned integer arithmetic, and mpn, low-level functions that operate on natural numbers. Most mpn functions contain machine-dependent code and are used by other function categories including mpz. CLN and NTL use GMP mpn functions to build a different user interface, while LiDIA uses mpz functions. There are two different types of editions of PIOLOGIE; the normal editions, dependent on specific processors, compilers and operating systems; and the special editions, independent of these factors. A special scientific edition v1.3.2 was used in this paper. This edition is distributed under the terms and conditions of the GNU General Public License.

Two machines were used for the performance analysis, 2.0GHz Pentium IV with 512 MB RAM hosting two operating systems, Windows XP (Cygwin) and RedHat Linux 9.0 and 2x 400MHz UltraSPARC-Solaris-II with 4-MB E-cache and 2048 MB RAM hosting Solaris 5.8. The libraries were compiled using GNU C/C++ compiler on all three operating systems using instructions provided by the libraries writers. Measurements were analyzed to obtain a general overall ranking of the libraries with respect to one another on each platform based on the overall rank of each operation.

3 Cryptographic Operations

The primitive cryptographic operations and sizes of operands were chosen based on their use in practical cryptographic algorithms whose security is based on the three well-known mathematical problems, integer factorization, discrete logarithm and elliptic curve discrete logarithm, such as RSA, DSA and ECDSA. The operations are divided into two main sets according to the operand sizes and types. The first set contains operations on large integers: multiplication, modular exponentiation, greatest common divisor and multiplicative inverse (extended greatest common divisor), with operands sizes 768, 1024 and 2048 bits.

The second set, contains operations on elliptic curve points; point addition and scalar multiplication with base point order lengths 163, 233 and 409 bits (equivalent to 1024, 2240 and 7680 bit RSA/DSA keys [21]) for elliptic curves over binary fields ($E(F_{2^n})$) and 162, 226 and 386 bits (equivalent to 1024, 2048 and 7680 bit RSA/DSA keys[21]) for randomly generated elliptic curves over prime fields ($E(F_p)$).

3.1 Large Integer Operations

Multiplication: Multiplication Algorithms implemented in the libraries are summarized in Table 2. The Karatsuba[1][11] algorithm has a running complexity of $O(n^{log\, 3})$ which is an improvement over the classical multiplication [11] algorithm at $O(n^2)$. Classical, Comba[4] and Karatsuba multiplication algorithms are of practical importance for the operand sizes used in the performance testing. The Toom-Cook (T-C) algorithm[11], with a running complexity

Table 2. Multiplication Algorithm Ranges

	CLN	CryptoPP	GMP LiDIA NTL/PIV	GMP LiDIA NTL/SPARC	OpenSSL	PIOLOGIE
Classical	[0,1120)	-	[0,576)	[0,1280)	[0,512)	[0,256]
Comba	-	[0,256]	-	-	-	-
Karatsuba	[1120,80000)	> 256	[576,4448)	[1280,7104)	≥ 512	(256,160000)
T-C	-	-	[4448,188416)	[7104,122800)	-	-
FFT	≥ 80000	-	≥ 188416	≥ 122800	-	≥ 160000

of $O(n\ 2^{\sqrt{2\ log\ n}}\ log\ n)$ and Fast Fourier Transform (FFT) algorithm[11], with running complexity of $O(n\ log\ n\ log\ log\ n)$ are asymptotically superior to Karatsuba algorithm. However these algorithms do not offer any speed improvements for the operand sizes currently used in public key cryptography. Table 3 shows the different operand sizes in bits and the algorithms used for their multiplication. All libraries were compiled using default ranges and thresholds, however these thresholds can be changed to best fit the underlying platform/processor. GMP is the only library that adjusts the threshold not only depending on operand sizes, but also on the underlying processor architecture, a set of tune up programs are supplied with GMP that can be invoked on the targeted machine to measure the timing of GMP routines and propose thresholds that produce better results. The library must be recompiled in order for the change to be effective. This directly affects LiDIA and NTL which use GMP implementations of multiplication algorithms. On the other hand CLN uses GMPs mpn functions to build its own multiplication algorithms as a result is not directly affected.

CryptoPPs implementation of the Karatsuba algorithms requires the input sizes to be powers of 2. In case they are not, they have to be extended to the next power of 2 before applying the algorithm e.g. an input of size 768 bit is extended to 1024 bits. Although, OpenSSLs implementation of Karatsuba-Comba algorithm also requires the input sizes to be powers of 2, classical multiplication is used when this condition does not hold. MIRACL implements classical multiplication for all sizes.

Modular Exponentiation: Modular exponentiation algorithms implemented in the libraries and their corresponding thresholds are summarized in Table 4. Left-to-right[15] (denoted as LR) and right-to-left[15] (denoted as RL) algorithms require $L(E) - 1$ squarings, where $L(E)$ is the bitlength of the exponent, and $W(E) - 1$ multiplications, where $W(E)$ is the Hamming weight of the exponent. Both algorithms do not require precomputations. Left-to-right

Table 3. Multiplication Algorithms for Different Key Sizes

Library	Classical	Karatsuba
Piologie	-	768, 1024, 2048 bits
OpenSSL	768 bit	1024, 2048 bits
MIRACL	768, 1024, 2048 bits	-
GMP/LiDIA/NTL{SPARC}	768, 1024 bits	2048 bits
GMP/LiDIA/NTL{PIV}	-	768, 1024, 2048 bits
CryptoPP	-	768, 1024, 2048 bits
CLN	768, 1024 bits	2048 bits

k-ary[15](denoted as LR k-ary), simultaneous multiple exponentiation[15] (denoted as SME) and sliding window[12][15] (denoted as k-ary SW) algorithms need precomputations. Table 5 shows the algorithms used by the respective libraries for three different exponents, $E = 3$, $E = 65537$, and a random exponent the same size as the size of the modulus N.

Table 4. Modular Exponentiation Algorithm Ranges

Library	CLN	CryptoPP	GMP	LiDIA	MIRACL	NTL	OpenSSL	PIOLOGIE
LR	[2,8]	-	[2,32]	-	-	[2,512)	-	-
RL	-	-	-	≥ 2	-	-	-	≥ 2
LR k-ary	(8,)	-	-	-	-	-	-	-
SME	-	≥ 2	-	-	-	-	-	-
k-ary SW	-	-	> 32	-	≥ 2	≥ 512	≥ 2	-

GCD and xGCD: Table 6 summarizes the libraries implementations for the GCD and xGCD algorithms. Complete description and analysis of these algorithms can be found in [11].

Table 5. Modular Exponentiation Algorithms for Different Exponent Sizes

Library	LR	RL	LR k-ary	k-ary SW	SME
Piologie	-	E=3, E=65537, E	-	-	-
OpenSSL	-	-	-	E=3, E=65537, E	-
NTL	E=3	-	-	E=65537, E	-
MIRACL	-	-	-	E=3, E=65537, E	-
LiDIA	-	E=3, E=65537, E	-	-	-
GMP	E=3	-	-	E=65537, E	-
CryptoPP	-	-	-	-	E=3, E=65537, E
CLN	E=3	-	E=65537, E	-	-

Table 6. GCD and xGCD Algorithms

Library	GCD	xGCD
CLN	Lehmer[13][11][15]	Lehmer
CryptoPP	Euclid[11][15]	Binary[11][15]
GMP/LiDIA/NTL	Generalized Binary[10][23]	Lehmer
MIRACL	Lehmer	Lehmer
OpenSSL	Binary	Binary
PIOLOGIE	Generalized binary	Euclid

3.2 Elliptic Curve points

Operations on elliptic curves are limited to four libraries: CryptoPP, LiDIA, MIRACL and OpenSSL. The version of OpenSSL used supports only $E(F_p)$, and does not support $E(F_{2^n})$.

Scalar Multiplication: Table 7 summarizes the libraries implementations of EC point scalar multiplication.

4 Methodology

Measurements were conducted in two different ways depending on the platform. The first method of testing referred to as RDTSC method was used on Pen-

Table 7. Elliptic Curve Scalar Multiplication Algorithms

Library	Scalar Multiplication
CryptoPP	Simultaneous Sliding Window
LiDIA	Left-to-Right
MIRACL	wNAF-based interleaving [16]
OpenSSL	wNAF-based interleaving

tium IV platforms. The RDTSC method uses the RDTSC[9][2] (read time-stamp counter) instruction to access the time-stamp counter, a 64 bit model specific register that is incremented every clock cycle, present on Intel processors beginning with the Pentium processor. The CPUID instruction is used as a serializing instruction to prevent out-of-order execution. The RDTSC method was used to determine the number of clock cycles required to perform the given operation. The overhead associated with the call of the instructions was calculated and subtracted from the final result. Both instructions were called several times before testing the given operation to flush the instruction cache.

The second method referred to as Timing method was used on the UltraSPARC platform. The Timing method uses the function `gettimeofday()`[1] to determine the amount of time in milliseconds consumed in the execution of the given operation due to the lack of CPU cycle counter in the UltraSPARC platform. The function `gettimeofday()` gives resolution in the range of microseconds.

4.1 Operands

For large Integer Operations, two groups of operands were used: Group A, a group of randomly generated integers containing three sets of numbers with sizes 768, 1024 and 2048 bits respectively. Each set contains three large integers denoted as I_i, J_i and K_i, where i is the size of the integer in bits, e.g. the first set contains I_{768}, J_{768}, K_{768} . The values of I_i, J_i and K_i are listed in Appendix A to [24]. Group B, a group of randomly generated large prime numbers contains three sets of numbers with sizes 768, 1024 and 2048 bits respectively. Each set contains ten large prime integers denoted as P_i^j, where i is the size of an integer in bits and j is the index of a given prime in the set. For example, the first set, contains P_{768}^0, P_{768}^1, ... P_{768}^9. Table 8 summarizes the operations tested and the corresponding groups of operands. With respect to a particular library under a particular operating system, each operation using group A of operands is tested using either RDTSC or Timing method. All operations are tested using three operand sizes. Thus, each experiment on a given operation produces three sets of 100 execution times. Each value represents one

Table 8. Large Integer Operations

Operation	OP	Group	Comments
Multiplication	MUL	A	$I_i \times J_i$
MOD Exp E = 3	E_3	A	I_i^3 MOD K_i
MOD Exp E = 65537	E_{65537}	A	I_i^{65537} MOD K_i
MOD Exp E, size of modulus	E	A	$I_i^{J_i}$ MOD K_i
Greatest Common Divisor	GCD	B, A	GCD(P_i^j, K_i)
Extended GCD	xGCD	B, A	xGCD(P_i^j, K_i)

iteration while each set represents one operand size. The three sets of 100 execution times are sorted and the minimum value for each set is recorded and denoted as $LIB^{OP}_{AMIN_{768}}$, $LIB^{OP}_{AMIN_{1024}}$, $LIB^{OP}_{AMIN_{2048}}$. The final set of raw results for each operation tested on a particular library under a certain operating system OS is denoted by $LIB^{OP}_{OS} = \{\ LIB^{OP}_{AMIN_{768}}, LIB^{OP}_{AMIN_{1024}}, LIB^{OP}_{AMIN_{2048}}\ \}$. The same approach was used with operations using group B, except that each operation is tested using 10 different operands P_i^j of the same size e.g. there are 10 different operands for 768 bits P^j_{768}, $0 \leq j \leq 9$ so for each j there will be 100 different values. For each P_i^j the 100 values are sorted and the minimum recorded and denoted as $LIB^{OP}_{BMIN^j_i}$. The 10 minimum values for each operand size ($LIB^{OP}_{BMIN^j_{768}}$, $LIB^{OP}_{BMIN^j_{1024}}$, $LIB^{OP}_{BMIN^j_{2048}}$),$0 \leq j \leq 9$ are then averaged, the result is denoted as $LIB^{OP}_{BAVG_i}$.

$$LIB^{OP}_{BAVG_i} = \frac{1}{10}\sum_{j=0}^{9} LIB^{OP}_{BMIN^j_i} \qquad (1)$$

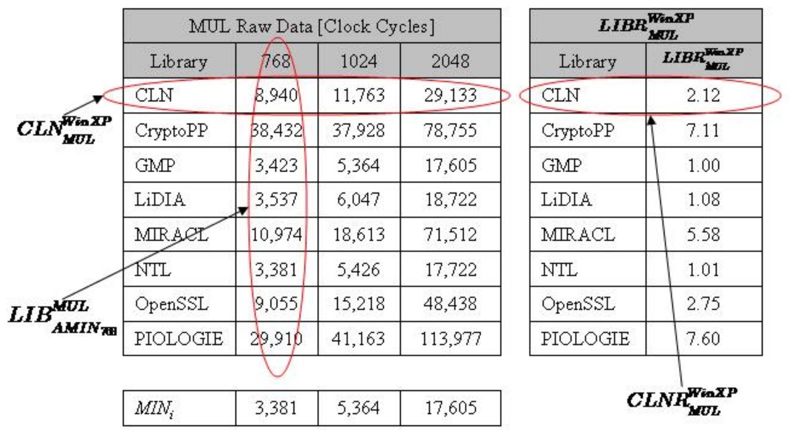
The final set of raw results for each operation in a particular library under a certain operating system is denoted by $LIB^{OP}_{OS} = \{\ LIB^{OP}_{BAVG_{768}}, LIB^{OP}_{BAVG_{1024}}, LIB^{OP}_{BAVG_{2048}}\ \}$.

Elliptic curve point operations tested are Point Addition and Scalar Multiplication with input sizes of 163, 233 and 409 bits for $E(F_{2^n})$ and 162, 226 and 386 bits for $E(F_p)$. For each elliptic curve, two randomly generated points T_i and S_i (i = 163, 233, 409 for $E(F_{2^n})$, i = 162, 226, 386 for $E(F_p)$) were used as operands. Addition: $T_i + S_i$. Scalar Multiplication: (r - 2) T_i were r is the order of the base point. The final raw results are collected as described for large integer operations using group A of operands. The elliptic curves and points are listed in Appendix A to [24].

4.2 Operation Ranking

After obtaining all values of execution times for all operations of the eight libraries under the three operating systems, rankings of the operations were calculated as follows: With respect to an operation OP tested on eight libraries under a particular operating system OS, execution times of OP are rearranged into three sets of eight values such that each set contains the results for a particular operand size under the eight libraries (one value for each execution time under a particular library i.e. $LIB^{OP}_{AMIN_i}$ or $LIB^{OP}_{BAVG_i}$ according to the operand group). The minimum value in each set, denoted by MIN_i, where i = 768, 1024, 2048, is determined and all values in a given set are divided by that value. The resulting values, denoted by $LIB_i r^{OS}_{OP}$ represent operation OP ranks with operands of size i on library LIB. For an operation OP, $LIB_i r^{OS}_{OP} = 1.00$ corresponds to the fastest library. A rank equal to r means that an operation under a given

Fig. 1. CLN Multiplication Rank


CLN^{WinXP}_{MUL}
$LIB^{MUL}_{AMIN_{768}}$
$CLNR^{WinXP}_{MUL}$

MUL Raw Data [Clock Cycles]			
Library	768	1024	2048
CLN	8,940	11,763	29,133
CryptoPP	38,432	37,928	78,755
GMP	3,423	5,364	17,605
LiDIA	3,537	6,047	18,722
MIRACL	10,974	18,613	71,512
NTL	3,381	5,426	17,722
OpenSSL	9,055	15,218	48,438
PIOLOGIE	29,910	41,163	113,977
MIN_i	3,381	5,364	17,605

$LIBR^{WinXP}_{MUL}$	
Library	$LIBR^{WinXP}_{MUL}$
CLN	2.12
CryptoPP	7.11
GMP	1.00
LiDIA	1.08
MIRACL	5.58
NTL	1.01
OpenSSL	2.75
PIOLOGIE	7.60

library is r times slower than the same operation under the fastest library for a given operand size i. Operation OP overall rank under a library LIB denoted by $LIBR^{OS}_{OP}$ is the geometric mean of its ranks for the three operand sizes 768, 1024 and 2048. Figure 1 shows the raw results and ranks for multiplication under Pentium IV-Windows XP for all libraries. The final rank of CLN multiplication is calculated as follows:

$$CLN_{768} r^{WinXP}_{MUL} = \frac{CLN^{MUL}_{AMIN_{768}}}{MIN_{768}} = \frac{8,940}{3,381} = 2.64 \tag{2}$$

$$CLN_{1024} r^{WinXP}_{MUL} = \frac{CLN^{MUL}_{AMIN_{1024}}}{MIN_{1024}} = \frac{11,763}{5,364} = 2.19 \tag{3}$$

$$CLN_{2048} r^{WinXP}_{MUL} = \frac{CLN^{MUL}_{AMIN_{2048}}}{MIN_{2048}} = \frac{29,133}{17,605} = 1.65 \tag{4}$$

$$CLNR_{MUL}^{WinXP} = \sqrt[3]{\prod_i CLN_i r_{MUL}^{WinXP}} = \sqrt[3]{2.64 \times 2.19 \times 1.65} = 2.1208 \qquad (5)$$

4.3 Library Ranking

As a result we will have a set of operation rankings for each library on each operating system; the overall rank of the library denoted by $LIBR^{OS}$ on a particular operating system OS is determined by calculating the geometric mean of its individual operation ranks.

$$LIBR^{OS} = \sqrt[N]{\prod_{OP} LIBR_{OP}^{OS}} \qquad (6)$$

N is the number of operations considered. N = 6 for large integer operations and N=2 for EC point operations. The two sets of rankings are considered separately because EC point operations are not supported by all libraries.

5 Performance Results

As discussed in the previous sections, the following tables show the individual operations, their ranks and the overall rankings of the libraries on the three platforms used for performance testing. Each table lists the individual operation ranking of each library and the overall ranking of the library (Geometric Mean of individual operation rankings).

5.1 Operations On Large Integers

The tables presented in this section show the overall ranking of the libraries for operations on large integers. The operations rankings are, MUL: Multiplication ranking, E_3: Modular Exponentiation Ranking with exponent = 3, E_{65537}: Modular Exponentiation Ranking with exponent = 65537, E: Modular Exponentiation Ranking with exponent of the same size as the modulus, GCD: Greatest Common Divisor ranking, and xGCD: Extended Greatest Common Divisor ranking.

Table 9 lists the performance results under Pentium IV, Windows XP. In terms of the overall rank $LIBR^{WinXP}$, GMP has the best rank and PIOLOGIE the worst. MIRACL and OpenSSL are very close. OpenSSL Multiplication and Modular Exponentiation are higher in rank than MIRACL; while MIRACL GCD and xGCD rank higher than OpenSSL. CryptoPP GCDs rank is higher than xGCDs rank unlike all other libraries while it is completely the opposite for PIOLOGIE; in both cases the reason behind that is the choice of algorithms (see Table 6). Table 10 lists the performance results under Pentium IV, RedHat 9.0. In terms of the overall rank $LIBR^{RH}$, GMP has the best rank and CryptoPP the worst. CryptoPP and PIOLOGIE are slower under RedHat than under Windows

XP; the order of GMP, NTL, LiDIA and CLN is the same as for Pentium IV-Windows XP. MIRACLs rank is now slightly better than OpenSSLs due to rankings of GCD and xGCD. PIOLOGIE rank is slightly better than CryptoPP due to Modular Exponentiations rank with $E = 3$ and GCDs rank. Table 11 lists the performance results under Ultra-SPARC, Solaris. In terms of the overall rank (LIB-R)SPARC, GMP has the best rank and CryptoPP the worst. GMP, NTL, LiDIA and CLN have the same order. PIOLOGIE has a better ranking than Pentium IV, while OpenSSLs rank remains the same.

Under all operating systems, LiDIA and NTL use GMP functions for Multiplication, GCD and xGCD. This makes their rankings according to these operations very much close to GMP. For Modular Exponentiation, NTL and LiDIA have their own different implementations, with NTL choice of algorithms similar to GMP. CLN has its own implementation for all operations. As an overall result for large integer operations, GMP has the best ranking under all three platforms followed by NTL, LiDIA and CLN.

5.2 Operations On $E(F_{2^n})$ and $E(F_p)$ Points

Tables 12 and 13 show the performance results for operations on $E(F_{2^n})$ and $E(F_p)$ points respectively, on all platforms.

For $E(F_{2^n})$, LiDIA has the best rank under Pentium IV,RedHat 9.0 and Ultra-SPARC,Solaris while CryptoPP has the worst rank under all platforms. MIRACL has the best rank under Pentium IV, Windows XP.

For $E(F_p)$, under all platforms, OpenSSL has the best rank and CryptoPP has the worst rank.

Table 9. Large Integer Operation Rankings Pentium IV, Windows XP

Library	MUL	E_3	E_{65537}	E	GCD	xGCD	$LIBR^{WinXP}$
CLN	2.12	2.23	2.25	2.79	1.34	1.37	1.95
CryptoPP	7.11	15.17	4.71	4.04	464.90	9.99	14.56
GMP	1.00	1.00	1.00	1.00	1.01	1.08	1.01
LiDIA	1.08	1.45	1.08	1.65	1.03	1.10	1.21
MIRACL	3.58	22.40	4.56	2.62	5.15	3.15	5.00
NTL	1.01	1.42	1.17	1.18	1.00	1.00	1.12
OpenSSL	2.75	8.07	2.65	2.33	8.31	12.17	4.90
PIOLOGIE	7.60	7.40	6.63	10.65	16.41	213.30	15.51

Table 10. Large Integer Operation Rankings Pentium IV, RedHat 9.0

Library	MUL	E_3	E_{65537}	E	GCD	xGCD	$LIBR^{RH}$
CLN	1.50	1.27	1.39	1.87	1.37	1.27	1.43
CryptoPP	4.49	9.19	3.79	5.04	65.96	16.82	9.78
GMP	1.00	1.00	1.01	1.00	1.00	1.08	1.01
LiDIA	1.00	1.10	1.06	1.84	1.00	1.09	1.15
MIRACL	3.60	21.06	4.30	2.77	3.99	2.36	4.52
NTL	1.01	1.20	1.10	1.29	1.01	1.00	1.10
OpenSSL	2.80	7.12	2.43	2.43	8.93	12.49	4.86
PIOLOGIE	5.35	5.01	5.07	8.79	21.95	24.22	9.27

Table 11. Large Integer Operations Rankings UltraSPARC, Solaris

Library	MUL	E_3	E_{65537}	E	GCD	xGCD	$LIBR^{SPARC}$
CLN	1.21	1.60	1.70	1.98	1.67	1.40	1.58
CryptoPP	16.43	38.52	18.10	17.68	184.68	49.08	34.99
GMP	1.00	1.00	1.00	1.00	1.00	1.12	1.02
LiDIA	1.00	1.20	1.10	1.55	1.02	1.14	1.16
MIRACL	9.08	23.85	2.98	7.41	8.77	3.71	7.33
NTL	1.00	1.25	1.15	1.13	1.05	1.00	1.09
OpenSSL	2.16	7.80	2.81	2.45	7.67	7.74	4.36
PIOLOGIE	3.51	4.13	4.06	5.95	9.13	37.22	7.01

6 Observations And Comments

6.1 Portability, Documentation and Ease of Use

The number of supported compilers was considered as a measure of portability of a given library. For CLN, GMP and LiDIA the only supported compiler is GNU C/C++. PIOLOGIE supports the largest set of compilers followed by CryptoPP, OpenSSL, MIRACL and NTL.

In terms of documentation and ease of use, PIOLOGIE simple structure makes it the easiest among all libraries, on the other hand CryptoPP complex structure and insufficient documentation makes it the hardest among all

Table 12. $E(F_{2^n})$ Operations Rankings

	SPARC			PIV, RH			PIV, WinXP		
Library	Add	Scalar MUL	$LIBR^{SPARC}$	Add	Scalar MUL	$LIBR^{RH}$	Add	Scalar MUL	$LIBR^{WinXP}$
CryproPP	17.1	16.65	16.87	9.55	8.67	9.1	4.56	4.35	4.46
LiDIA	1	1	1	1	1	1	1.33	1.22	1.28
MIRACL	1.17	1.15	1.16	1.48	1.32	1.4	1	1	1

Table 13. $E(F_p)$ Operations Rankings

	SPARC			PIV, RH			PIV, WinXP		
Library	Add	Scalar MUL	$LIBR^{SPARC}$	Add	Scalar MUL	$LIBR^{RH}$	Add	Scalar MUL	$LIBR^{WinXP}$
CryproPP	9.46	7.15	8.23	5.4	2.38	3.58	6.02	3.72	4.73
LiDIA	1.05	1.73	1.35	1.06	1.44	1.24	1.97	3.45	2.61
MIRACL	1.49	2.6	1.97	1.61	2.02	1.8	1.05	2.24	1.53
OpenSSL	1	1	1	1.02	1	1.01	1.3	1	1.14

libraries. CLN, GMP, LiDIA, MIRACL and NTL have complex structure but their documentation and documentation sample code and test suites decrease their difficulty.

6.2 CryptoPP GNU C/C++ vs. MS VC++ 6.0

The CryptoPP's performance was tested under MS VC++ 6.0 and the results were compared to the results obtained under GNU C/C++. The execution time ratios for GNU C/C++ vs MS VC++ 6.0 were computed for multiplication and modular exponentiation for three input sizes under the Pentium IV, Windows XP machine. It was found that CryptoPP compiled under MS VC++ 6.0 is more than twice as fast as that compiled under GNU C/C++. This is due to the library's optimization for Pentium IV processors under MS VC++ 6.0 versus its generic Pentium optimization under GNU C/C++. MIRACL and PIOLOGIE were also compiled under MS VC++ 6.0 with no significant change in their performance as compared to GNU C/C++.

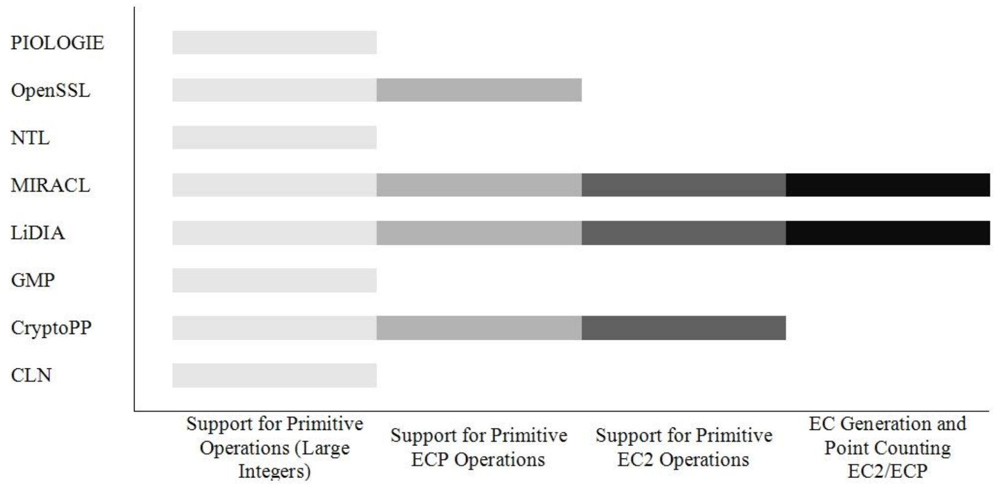
6.3 Support For Public Key Cryptosystems

Support for public key cryptosystems is based on the support of primitive arithmetic and number theoretical operations needed by the three main categories of public key cryptosystems, and also on complete implementations of public key schemes present in the library.

Support for Primitive Operations Figure 2 summarizes the libraries support for primitive operations on large integers, $E(F_p)$ and $E(F_{2^n})$. Support for $E(F_p)$ operations is limited to CryptoPP, LiDIA, MIRACL and OpenSSL. Support for $E(F_{2^n})$ is limited to CryptoPP, LiDIA and MIRACL. Elliptic curve generation and point counting is supported by LiDIA and MIRACL only.

Support for Cryptographic Schemes Complete implementation of Cryptographic schemes is limited to three libraries, CryptoPP, MIRACL and OpenSSL. CryptoPP has the largest collection of public key cryptographic schemes followed by MIRACL and OpenSSL. Moreover, CryptoPP contains a collection of secret key ciphers, hash functions, and MAC functions and I/O support. Version 5.0.4 of the library has received FIPS 140-2 level 1 validation in 9/5/2003. MIRACL implements cryptographic primitives in IEEE P1363[8]. It also has implementations for AES and SHA (1, 256, 384, and 512). OpenSSL implements DH, DSA and RSA, and a collection of secret key ciphers, hash functions and MAC functions.

Fig. 2. Support for Primitive Operations



7 Conclusion

In terms of support for operations on large integers, GMP, NTL, LiDIA, and CLN have the best performance under all platforms tested, with GMP being the fastest and CLN the slowest among the group. LiDIA is the only library in the group that needs a license for commercial use. For a developer targeting operations on large integers, GMP would be the best choice in terms of performance. The trade off however, is the amount of time and effort needed for implementation, and portability.

OpenSSL and MIRACL trail libraries from the first group in terms of overall performance. OpenSSL is faster than MIRACL for all operations except GCD and xGCD. While having an acceptable performance as compared to other libraries and support implementations of cryptographic schemes, this group is a good choice for fast development of public key cryptosystems based on operations on large integers.

CryptoPP is the best choice for the fast development based on the complete implementations of a wide range of cryptographic schemes involving large integers; the drawback is its performance as compared to other libraries.

Fig. 3. Support vs Performance, Pentium IV, RedHat 9.0

Support	**PKS**(1)		CryptoPP	MIRACL OpenSSL	
	EC2	**PG**(2) **PC**(3)	CryptoPP	MIRACL LiDIA	
	ECP	**PG** **PC**	CryptoPP	MIRACL LiDIA OpenSSL	
	LINT	**High**	CryptoPP	MIRACL OpenSSL	CLN LiDIA NTL GMP
		Low	PIOLOGIE		
			low	Performance	high

(1)PKS: Public Key Schemes.
(2)PG: Point Generation.
(3)PC: Point Counting.

For elliptic curves over binary fields, the competition is between LiDIA, MIRACL and CryptoPP. LiDIA has the best performance under Pentium IV-RedHat 9.0 and UltraSPARC-Solaris. Under Pentium IV, Windows XP, MIRACL has the best performance. CryptoPP is the slowest under all platforms.

For elliptic curves over prime fields, OpenSSL has the best performance under all platforms, LiDIA performance is better than MIRACL on Pentium IV-RedHat 9.0 and UltraSPARC-Solaris, while under Pentium IV-Windows XP, MIRACL is better than LiDIA. Again CryptoPP has the lowest performance. For a developer targeting $E(F_p)$ cryptosystems, OpenSSL is a good choice since it has the best performance, and is portable and free.

Although public key schemes implemented in CryptoPP, MIRACL and OpenSSL were not compared for performance, one can estimate their performance based on the performance of primitive operations. Accordingly, OpenSSL is expected to have better performance followed by MIRACL and CryptoPP respectively, with CryptoPP having the richest collection of cryptographic schemes.

Figure 3 summaries the libraries support of public key cryptosystems versus their performance under Pentium IV, RedHat 9.0. The x-axis represents the performance scale from low performance to high performance, y-axis represents support starting from support for large integers (LINT) and ending with support for public key schemes (PKS). GMP has the highest performance and lowest support, while CryptoPP has the highest support and lowest performance.

This work has been done before the introduction of eBATS [24], and currently we are trying to use it to test the various asymmetric operations implemented in the eight libraries . We hope that this will give more insight on some of the reasons why one library cryptographic operation might perform better than the others under a specific platform.

References

1. A. Karatsuba and Yu. Ofman, Multiplication of Multidigit Numbers on Automata. Soviet Physics-Doklady, 7 (1963), 595-596.
2. R. E. Bryant and D. O'Hallaron. Computer Systems, A Programmer's Perspective. Prentice-Hall, 2003.
3. CLN: Class Library for Numbers http://www.ginac.de/CLN/
4. P. G. Comba. Exponentiation cryptosystems on the IBM PC. IBM Systems Journal, vol. 29, n. 4, pp. 526538, 1990.
5. Crypto++ Library 5.1: a Free C++ Class Library of Cryptographic schemes http://www.eskimo.com/ weidai/cryptlib.html
6. The GNU MP Library http://www.swox.com/gmp/
7. HiPiLib Piologie http://www.hipilib.de/piologie.htm
8. IEEE P1363: Standard Specifications for Public-Key Cryptography. http://grouper.ieee.org/groups/1363/
9. Intel Corporation. IA-32 Intel Architecture, Software Developers Manual, vol 2B: Instruction Set Reference, N-Z http://developer.intel.com/design/pentium4/manuals/25366713.pdf
10. Tudor Jebelean. A Generalization of the Binary GCD Algorithm. ISSAC 93, 111-116.
11. D.E. Knuth.The Art in Computer Programming. Vol2 : Seminumerial Algorithms. Addison-Wesley, 2nd.Ed. 1981.
12. C.K. Koc. Analysis of sliding window techniques for exponentiation. Computers and Mathematics with Applications, vol.30, n.10, pp.1724,195.
13. D. H. Lehmer. Euclid's Algorithm for Large Numbers. American Mathematical Monthly. 45 (1938), 227-233.
14. LiDIA: A C++ Library For Computational Number Theory http://www.informatik.tu-darmstadt.de/TI/LiDIA/
15. A. Menesez, P. van Oorschot, and S. Vanstone. Handbook of Applied Cryptography. CRC Press, 1997.
16. B. Möller. Algorithms for multi-exponentiation. Selected Areas in Cryptography SAC 2001 (2001), S. Vaudenay and A.M. Youssef (Eds.), LNCS 2259, pp. 165180.
17. NTL: A Library for doing Number Theory http://www.shoup.net/ntl/
18. OpenSSL: The Open Source toolkit for SSL/TLS http://www.openssl.org/
19. R. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public Key Cryptosystems. Communications of the ACM, vol. 21, no.2, pp. 158-164, 1978.
20. Shamus Software Ltd MIRACL http://indigo.ie/ mscott/
21. Standards for Efficient Cryptography Group. SEC 2: Recommended Elliptic Curve Domain Parameters. Version 0.6, 1999.
22. S. Y. Yan. Number Theory for Computing. Springer-Verlag 2000
23. T. Jebelean. A Generalization of the Binary GCD Algorithm. ISSAC 93, pp. 111-116.
24. eBATS: ECRYPT Benchmarking of Asymmetric Systems http://www.ecrypt.eu.org/ebats/

Fast Integrity for Large Data

Go Yamamoto, Satoshi Oda, Kazumaro Aoki

NTT Corporation

Abstract. We present a scheme that provides data integrity with extreme speed. The proposed scheme bases on a different design than those from hash functions. Like a hash function a deterministic algorithm outputs a small string called tag with a message on input, however, another algorithm probabilistically verifies integrity of the message without recomputing the tag. We implemented the integrity checking algorithm to deliver speed over 12 Gbit/second for 80bit-security on Athlon X2 2GHz. The scheme is collision-resistant under the hardness of integer factorization.

Keywords Data Integrity, Collision-Resistant, Integer Factoring, Fast Implementation

1 Introduction

When we transfer data over open networks, data can be changed possibly by intentional attempts, as well as by unintentional accidents. In this paper we mean data integrity in the sense against intentional change attempts. A standard way to deliver data integrity is to compute a cryptographic hash function.

Speed of integrity checking is important. One of the most important application is in digital signatures. In principle a digital signature is usually issued over only short messages, so when we sign a large message M, we sign $h(M)$ instead, where h is a hash function. In practice, it would be convenient to attach a hash list $H = (h(M_1), h(M_2), \ldots, h(M_d))$ and sign on $h(H)$, where each M_i is a message block from M with appropriate padding. Since integrity of M is verified by recomputing each $h(M_i)$, if integrity checking on some M_i fails, then we can correct only M_i. In both uses, speed of hash function is important as well as that of digital signature. The throughputs of SHA-1 and SHA-256 are 1.4 Gbit/second and 0.56 Gbit/second respectively, where we measured the assembly implementations from OpenSSL on a core of Athlon X2 2GHz.

In this paper we present an alternative way to provide integrity on data and a hash list, desiring speed from 5 Gbit/second to over 10 Gbit/second.

For example, if signature verification is faster than the speed of the network, then we can design a cache server that stores only authorized files. The performance of next generation Ethernet will be 10 Gbit/second [6].

The proposed scheme bases on a different design than those from hash functions, consisting of 2 algorithms *Tag Generation* and *Integrity Checking.* Tag Generation outputs a small string called tag with a message on input, like a hash function. However, Integrity Checking verifies integrity of the message without recomputing the tag. We call such a design *asymmetric integrity checking.* In the proposed scheme, the Integrity Checking algorithm does not recompute tag and is probabilistic, while the Tag Generation algorithm is deterministic.

The Integrity Checking algorithm is extremely efficient. We describe an implementation for the Integrity Checking algorithm to obtain 3.13bit/cycle on a core of Athlon X2. Throughput is over 12Gbit/second on a dual-core implementation running at 2GHz. The size of the tag is about 6.4 times larger than the hash list from SHA-1. However, in digital signatures it suffices to sign $h(T)$, where T is the tag for the message M.

The point of the proposed algorithms is batch processing [5] for a homomorphic hash function. That is, if h is a homomorphic function, then to check that all of $y_1 = h(x_1)$, $y_2 = h(x_2)$, …, $y_d = h(x_d)$ hold, it suffices to check $\bar{y} = h(\bar{x})$, where $\bar{x}$ and $\bar{y}$ are linear combinations of x_i and y_i respectively with a weight randomly chosen by the verifier. In particular when $h(x)$ is set to $g^x \pmod{N}$, where N is a number whose prime factors are unknown, then the integrity checking algorithm is close to short exponent test [2], except for the amount of the required random bits.

Because of the construction above, the scheme is collision-resistant in the sense that it is hard to find a tag T and distinct messages M and M' such that (T, M') is accepted by the Integrity Checking algorithm with non-negligible probability while T is the tag for M. Recent researches [16, 15], for example, reveal that MD4, MD5, and SHA-1 are not as secure as considered before. It is nice that we have a scheme for data integrity with extreme speed that is provably secure under well-known assumptions with long history of analysis.

2 Proposed Algorithm

When we deal with large data, maintaining its integrity only by the hash value of the entire data is not a nice idea. If the entire data is large, it would be nice if one can check integrity of partially received data. For that purpose, hash listing is a basic idea. Data x is split into segments $x_1, x_2, \ldots, x_d$ with appropriate padding, then they are summarized to $(h(x_1), h(x_2), \ldots, h(x_d))$. Each hash value $h(x_i)$ provides integrity of the segment. In this paper we call the list $(h(x_1), h(x_2), \ldots, h(x_d))$ *tag* for x. Unfortunately, attaching a hash list on the segmented data makes no advantage on efficiency of recomputing the tag.

To obtain fast integrity checking algorithm, we design a data integrity checking algorithm without recomputing the tag. Our idea is to use a homomorphic hash function to design a tag and to make batch verification over the tag. A typical and possibly known only practical homomorphic hash function is that is based on a group with hidden order such as in [13] and in [7] as a commitment scheme.

Choose random prime numbers p, q such that $(p-1)/2$ and $(q-1)/2$ are both prime numbers. Set a system parameter $N = pq$ and keep p and q secret to everyone. Each segment x_i is represented as integer by binary notation.

Algorithm 1 described below generates tag.

Algorithm 1 (Tag Generation)
Input : *List of bit-strings* $(x_1, x_2, \ldots, x_d)$,
Output : *Tag* $(y_1, y_2, \ldots, y_d)$.

```
1. For all $i = 1, 2, \ldots, d$ compute $y_i = g^{x_i} \pmod{N}$.
2. Output $(y_1, y_2, \ldots, y_d)$.
```

We suppose the list of bit-strings are obtained with appropriate padding, so the length of each bit-string x_i is set to a given parameter L. dL is supposed to be bounded by a polynomial of k, where k is the security parameter.

Remark 1. Our Tag Generation algorithm is very slow. It is about 1 Mbit/second in our test environment, where we used GMP [9] with patch [8] on Athlon X2 running at 2GHz. The proposed scheme is useful when we repeat verification for many times on an existing tag.

Remark 2. Unlike UMAC [4] and Poly1305 [3], we do not need nonce to generate a tag. Instead, we will need k random bits for Integrity Checking.

Integrity between the data and the tag is checked by the following algorithm. s is a system parameter and is a prime number such that $|s| > k$.

```
**Algorithm 2 (Integrity Checking)**
Input : $(y_1, x_1), (y_2, x_2), \dots, (y_d, x_d) \in \mathbb{Z}/N\mathbb{Z} \times \{0,1\}^L$,
Output : 1 or 0.

1. Choose $e \in [0, s-1]$ randomly. Set $e_i = e^{i-1} \pmod{s}$ for each $i = 1, 2, \dots, d$.
2. Compute $\bar{x} = \sum_i e_i x_i$ in $\mathbb{Z}$.
3. Compute $\bar{y} = \prod_i y_i^{e_i}$ in $\mathbb{Z}/N\mathbb{Z}$.
4. If $g^{\bar{x}} = \bar{y} \pmod{N}$ holds, output 1. Otherwise output 0.
```

The random number e must be unpredictable. We claim that Algorithm 2 is collision-resistant in the sense that it is hard to find a tag T and distinct messages M and M' such that (T, M') is accepted by Algorithm 2 with non-negligible probability while T is the tag for M.

Let V be Algorithm 2. Consider relation R defined by

$$R = \{(y, x) \mid y = g^x \pmod{N}\}.$$

Let FACTOR be the assumption stated in Appendix, the hardness of integer factoring. Then we have,

Theorem 1. *Suppose $(y_1, x_1, x_1^*), \dots, (y_d, x_d, x_d^*)$ be triples of bitstrings generated by a probabilistic polynomial time algorithm and suppose that $(y_1, x_1), \dots, (y_d, x_d) \in R$. If there exists some i such that $x_i \neq x_i^*$, then*

$$\text{Prob}[V((y_1, x_1^*), \dots, (y_d, x_d^*)) = 1]$$

is negligible under FACTOR and Extended Riemann Hypothesis.

Proof is in Appendix.

Remark 3. In Step 2 $\bar{x}$ is computed in $\mathbb{Z}$, unlike in short exponent tests [2] for groups with known order. We claim that Algorithm 2 is still fast if implemented carefully.

Remark 4. It does not seem to be obvious that a short exponent test on a group with known order remains sound if one generate random short exponents e_i by $e_i = e^{i-1} \pmod{s}$, where s is close to 2^k and s is much smaller than the order of the group that is approximately 2^{2k}.

3 Implementation Result

In this section we describe some benchmark result from our implementation of Algorithm 2 on a standard PC.

3.1 Test Environment

We implemented the proposed algorithm on AMD64 processor. Summary of test environment is shown in Table 1. We purchased the system for about 85,000 Yen (about 550 Euro) in 2006. The implementations of SHA-1 and SHA-256 are from OpenSSL [14].

CPU	Athlon64x2 3800+
Clock Frequency	2.0GHz
L1 Cache	16KB / core
L2 Cache	512KB / core
chipset	NVIDIA nForce 410 MCP
main memory	DDR SDRAM 400MHz 512 MB×2
OS	SuSE Linux 10.1 for AMD64
compiler	gcc 4.1.0 -O3
library	gmp 4.2.1

Table 1. Test environment

3.2 How to Measure

We measured throughput of Algorithm 2 for large data on memory generated by Mersenne Twister [10]. The utmost size in our tests will be 400 Gbit, because we have industrial interest in how long it take to check integrity over data in next generation DVDs such as HD DVD and Blu-ray.

Since such large amount of memory is not common at this time we used a 128 MB memory area repeatedly for many times. We suppose it is unlikely that when applying the algorithm on the same memory area there is any significant data remaining in L1 or L2 cache, since 128 MB is large enough compared with the sizes of caches.

3.3 Choice of parameters

N is set to pq, where both p and q are prime number such that $(p-1)/2$ and $(q-1)/2$ are also prime. g is set to 2, since it is guaranteed that the order of 2 in $\mathbb{Z}/N\mathbb{Z}$ is $(p-1)(q-1)/2$. p and q are randomly chosen to be 512-bit length. The security parameter k is set to 80.

3.4 Optimizing Segmentsize

For each target data size, the performance of Algorithm 2 varies on choice of L the size of the segments. Let us fix the size of the entire message and estimate how the time depends on L for each steps in Algorithm 2. Step 1 takes only negligible time. Step 2 takes time in proportion to the target data size, so constant here. Step 3 is proportional to d, so inverse proportional to L. Step 4 takes time in proportion to L. So the required time for Algorithm 2 is estimated as

$$0 + a + \frac{b}{L} + cL$$

for some constants a, b, c. It is likely that there exists an L that attains the fastest.

We measured running time for different Ls, where the size of the target is set to 8Gbit for convenience of the experiments, because the required time is almost linear to the size for large messages. Results are shown in Table 2. In the test environment, the optimal segmentsize is from 768 Kbit to 1024 Kbit. We fix L to 1024 Kbit hereafter.

L(Kbit)	512K	768K	1024K	1280K	1536K
Step 2(sec)	0.66	0.67	0.66	0.67	0.65
Step 3(sec)	1.26	0.83	0.62	0.50	0.42
Step 4(sec)	0.61	0.92	1.22	1.53	1.83
Total(sec)	2.53	2.42	2.50	2.70	2.90

Table 2. L in Kbits and time in seconds

3.5 How Implemented

We implemented Algorithm 2 with C and assembly language. Some ideas for optimization in our implementation are described.

Step 1 We precompute e_i and store on memory. Each e_i is about 80-bit length. It is split into two 40-bit strings, each is stored in its table of 64-bit integers. Let E be a 64-bit register. Algorithm 3 describes the detail.

```
Algorithm 3
Input : 80-bit integer $e$,
Output : Tables of 64-bit integers $E_H[\ ]$ and $E_L[\ ]$

1. Set $E = 1$.
2. For $i = 1, 2, \ldots, d$,
   (a) Set $E = eE \pmod{s}$.
   (b) Set $E_H[i] = E \gg 40$, $E_L[i] = E \ \& \ (2^{40} - 1)$.
```

Step 2 Let l be the least positive integer such that $L < 64l$. A bitstring x with size L is split into 64-bit unsigned integers so as to $x = x[l-1]||x[l]|| \cdots ||x[1]||x[0]$. Let A'_L, A_L, A'_H, A_H be 64-bit registers initialized to be zero. The detailed algorithm for our implementation is shown in Algorithm 4.

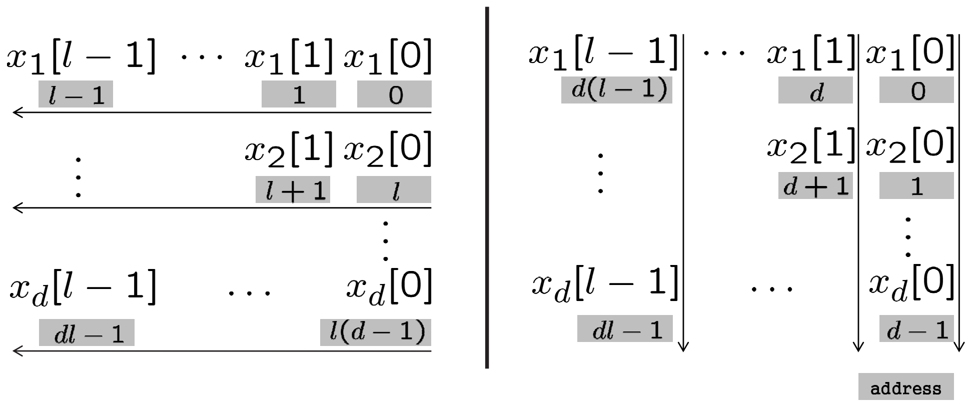
```
Algorithm 4
Input : List of bitstrings $(x_1, x_2, \ldots, x_d)$,
Output : Bitstring $\bar{x}$

1. For $i = 0, 1, \ldots, l-1$,
   (a) For all $j = 1, 2, \ldots, d$,
        i. Set $(A'_L||A_L) = x_j[i] \times_{128} E_L[j] +_{128} (A'_L||A_L)$.
       ii. Set $(A'_H||A_H) = x_j[i] \times_{128} E_H[j] +_{128} (A'_H||A_H)$,
   (b) Set $(A_H||A_L||\bar{x}[i]) = (0||A'_H||A_H) \ll_{192} 40 +_{192} (0||A'_L||A_L)$.
   (c) Set $A'_L = 0$, $A'_H = 0$.
2. Set $\bar{x}[l] = A_L$, $\bar{x}[l+1] = A_H$.
```

$\times_{128}$ is the multiplication of two 64-bit unsigned integers that outputs 128-bit unsigned integer. $+_{128}, +_{192}$ are 128-bit addition and 192-bit addition respectively. $\ll_{192}$ is the left shift over 192-bit integer. We supposed $d < 2^{24}$ so that both Step 1-(a)-i and Step 1-(a)-ii have no overflow.

Remark 5. On computing $\bar{x} = \sum_i e_i x_i$, a simple idea will be to compute the entire sum from the least significant leaf. Suppose for each x_i memory is assigned in the way shown in the left of Figure 1. According to [1], this data structure is not cache friendly. Assignment shown in the right of Figure 1 improves performance.

Step 3 and Step 4 On computing $\bar{y} = \prod_i y_i^{e_i}$, exponentiations are computed once with Algorithm 14.88 in [11], for example. This algorithm is



$x_1[l-1]$ $\cdots$ $x_1[1]$ $x_1[0]$
$l-1$
1
0
$x_2[1]$ $x_2[0]$
$l+1$
l
$x_d[l-1]$ $\cdots$ $x_d[0]$
$dl-1$
$l(d-1)$
$x_1[l-1]$ $\cdots$ $x_1[1]$ $x_1[0]$
$d(l-1)$
d
0
$x_2[1]$ $x_2[0]$
$d+1$
1
$x_d[l-1]$ $\cdots$ $x_d[0]$
$dl-1$
$d-1$
address

Fig. 1. Memory assignments

about 3 times faster than computing each $y_i^{e_i}$ with binary method. We used GMP [9] with patch [8] for the multiplications of multiple precision long integers in Step 3, for exponentiation $g^{\bar{x}}$ (mod N), and for the comparison in Step 4.

3.6 Test Results

We tested the implementation both in single-thread and in multi-thread. The purpose of single-thread test is to compare with hash functions. Multi-thread test is to obtain figure how fast can we check integrity.

Single-Thread Implementation We measured the required time for verifying integrity with the proposed algorithm. The result is shown in Table 3.

data size(Gbit)	100	200	300	400
verify(sec)	16.86	32.36	48.42	63.94

Table 3. The required time on single-thread implementation

The throughput of the tested implementation is 6.26 Gbit/second, that is 3.13 bit/cycle. This is 4.5 times faster than SHA-1, and is 11.3 times faster than SHA-256 in OpenSSL [14].

Multi-Thread Implementation In dual-core environment, $\bar{y}$ and $\bar{x}$ are computed in parallel. We implemented Algorithm 2 with POSIX standard threads. The results are shown in Figure 2, which indicates about 12.5 Gbit/second throughput.

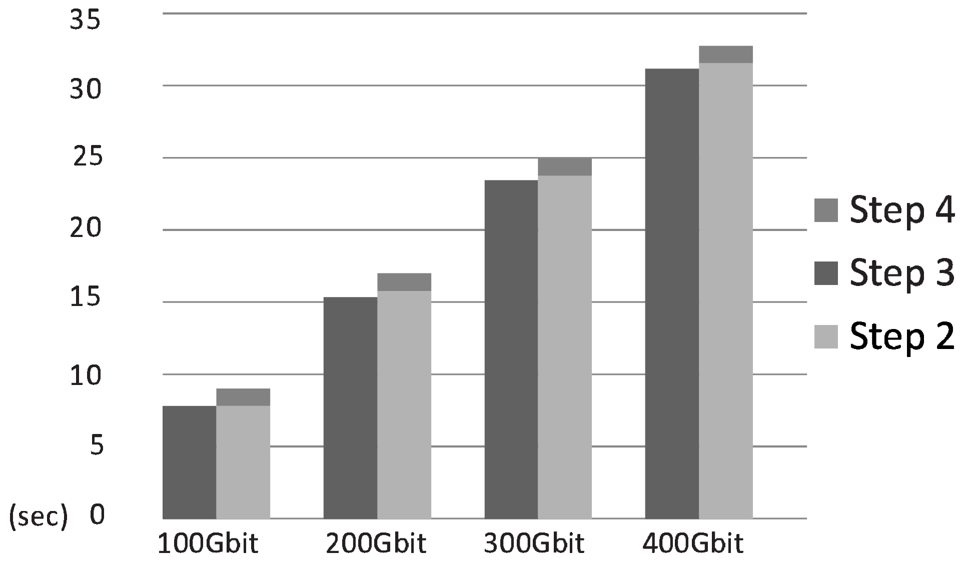


Fig. 2. The required time and data size on multi-thread implementation

3.7 Consideration on Tag Size

Since L is set to 1024 Kbit, the size of the tag is about 0.1% of the entire data, which is about 6 times larger than the tag from hash list by SHA-1. We must allow increase of size by about 0.086% to apply the proposed scheme.

4 Concluding Remarks

We proposed a scheme that provides data integrity that bases on a new design, which we call asymmetric integrity checking. In the proposed scheme, we have two algorithms Tag Generation and Integrity Checking for data integrity.

The experimental results show the proposed Integrity Checking runs at speed over 12 Gbit/second on a standard PC. Our test environment uses DDR-400 memory, which delivers 3.2 Gbytes/second bandwidth per channel, so our implementation result is about half speed of the memory bandwidth. To obtain more speed, it is likely that optimization for memory architecture will matter. We leave further optimization as future work.

The sizes of the tags are proportional to the sizes of the messages, as well as hash lists are. It is open question whether there is a fast algorithm from asymmetric integrity checking with a constant size tag.

References

1. Advanced Micro Devices, Inc. Software optimization guide for the AMD64 processors. `http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/25112.PDF`.
2. Mihir Bellare, Juan A. Garay, and Tal Rabin. Fast batch verification for modular exponentiation and digital signatures. In *EUROCRYPT*, pages 236–250, 1998.
3. Daniel J. Bernstein. The Poly1305-AES message-authentication code. In Henri Gilbert and Helena Handschuh, editors, *Fast software encryption: 12th international workshop, FSE 2005, Paris, France, February 21–23, 2005, revised selected papers*, volume 3557 of *Lecture Notes in Computer Science*, pages 32–49. Springer, 2005. URL: `http://cr.yp.to/papers.html#poly1305`.
4. John Black, Shai Halevi, Hugo Krawczyk, Ted Krovetz, and Phillip Rogaway. UMAC: Fast and secure message authentication. *Lecture Notes in Computer Science*, 1666:216–233, 1999.
5. Koji Chida and Go Yamamoto. Batch processing of interactive proofs. In Masayuki Abe, editor, *CT-RSA*, volume 4377 of *Lecture Notes in Computer Science*, pages 196–207. Springer, 2007.
6. IEEE Standard for Information technology. Telecommunications and information exchange between systems- Local and metropolitan area networks- Specific requirements Part 3: Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications Amendment: Media Access Control (MAC) Parameters, Physical Layers, and Management Parameters for 10 Gb/s Operation. *IEEE Std 802.3ae-2002.*
7. Eiichiro Fujisaki and Tatsuaki Okamoto. Statistical zero knowledge protocols to prove modular polynomial relations. In Burton S. Kaliski Jr., editor, *CRYPTO*, volume 1294 of *Lecture Notes in Computer Science*, pages 16–30. Springer, 1997.
8. Pierrick Gaudry. Assembly support for GMP on AMD64. `http://www.loria.fr/~gaudry/mpn_AMD64/`.
9. The GNU MP Bignum Library. gmp 4.2.1. `http://www.swox.com/gmp/`.
10. Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, 1998.
11. Alfred John Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of applied cryptography.* CRC Press, 1997.
12. Gary L. Miller. Riemann's hypothesis and tests for primality. In *STOC '75: Proceedings of seventh annual ACM symposium on Theory of computing*, pages 234–239, New York, NY, USA, 1975. ACM Press.
13. David Pointcheval. The Composite Discrete Logarithm and Secure Authentication. In Hideki Imai and Yuliang Zheng, editors, *Public Key Cryptography*, volume 1751 of *Lecture Notes in Computer Science*, pages 113–128. Springer, 2000.
14. OpenSSL Project. OpenSSL 0.9.8d.

15. Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding collisions in the full SHA-1. In Victor Shoup, editor, *CRYPTO*, volume 3621 of *Lecture Notes in Computer Science*, pages 17–36. Springer, 2005.
16. Xiaoyun Wang and Hongbo Yu. How to break MD5 and other hash functions. In Ronald Cramer, editor, *Advances in Cryptology — EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 19–35. Springer-Verlag, Berlin, Heidelberg, New York, 2005.

Appendix: Security Proof

In this section collision resistance of Algorithm 2 is shown under the hardness of integer factoring as stated below. Let relation R be defined by $R = \{(y, x) \mid y = g^x \pmod{N}\}$.

Assumption 1 (FACTOR) *Let* $I_k = \{n \mid n = pq, |p| = |q| = k\}$. *Let* $\mathbf{A}$ *be a probabilistic polynomial time algorithm that takes input* N *and outputs* (p, q)*. We define the advantage of* $\mathbf{A}$ *to be* $\mathrm{Adv}(\mathbf{A})(k) = \Pr[N = pq : N \in_U I_k, (p, q) \leftarrow \mathbf{A}(N)]$*. Then we assume that for all* $\mathbf{A}$*,* $\mathrm{Adv}(\mathbf{A})(k)$ *is negligible in* k*.*

Theorem 1. *Suppose* $(y_1, x_1, x_1^*), \ldots, (y_d, x_d, x_d^*)$ *be triples of bitstrings generated by a probabilistic polynomial time algorithm and suppose that* $(y_1, x_1), \ldots, (y_d, x_d) \in R$*. If there exists some* i *such that* $x_i \neq x_i^*$*, then*

$$\mathrm{Prob}[V((y_1, x_1^*), \ldots, (y_d, x_d^*)) = 1]$$

is negligible under FACTOR and Extended Riemann Hypothesis.

Remark 6. One might like to state the soundness of Algorithm 2 in such a way as "if there exists some i such that $(y_i, x_i^*) \notin R$, then $\mathrm{Prob}[V((y_1, x_1^*), \ldots, (y_d, x_d)) = 1]$ is negligible." We remark that Theorem 1 does not claim soundness in that sense.

Theorem 1 claims soundness of the scheme only when (y, x) is generated by a polynomial time algorithm. If (y, x) is generated by some super-polynomial-time algorithm, or using factors of N in an essential way, soundness of Algorithm 2 may not guaranteed.

However, from the practical point of view, it seems reasonable to assume that tagged strings are generated by some polynomial time algorithms, as long as factors of N are secret to everyone. Therfore one should be careful if one generates a tagged string using factors of N for efficiency.

Proof. Let $\eta = \mathrm{Prob}[V((y_1, x_1^*), (y_2, x_2^*), \ldots, (y_d, x_d^*)) = 1]$. We show that η is negligible under FACTOR and ERH.

Let $\tilde{V}$ is the algorithm exactly same as V except that $\tilde{V}$ outputs e in addition if V returns 1. Consider the algorithm described below.

Algorithm 5
Input : $(y_1, x_1), (y_2, x_2), \ldots, (y_d, x_d) \in \mathbb{Z}/N\mathbb{Z} \times \{0,1\}^L$,
Output : *factors of* N.

```
1. Set $\mathbf{e}$ the empty set.
2. Run $\tilde{V}((y_1, x_1^*), (y_2, x_2^*), \ldots, (y_d, x_d^*))$. If the output is $0$ then go
   back to Step 2. Otherwise set $\mathbf{e} = \mathbf{e} \cup \{e\}$.
3. If $\sharp\mathbf{e} < d$ then goto Step 2. Let $\mathbf{e} = \{e_1, e_2, \ldots, e_d\}$.
4. Compute $d \times d$ matrix $E \in Mat(\mathbb{Z})$ defined by $E_{i,j} = e_i^{j-1} \pmod{s}$.
5. For each $i$ compute $h_i = \sum_j E_{i,j}(x_j^* - x_j)$ in $\mathbb{Z}$. If $h_i$ such that
   $h_i \neq 0$ is found, then proceed to Step 6.
6. Run the algorithm described in [12] Lemma 4, where $g(N)$ is set
   to $h_i$.
```

First of all we show that Step 5 always finds some i such that $h_i \neq 0$. Since $\det_{\mathbf{F}_s}(E) = \prod_{i>j}(e_i - e_j)$, we have $\det_{\mathbf{F}_s}(E) \neq 0$ in $\mathbf{F}_s$, so $\det_{\mathbb{Z}}(E) \neq 0$ holds because $\det_{\mathbb{Z}}(E) \pmod{s} = \det_{\mathbf{F}_s}(E)$. Suppose $h_i = 0$ for all i. Then we have $0 = \det_{\mathbb{Z}}(E)(x_i^* - x_i)$ for all i. Since we have some i such that $x_i^* \neq x_i$, this is a contradiction. Thus Step 5 outputs h_i such that $g^{h_i} = 1 \pmod{N}$, where $h_i \neq 0$.

h_i satisfies both $\lambda'(N) \mid h_i$ and $|h_i| = O(|n|^c)$ for some constant c, so the algorithm described in [12] Lemma 4 outputs factors of N in polynomial time under ERH, where $\lambda'(N)$ is as defined in [12].

The expected running time for Algorithm 5 is bounded by $\frac{k^{c'}}{\eta}$ for some constant c'. If η is not a negligible function, then we obtain a probabilistic polynomial time algorithm that output factors of N with probability that is not negligible. ∎

Performance Analysis of Contemporary Light-Weight Block Ciphers on 8-bit Microcontrollers

Sören Rinne, Thomas Eisenbarth, and Christof Paar

Horst Görtz Institute for IT Security
Ruhr University Bochum
44780 Bochum, Germany
`soeren.rinne@rub.de`,{`eisenbarth`,`cpaar`}`@crypto.rub.de`

Abstract. This work presents a performance analysis of software implementations of ciphers that are specially designed for the domain of ubiquitous computing. The analysis focuses on the special properties of embedded devices that need to be taken into account like cost (given by memory consumption) and energy requirements. The discussed ciphers include DESL, HIGHT, SEA, and TEA/XTEA. Assembler implementations of the ciphers for an 8-bit AVR microcontroller platform were analyzed and compared with a byte-oriented AES implementation. While all ciphers fail to outperform AES on the discussed 8-bit platform, TEA/XTEA and SEA at least consume significantly less memory than the AES.

Keywords: microcontroller, software performance, embedded security, ubiquitous computing, SEA, TEA, XTEA, DESL, AES.

1 Motivation

As ubiquitous computing evolves, we have recently seen new ciphers being proposed specifically for the domain of ubiquitous computing. The target of these newly proposed ciphers is not a higher levels of security, as many of these have a shorter key length than the AES. These ciphers rather aim at providing sufficient security in the environment of restricted resources as can be found in many ubiquitous devices. Neither is their focus on a higher maximum performance, but primarily on a smaller footprint, needing less resources such as energy and computing power, and though giving ubiquitous devices a longer lifetime, smaller outline etc. At the same time a security goal of medium to high security is still achievable.

Two of these newly proposed ciphers, namely DESL [17] and HIGHT [11], were designed for a small hardware footprint rather than outstanding software performance. Yet many ubiquitous devices such as MICA Motes [3] and most low cost embedded devices ship with a contemporary low-power 8-bit microcontroller. Hence a software implementation of the cipher might be cheaper after all. Consequently we see a high necessity for a performance analysis of software implementations of these newly proposed light-weight ciphers.

This work features a performance analysis of light-weight ciphers targeted at highly constrained devices. The analysis focuses on the special properties of embedded devices that need to be taken into account like cost (given by memory consumption) and energy needs.

2 Overview of Ciphers

This Section provides a short description of each cipher. An overview of the ciphers' parameters is given in Table 1. Since the parameters of SEA can be chosen, the values that fit our implementation are given in this Table.

Table 1. Characteristic sizes of the focused ciphers

Cipher	AES	DES	DESL	DESX	HIGHT	SEA	TEA	XTEA
Block length [bit]	128	64	64	64	64	96	64	64
Key length [bit]	128	56	56	56	128	96	128	128
Rounds	10	16	16	16	32	94	32	32

The range of the ciphers is quite huge, starting with DESL comprising a 56-bit key providing only medium security. Other ciphers like HIGHT use a 128-bit key to provide high security but use a smaller block size than AES [4] to meet the needs of a restricted environment. Ciphers like SEA [19] are kept flexible in key size so each user may configure it for the security goal and performance needed.

2.1 AES

The Advanced Encryption Standard (AES) [4], also known as Rijndael, is the successor of the Data Encryption Standard (DES). It was announced by National Institute of Standards and Technology (NIST) as a U.S. FIPS in 2001. The cipher developed by J. Daemen and V. Rijmen was the winner of a 5-year standardization process. It has been deployed widely in many crypto applications, being the de-facto standard symmetric block cipher.

AES is a block cipher using an 128 bit block with an 128, 192 or 256 bit key as input. It operates on a 4×4 array of bytes. Each round of AES consists of four stages, namely `AddRoundKey`, `SubBytes`, `ShiftRows`, and `MixColumns`. The AES is known to be quite efficient, especially on 8-bit architectures, owing to its byte-oriented design. Our assembler implementation of the AES is inspired by the AES implementation of B. Gladman [9].

2.2 DES

The Data Encryption Standard (DES) [8] is a cipher selected as an official Federal Information Processing Standard (FIPS) for the United States in 1976. As a block cipher DES operates on blocks with a size of 64 bits. The key also consists of 64 bits; only 56 of these are actually used by the algorithm, the other ones are parity check bits.

The overall structure consists of a so-called Feistel network with 16 identical base rounds with 8 substitution boxes (S-Boxes), an initial permutation, a final permutation, and a separate key schedule. The whole cipher consists only of bit operations, namely shifts, bit-permutations and exclusive-or operations.

DES is not considered as secure anymore. Thanks to Moore's Law, DES can be broken by exhaustive key search in reasonable time. There are several confirmed DES crackers such as the EFF DES Cracker [7] or the COPACOBANA [14]. Furthermore attacks like differential cryptanalysis, linear cryptanalysis, and Davies' attack [2] have been published.

Yet for some applications where security is not as critical, DES and variants of it are still in use.

DESX The block cipher DESX (or DES-X) [13] is an extension to DES to improve some weaknesses of its predecessor. It is defined by $DESX_{K,K_1,K_2}(M) = K_2 \oplus DES_K(M \oplus K_1)$. It was originally suggested by Ron Rivest in 1984 to protect the cipher DES against exhaustive key-search attacks. DESX is said to be substantially more resistant than DES.

DESL Like the above mentioned DESX DESL (DES Lightweight Extension) [17] is an extension to DES to comply with the requirements of small computational devices like RFID devices or Smart Cards. It was suggested by A. Poschmann et al. in 2006 as a new alternative for ultra-low-cost encryption. To decrease chip size requirements it uses only one S-Box repeated eight times. It therefore requires 38% less transistors than the smallest DES implementation published.

2.3 HIGHT

HIGHT is another block cipher proposed by Deukjo Hong et al. [11] which is working on a 64-bit block length and a 128-bit key length. It was proposed to be used for ubiquitous computing devices such as a sensor in USN or a RFID tag at CHES '06 due to its low-resource hardware implementation. Like many of the discussed ciphers, HIGHT makes use of simple operations such as exclusive-or, addition mod 2^8, and bitwise rotation.

The cipher is a variant of generalized Feistel network. It consists of an initial transformation, 32 rounds using 4 subkeys at a time, a final transformation and a key schedule producing 128 subkeys. HIGHTs key schedule algorithm is designed to keep the original value of the master key after generating all whitening keys and all subkeys. Therefore the subkeys are generated on the fly in encryption and decryption.

2.4 SEA

The Scalable Encryption Algorithm ($\text{SEA}_{n,b}$) [19] is designed to be parametric in plaintext/key and processor size. In dependence on given hardware parameters like processor word size, the SEA parameters will be chosen. The main advantages of SEA are efficient combination of encryption/decryption and "on-the-fly" key derivation. It was designed to be an algorithm for small embedded applications like RFID or Smart Cards.

$\text{SEA}_{n,b}$ parameters in our case are plaintext/key size $n = 96$, processor word size $b = 8$, and number of words per Feistel branch $n_b = \frac{n}{2b} = 6$. Therefore we have a suggested number of cipher rounds of $n_r = \frac{3n}{4} + 2 \cdot (n_b + \lfloor b/2 \rfloor) = 92$. As we used the standard implementation provided by the author we have 94 rounds.

The cipher is targeted for processors with a limited instruction set and therefore uses only bit operations such as exclusive-or, word rotation, bit rotation, addition mod 2^b, and a substitution box.

2.5 TEA

The Tiny Encryption Algorithm (TEA) was first presented at the Fast Software Encryption workshop in 1994 by David Wheeler and Roger Needham [21]. The focus of the design was simplicity of description as well as implementation.

TEA is a block cipher operating on 64-bit blocks with a 128-bit key. The Feistel structure is dominated by suggested 64 identical rounds consisting of bit-operations like shifts, addition/substraction mod 2^8 and exclusice-or operations.

A number of revisions of TEA has been designed in order to obliterate some weaknesses of the original version. The revisions of the cipher, Block TEA (often referred to as XTEA) and XXTEA (published in 1998), were needed to secure the cipher.

TEA suffers from equivalent keys - each key is equivalent to three others, which means that the effective key size is only 126 bits. Due to this weakness a method for hacking the Microsoft's Xbox game console, where TEA was used as a hash function, has been developed [18]. The cipher is also vulnerable to a related-key attack which requires 2^{23} chosen plaintexts under a related-key pair, with 2^{32} time complexity [12].

2.6 XTEA

As mentioned before the effective key length of TEA is 126 bits not 128. So in 1996 Needham and Wheeler made two adjustments [15]. The first was to adjust the key schedule and the second was to introduce the key material more slowly. With these adjustments the weaknesses should be repaired and the simplicity is almost retained.

3 Framework Set-Up and Tools

In this Section we will show how the ciphers were implemented for the 8-bit AVR architecture. A short introduction to the software development tools and how to measure clock cycles and throughput are given as well.

3.1 Platform Specification

AVR microprocessors are a family of 8-bit RISC microcontrollers. Their memory is organized as a Harvard architecture with a 16-bit word program memory and an 8-bit word data memory. Due to its easy usage, its low power consumption and its comparatively low price, the AVR microcontrollers have reached a high popularity in embedded system design. Most of the AVR instructions working on the 32 registers are handled in one clock cycle. Reading from the program memory, where our look-up tables are stored, costs 3 clock cycles, whereas reading an writing from an to the Flash memory can be performed in 2 cycles.

Since we aimed to implement the ciphers for ubiquitous devices, we took MICA Motes as an adequate target platform. MICA motes (e.g. MICAz, MICA2, MICA2DOT [3]) are designed for development of wireless sensor networks and use an ATmega128 or ATmega128L microcontroller as CPU. The ATmega128(L) is equipped with 128 kbyte of Flash memory and 4 kbytes of SRAM. Yet our implementations of the ciphers, as presented in Section 4, are also executable on smaller AVR devices comprising less SRAM and Flash memory.

3.2 Porting to the AVR Microcontroller

The authors of a cipher usually provide a reference implementation. We had many different programming languages, e.g. C or Java. In order to reduce the size of the code and to reach a maximum performance on our hardware, all of the ciphers were reimplemented in AVR-Assembly language. We implemented them ourselves except for the AES, which is an implementation of the Chair for Communication Security at the Ruhr-University of Bochum, and SEA, which is an existing implementation in assembly language available at [5]. Other implementations of AES on an AVR platform can be found at [16][10]. For other TEA and XTEA implementations on AVR see [6]. To ensure a fair benchmarking process, we used these reference implementations as a starting point for the assembly implementations. The implementations were neither solely optimized for performance only nor for extremely small code size. Instead we tried to yield a good trade-off between both. For the implementation of SEA we made use of an existing version of the author available at [5]. We only made slight changes with respect to our compiler.

In our performance analysis in Section 4 we will use the AES as a reference implementation for the other ciphers.

3.3 Development Tools

For the software development we used the tool Programmer's Notepad 2 [20]. This is a open source text editor with special features for coders hosted on the Windows platform. Programmer's Notepad 2 contains an automatic makefile execution. It compiles the C program, assembles the assembly language program, links it to an ELF file, and then converts it to a COF file. After this procedure has run without errors we used the output file from Programmer's Notepad 2 to execute and debug the code in AVR Studio 4 [1] and simulate it on an ATmega128 device. AVR Studio 4 is an Integrated Development Environment (IDE) for writing and debugging AVR applications on the Windows platform. The tool provides a full-scale debugger which was used to obtain cycle counts of the execution of the ciphers. Cycle counts were used to benchmark the throughput. Code size was measured using the Programmer's Notepad 2 and GCC compiler.

4 Results

In this Section we present the results of our implementations. The results are compared to an implementation of the AES that was optimized for the 8-bit AVR microcontroller environment as well. The comparison focuses on code size, because memory is an important for size and price of an embedded or ubiquitous device, and on execution time, i.e. throughput, as execution time corresponds to the power consumption of a device.

4.1 Memory Usage

As embedded systems development is strongly price-driven, there are high restrictions in the size of available Flash memory and SRAM. This applies even more to applications like ubiquitous computing or even RFIDs, where power consumption is an important issue, too. The Flash (program) memory of the device is used to store program code and look-up tables, if applicable. The smaller SRAM is used for dynamic access during program execution.

Table 2 shows the memory allocation in flash memory of every cipher. Figure 1 visualizes the results ordered by size.

Table 2. Memory allocation of program code in Flash in bytes

Cipher	AES	DES	DESL	DESX	HIGHT	SEA	TEA	XTEA
Code size	3410	4314	3098	4406	5672	2132	1140	1160

As shown in Figure 1, TEA is the smallest cipher followed by XTEA and SEA. DESL is only slightly smaller than AES. The two implementations of HIGHT are using the highest amount of program memory. Yet all of the ciphers could be run on smaller engines than the used ATmega128.

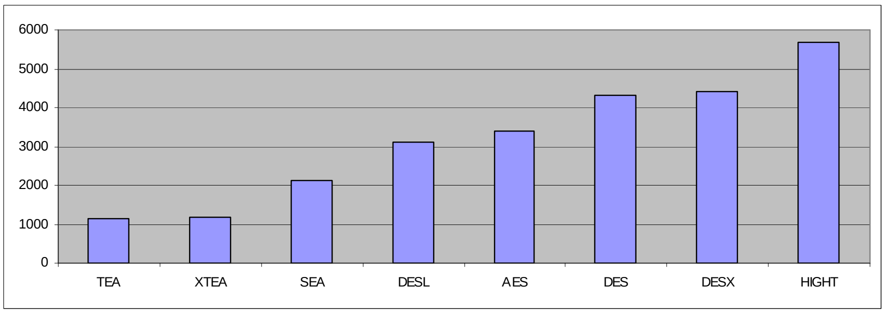


Fig. 1. Code size of ciphers in bytes

4.2 Performance

In the following performance benchmark input and output arrays are of the size of the block size of each cipher. That is to say that we encrypt or decrypt one block with each cipher.

Table 3 shows the number of cycles needed for encryption and decryption for each cipher.

Table 3. Performance of encryption and decryption in measured CPU cycles

Cipher	AES	DES	DESL	DESX	HIGHT	SEA	TEA	XTEA
Encryption	3766	8633	8365	8699	2964	9654	6271	6718
Decryption	4558	8154	7885	8220	2964	9654	6299	6718

Recall that the implementation of HIGHT is the one with the highest use of flash memory. Though in this benchmark it achieves the lowest number of cycles for encryption and decryption of one block of data.

Table 4 and table 5 focus on the throughput of encryption and decryption of each cipher. Column 2 in Table 4 and Table 5 shows the block size in bytes, column 3 replicates the count of cycles from table 3. Column 4 is the quotient of column 3 and 2 and column 5 shows the throughput of encryption/decryption in cycles per byte. The throughput in column 5 is computed assuming the CPU being clocked at 4 MHz.

Figures 2 and 3 reprints the values of Tables 3, 4 and 5 ordered by cycles and respectively by throughput.

Table 4. Throughput of encryption

Cipher	Block Size [bit]	Encryption [cycles]	Encryption [cycles/bit]	Throughput [bit/sec]
AES	128	3766	29.42	135953
DES	64	8633	134.89	29654
DESL	64	8365	130.70	30604
DESX	64	8699	135.92	29429
HIGHT	64	3188	49.81	80301
IDEA	64	2700	42.19	94815
$\text{SEA}_{96,8}$	96	9654	100.56	39776
TEA	64	6271	97.98	40823
XTEA	64	6718	104.97	38107

Table 5. Throughput of decryption

Cipher	Block Size [bit]	Decryption [cycles]	Decryption [cycles/bit]	Throughput [bit/sec]
AES	128	4558	35.61	112330
DES	64	8154	127.41	31396
DESL	64	7886	123.22	32463
DESX	64	8220	128.44	31144
HIGHT	64	3188	49.81	80301
IDEA	64	15393	240.52	16631
$\text{SEA}_{96,8}$	96	9654	100.56	39776
TEA	64	6299	98.42	40641
XTEA	64	6718	104.97	38107

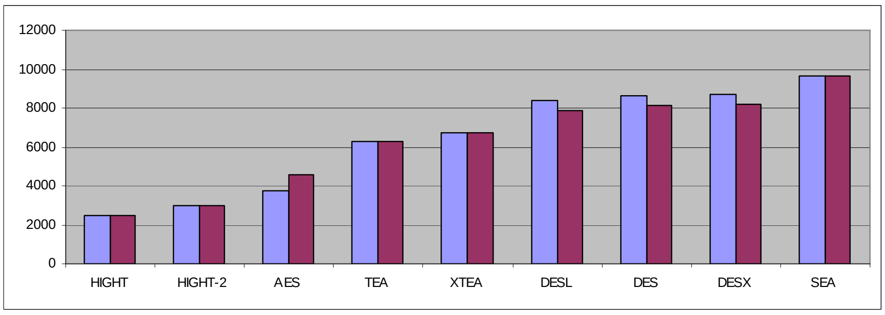


Fig. 2. Cycle count of ciphers

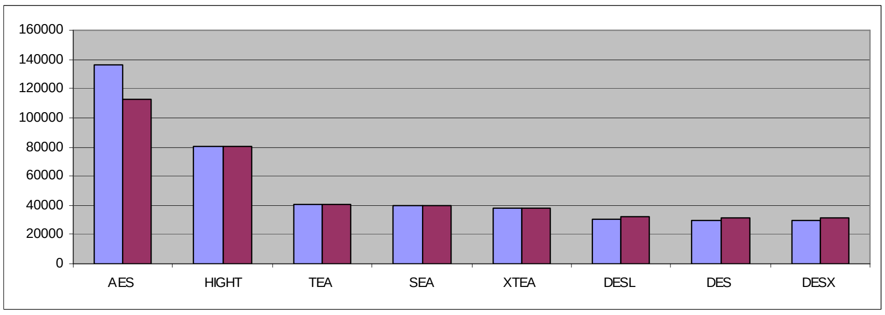


Fig. 3. Throughput of encryption and decryption

Figure 3 shows that the reference AES implementation has a higher throughput than all of the newly proposed ciphers. This is in some cases due to the design objectives of the ciphers. The DES family for example, including DESL, relies on bit permutations which are almost for free in hardware but very expensive in software. This is even true on an 8-bit microcontroller.

4.3 Discussion

Since we did not want to focus solely on code size or on performance, we introduced an additional metric. The ratio of throughput and code size was computed to visualize the combined metric. This metric is given in Figure 4.

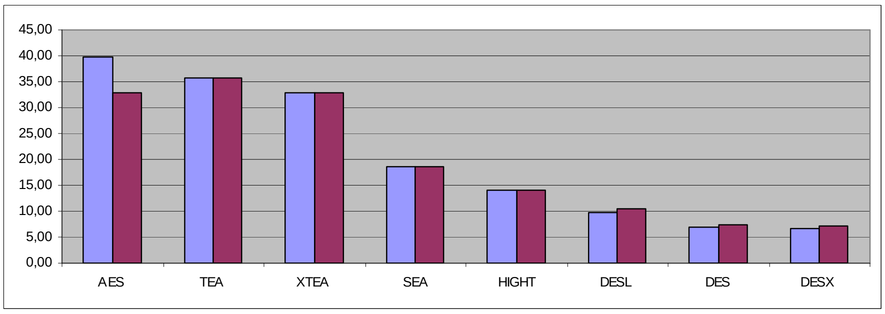


Fig. 4. Throughput-code size ratio of encryption and decryption

Still AES is doing quite well compared to the other ciphers. In this metric the TEA family is at least able to outperform AES in decryption. It can be seen that the ciphers designed for 8-bit software platforms, namely TEA/XTEA and SEA (and AES, of course) outperform the hardware-oriented ciphers HIGHT and the DES family, as expected.

5 Conclusion

We have presented a performance analysis of newly proposed light-weight block ciphers. Target architecture was the 8-bit AVR microcontroller family that can be found in many embedded devices and many applications of ubiquitous computing like wireless sensor networks.

We have shown that many (but not all) of the newly proposed ciphers outperform AES in code size. Especially TEA and XTEA have an extremely small footprint in memory consumption. Yet all of the implemented ciphers were outperformed by the AES in terms of throughput. This might be a disadvantage in wireless devices where computation time means power consumption. The HIGHT was even outperformed by the AES in both, code size and throughput. Though DESL is slightly smaller than AES in code size, it has a worse performance and does not provide comparable security.

As an overall summary one should consider well before using one of these light-weight block ciphers on an 8-bit microcontroller. Only if memory is highly critical, some of the Ciphers might be an alternative to be considered. Usually they just provide a worse performance at comparable or worse security target.

References

[1] Atmel Corporation. Avr studio 4.12, build 498. Available from: http://www.atmel.com/dyn/products/tools_card.asp?tool_id=2725.

[2] E. Biham and A. Biryukov. An Improvement of Davies' Attack on DES. In *Proceedings of EUROCRYPT '94*, pages 461–467. EUROCRYPT '94, 1994.

[3] Crossbow Technology Incl. *MPR-MIB User Manual*. Revision B, June 2006. Available from: http://www.xbow.com/Support/Support_pdf_files/MPR-MIB_Series_Users_Manual.pdf.

[4] J. Daemen and V. Rijmen. *The design of Rijndael, the Advanced Encryption Standard*. Springer-Verlag, 2003.

[5] Efton s.r.o. Implementing SEA on x51 and AVR. Available from: http://www.efton.sk/crypt/sea.htm.

[6] Efton s.r.o. TEA (Tiny Encryption Algoritm) a jeho implementacia v 8051 a AVR[, year = 2007, keywords = TEA, owner = Sren, timestamp = 2007.05.15, url = http://www.efton.sk/crypt/tea_s.htm.

[7] Electronic Frontier Foundation. *Cracking DES*. O'Reilly & Associates, 1998.

[8] Federal Information Processing Standards Publication 46-3. Data encryption standard (des). Technical report, FIPS, 1999.

[9] Brian Gladman. Byte Oriented AES Implementation. Available from: http://fp.gladman.plus.com/AES/.

[10] H.C. Roepke. AVR Implementation of AES. on website. Available from: http://www.christianroepke.de/studium_praktikumB.html.

[11] D. Hong et al. HIGHT: A New Block Cipher Suitable for Low-Resource Device. In *Proceedings of CHES 2006*, 2006.

[12] J. Kelsey et al. Related-key cryptanalysis of 3-WAY, Biham-DES, CAST, DES-X New DES, RC2, and TEA. In *First International Conference on Information and Communication Security*, pages 233–246, 1997.

[13] J. Kilian and P. Rogaway. How to Protect DES Against Exhaustive Key Search (an Analysis of DESX). *Journal of Cryptology*, Volume 14:17–35, 2001.

[14] S. Kumar, C. Paar, J. Pelzl, G. Pfeiffer, and M. Schimmler. Breaking Ciphers with COPACOBANA - A Cost-Optimized Parallel Code Breaker. In *Conference on Special-purpose Hardware for Attacking Cryptographic Systems*, 2006.

[15] R.M. Needham and D.J. Wheeler. Tea extensions. Computer Laboratory, Cambridge, 1997.

[16] B. Poettering. AVRAES: The AES block cipher on AVR controllers. published on website, 2003,2006. Available from: http://point-at-infinity.org/avraes/.

[17] A. Poschmann, G. Leander, K. Schramm, and C. Paar. New Light-Weight DES Variants Suited for RFID Applications. In *Proceedings of FSE 2007*. FSE 2007, 2007.

[18] Matthew D. Russell. Tinyness: An Overview of TEA and Related Ciphers. Draft v0.3, February 2004.

[19] F.X. Standaert, G. Piret, N. Gershenfeld, and J.J. Quisquater. SEA: A Scalable Encryption Algorithm for Small Embedded Applications. Workshop on RFIP and Lightweight Crypto, Graz, Austria, 2005.

[20] Simon Steele. Programmer's Notepad 2, Version v2.0.6.1-ella. Available from: http://www.pnotepad.org/.

[21] D. Wheeler and R. Needham. TEA, a Tiny Encryption Algorithm. In *Lecture Notes in Computer Science*, 1994.

The eSTREAM Project

Matt Robshaw

France Telecom R&D, France

In this presentation we survey the origins, ongoing results, and the goals of the ECRYPT eSTREAM project.

How fast is cryptography?

Daniel J. Bernstein

University of Illinois at Chicago, USA

Users of public-key cryptography have a choice of public-key cryptosystems, including RSA, DSA, ECDSA, and many more. Exactly how fast are these systems? How do the speeds vary among Pentium, PowerPC, etc.? How much network bandwidth do the systems consume?

The eBATS (ECRYPT Benchmarking of Asymmetric Systems) project, joint work with Tanja Lange, aims to answer these questions. I'll present the answers obtained so far, describe the eBATS benchmarking toolkit, and point out future directions in public-key benchmarking. I'll also discuss extensions of the toolkit beyond public-key cryptography, for example measuring the speed of secret-key authenticated encryption and measuring the speed of new hash functions.

The $\mathtt{mp}\mathbb{F}_q$ library and implementing curve-based key exchanges

P. Gaudry and E. Thomé

Abstract

We present a library for finite field arithmetic. The originality of this library lies in the fact that specialized code is automatically produced for the selected finite fields. The opportunity of compile-time optimizations yields substantial performance improvements compared to libraries which initialize the finite field at runtime. This library is used to present benchmarks on some curve-based public key cryptosystems.

1 Introduction

Cryptosystems based on the discrete logarithm problem in (Jacobians of) curves are competitive in many contexts. The main advantage compared to systems based on the factorization problem or on the discrete logarithm problem in finite fields is that the best known algorithm for attacking has exponential time instead of subexponential. In practice it means that for obtaining a given security, the sizes of the parameters are smaller.

The speed of an implementation of a curve-based cryptosystem is mostly given by the speed of the underlying finite field arithmetic. Once the particularities of the finite field implementation are known, one can search in the literature the most suitable choice for the coordinate systems and for the addition chain. This also depends on the amount of memory available, and if resistance to side-channel attacks is required.

In this paper we describe a new (still in development) finite field library called $\mathtt{mp}\mathbb{F}_q$ that we have used to write curve-based cryptosystems. The main objective of this library is speed. Portability, readability, ease of maintenance, ease of use are also wanted, but we accept no feature in the design of the library that would prevent us to apply certain optimizations.

Setting as a goal the implementation of fast curve-based cryptosystems forces a particularity of the underlying finite field operations: The finite field is known at compile-time. This enables a considerable amount of optimizations. The design of the $\mathtt{mp}\mathbb{F}_q$ library is suited to this problem: provide optimized code that takes advantage of all the information that is known at compile-time.

In Section 2 we give an overview of the existing software, and list the requirements for the $\mathtt{mp}\mathbb{F}_q$ library. In Section 3 we explain with more detail the design of $\mathtt{mp}\mathbb{F}_q$ and why certain choices were made. In Section 4 we give some timings. Finally in Section 5 we describe a few BATs that have been implemented with the help of mp$\mathbb{F}_q$. We conclude with some plans for the future.

2 Why yet another finite field library?

2.1 Existing finite field libraries

Obviously, there are several already existing software libraries that can be used to perform computations in finite fields. We briefly review here some of them.

The NTL library [17] by V. Shoup provides arithmetic modulo finite fields, and also goes well beyond that. NTL is written in a small subset of C++, and based on selectable multiprecision arithmetic packages (including the Gnu MP library [12]). NTL has good performance in general and very good performance for small prime fields, using IEEE floating point arithmetic for the reduction step.

The ZEN library [7] by F. Chabaud and R. Lercier is a C library for finite field arithmetic. ZEN handles arbitrary finite field (extension of extensions for instance). Although ZEN is written in ANSI C, it should really be regarded as an object-oriented implementation in the same spirit as X11: Almost every user-land identifier from the high-level interface is a macro that calls a function obtained by dereferencing a pointer in the last passed argument (the field). The high-level interface is well-documented. However, for best performance, if the finite field one is working in is known in advance when writing the code, it is possible to call directly the lower-level functions, thereby saving an indirect function call. ZEN has several lower-level layers, including specialized arithmetic for one-word-long modulus. This only goes to a limited extent, however, since the lower-level interface of ZEN is not documented.

The Miracl library by M. Scott [16] provides an optimized feature set for cryptographical operations. It relies on a finite field layer whose performance appears to be good. Miracl goes well beyond finite field operations, since algorithms such as elliptic curve point counting or algorithms for computing pairings are included.

We also mention the Givaro library by J.-G. Dumas *et al.* [9]. It is written in C++ templates, and claims to perform well for one-word-long modulus.

Besides the libraries cited above, which strive for providing arithmetic for all finite fields, there are also software libraries which focus on particular finite fields. The NuMongo library by R. Avanzi [3] handles specially selected prime fields, with the modulus having a special form. As far as we know, it contains only 32-bit code and is not publicly available.

2.2 The opportunity of compile-time optimizations

As mentioned in the introduction, the primary goal that started the development of mp$\mathbb{F}_q$ was the implementation of fast curve-based cryptosystems. For such an application, the finite field is known in advance. Since speed is desired, one wants to take advantage of this knowledge when building the library. While the performance improvement might seem limited if one has in mind fields modulo 1024-bit primes, this kind of optimization makes a vast difference for small fields.

The contexts with an opportunity for compile-time optimization also includes long-running computations on one or several selected finite fields. In particular, cryptanalysis attempts such as the breaking of the CERTICOM ECC Challenges [6, 13] fall in this category. In the precise example of [13], specially crafted code was written in order to have fast finite field operations. Other settings in which specific code was written include [10, 4].

Several kinds of optimizations are made possible by the knowledge of the finite field at compile time. These include the following list of methods. All these optimizations can easily be performed at compile time, but it is not so easy to do the same at runtime, and in many cases it is impossible.

- Code inlining. At runtime, this is not clear which code to inline, because it might depend on the field.
- Branch elimination. If the values determining the control flow are constant at compile time, then the branches can be avoided. This reduces the cost of loops and conditionals.
- Loop unrolling. Nowadays compilers do it automatically, but the unrolling is more efficient if the length of the loop is a constant.
- Choice of the best algorithm for a given task. This can be done also at runtime (ZEN does so), but is more comfortable at compile time (and comes with zero runtime overhead).

Of the finite field libraries mentioned above, only those focusing on a handful of specialized finite fields (like NuMongo) have the opportunity to take full advantage of the optimizations above. By design, none of the other libraries which provide arithmetic for general finite fields are in position to exploit these optimizations.

It is always conceivable to overload some particular library with a new class for the particular field we are interested in, but no simple mechanism is provided to help the developer in this task.

2.3 Wanted features of $\mathtt{mp}\mathbb{F}_q$

From our experience emerges a need for a software library for finite fields which differs from the existing material. Briefly put, the two main differences are:

- $\mathtt{mp}\mathbb{F}_q$ has to handle very efficiently finite fields that are known at compile time.
- $\mathtt{mp}\mathbb{F}_q$'s specialized code for the selected finite fields should be written automatically, rather than crafted by hand.

$\texttt{mp}\mathbb{F}_q$ sets some goals. The utmost concern is speed, obviously. $\texttt{mp}\mathbb{F}_q$ is not contented with merely working code, since this is not good enough for our claimed purposes. While the automatic process of generating specialized code should take sensible choices, we require that these choices may be overridden easily, or that radically different implementation choices may be taken by the user in a way that is reasonably compatible with the rest of the library. Indeed, there is quite often no single answer to the question "which is *the* fastest implementation of $\mathbb{F}_q$?" for a given value of q. Depending on the intended application, the different finite field operations are not necessarily used equally frequently; in situations where optimizing an operation penalizes another one at the same time, the relevant optimization choices therefore depend on the application.

Since the underlying finite field might be in several cases a parameter of an algorithm, $\texttt{mp}\mathbb{F}_q$ has to provide a consistent application programming interface (API) in order to allow code reuse.

In the last few years, the processors that are available in average workstations have become multi-core. This means that for most applications, having a multi-threaded implementation is a good way to gain efficiency. For this reason, we require that the $\texttt{mp}\mathbb{F}_q$ code be reentrant, so that they are usable within a multi-threaded application.

The dependencies of $\texttt{mp}\mathbb{F}_q$ are free software, $\texttt{mp}\mathbb{F}_q$ itself being free software licensed under the terms of the Gnu Lesser General Public License (LGPL)[1].

3 The design of $\texttt{mp}\mathbb{F}_q$

3.1 Choice of the programming languages

The programming languages used by $\texttt{mp}\mathbb{F}_q$ are `perl`, C, and assembly. The automatic generation of specialized code for a selectable finite field is done by `perl`. From the description of the finite field, the `perl` code creates a C source file and header, which provides the required set of functionalities.

The choice of `perl` calls for some comments. The C++ language offers several methods which could provide the needed genericity. An object-oriented approach with a virtual base class would fail on the speed requirement. The unavoidable indirect function call for virtual methods would be a major performance hit, in particular for the sizes we are focusing on (for small sizes, an inlined implementation of the operation would be most suitable). Furthermore, the variety of fields on which different specialized code is needed would require different classes to be generated by other means anyway.

The template mechanism from C++ provides essentially what is needed. Indeed, this allows a static overriding of functions, leaving the possibility of inlining. We believe however that the kind of syntactic manipulation that is offered by C++ templates can also be obtained by other means. At the expense of losing the type checking of code specializations, we have opted for doing the code generation using `perl`, which is best suited for text manipulation.

[1]By the time of the Speed conference, we plan to distribute a first beta release of $\texttt{mp}\mathbb{F}_q$.

3.1.1 The dilemma of assembly

In many cases, writing critical routines in assembly is required for gaining speed. This is due to strict limitations of the C language concerning arithmetic. For instance, after adding two machine-words, if there is an overflow (a carry), the C language ignores it, whereas on many platforms this information is still available at the assembly level. Also the popular `x86` architecture gives access to the full double-word result of a multiplication, whereas only the low significant word is reachable from C. A third example is assembly instructions specific to some platform, like for instance the SSE-2 instructions set [1]. Such instructions and the corresponding data types are not accessible with standard C.

There are several approaches to writing assembly. It is possible to write a standalone function, that respects the application binary interface (ABI) of the C compiler/system we use. This is very convenient but it means that a possibly costly function call has to be payed each and every time one needs it. The other approach is not standard, but available in many compilers, including the Gnu C compiler (GCC). It consists in inline assembly language insertion using the `asm()` keyword. The programmer must tell the compiler which registers are used for input, ouput and temporary usage during the assembly stage. This has the advantage of avoiding the function call. Another option is language extensions which are specific to some compilers. Several compilers (including GCC as well as compilers from Intel, Microsoft) provide the `emmintrin.h` include file, with for instance the `_mm_slli_epi64` macro which corresponds to the `psrlq` assembly instruction (which right-shifts a 128-bit SSE-2 register).

Deciding between standalone functions and inline assembly language is a matter of length of the assembly code, but not only. One could think that writing a C function as a list of small blocks of `asm()` interleaved with pure C code is a good idea, but sometimes it appears that it is better to write the whole function in assembly to help the compiler in register allocation and handling the data flow. Unfortunately, right now, we have not yet been able to find a strict rule for when an approach should be chosen or another, and trying several implementations is the only resort.

3.2 The API and function naming

The API of $\mathtt{mp}\mathbb{F}_q$ is as follows: Let `TAG` be a mnemonic that corresponds to a finite field or a family of finite fields. In fact, `TAG` will also be different if the internal representation of elements of the same finite field are different. For instance, `2_27` can be the mnemonic for the finite field with 2^{27} elements in classical (polynomial basis) representation; or `p_mgy_3` can be the mnemonic for the family of prime finite fields where the modulus fits in 3 machine-words and the elements are in Montgomery representation. Then the C types and the C functions corresponding to this mnemonic start with `mpfq_TAG`. For instance, an element of $\mathbb{F}_{2^{27}}$ will have a type `mpfq_2_27_elt`, and the multiplication function between two such elements will be called `mpfq_2_27_mul`.

3.3 Macros and inline functions

All these types and functions are stored in two files: `mpfq_TAG.c` and `mpfq_TAG.h`. The speed requirement means that all the functions that take less than (say) a few hundred cycles must be inlined. Instead of using the C preprocessor for defining macros, mp$\mathbb{F}_q$ is developed using `static inline` C functions that are included in the `.h` file. The inlining effect is the same, but compared to a macro, this has the advantage to allow the compiler to perform a type checking and to facilitate the debugging (when switching off the inlining optimisation options of the compiler). For instance, in $\mathbb{F}_{2^{89}}$ the `add` function will be as follows:

```
static inline mpfq_2_89_add(mpfq_2_89_field_ptr K,
  mpfq_2_89_dst_elt r, mpfq_2_89_src_elt s1, mpfq_2_89_src_elt s2)
{
    r[0] = s1[0]^s2[0];
    r[1] = s1[1]^s2[1];
}
```

This example calls for additional remarks:

- The first argument of most mp$\mathbb{F}_q$ functions is a pointer to the finite field in which the operation takes place. For most implementations, it is not used, but in some case it is convenient. For instance if the implementation of the multiplication covers all prime fields of a given size, then the modulus should be accessible to the function, and is then obtained from the first argument.
- The type for an element is split into two subtypes marked `src` and `dst`. This follows GMP practice to distinguish between `const` and variable arguments. Adding the keyword `const` to a variable sometimes helps the compiler to choose the right optimization.

3.4 Code generation

It should be now evident that there will be a lot of redundancy between different `.c` and `.h` files of mp$\mathbb{F}_q$. To avoid the problems with maintaining a code with a lot of code duplication, we have chosen to have most of the C and assembly sources of mp$\mathbb{F}_q$ generated automatically by `perl` scripts. The power of `perl` with manipulating files, strings, regular expressions makes it a very nice alternative to any macro-based preprocessing (like CPP or M4) or to a template-based C++ approach.

We give an example of the power of this approach: when writing the code for the trace of an element in a specific finite field of characteristic 2 in polynomial basis representation, one wants to precompute the powers of the defining element x that have trace 1, in order to create the mask. If you do not allow a powerful enough language for the preprocessing, this precomputation will have to be stored and shipped with the library sources, whereas a `perl` script has no problem to do this precomputation on the fly, just before creating the appropriate C function. It is even possible, if some precomputations would be tedious in `perl`, to hand off some of the work to an external program in C ($\mathtt{mp}\mathbb{F}_q$ uses such a convenience).

We have implemented a main `perl` module that helps in the organization of the code generation. Several `perl` scripts in $\mathtt{mp}\mathbb{F}_q$ generate code, but they are not handling the global organization themselves. Instead, the API for $\mathtt{mp}\mathbb{F}_q$ for all finite fields is concentrated in a single file `api.pl` that lists the functions that should be present, together with their prototypes. The generation of the `.c` and `.h` files for a given mnemonic `TAG` is done by the main module, which iterates over the functions in the API. It delegates the generation of these functions to the specialized scripts, fetching `perl` subroutines named `code_for_<function_to_generate>`. At the end the main module reconstructs both files from all the codes, and creates the appropriate prototypes. This approach allows to enforce conformance to the API.

4 Benchmarks

4.1 Development status of $\mathtt{mp}\mathbb{F}_q$

The API and the main `perl` module of $\mathtt{mp}\mathbb{F}_q$ are more or less fixed. $\mathtt{mp}\mathbb{F}_q$ will probably gain more functionalities gradually, and most importantly we need to improve speed at every possible level. Until now we have focused on 64-bit architectures based on the AMD64 instruction set. This covers essentially all the processors currently sold for personal computers and workstations (Athlon64, Opteron, Core2, recent Xeon). We believe that in a near future, 32-bit architectures will be found only in embedded systems. Our code works on 32-bit architectures but the assembly support is inexistent or very poorly written. Furthermore, on many 32-bit architectures, the floating-point unit is more powerful than the integer unit, so that this would probably give the best performance, and we didn't implement this.

Apart from the target architecture, we have been concentrating in optimizing the finite fields that are needed for our BATs. In particular, for a prime field modulus which is not sparse, the Montgomery representation should probably be used, but this is not yet properly set in $\mathtt{mp}\mathbb{F}_q$, so that the benchmarks are somewhat deceiving[2].

While $\mathtt{mp}\mathbb{F}_q$ has good performance for some operations, some other are in dire need for improvement (finite field inversion notably).

4.2 Benchmarking methodology

We start with a word of warning: measuring the cost of a small operation is essentially impossible on modern computers. Assume that an operation takes 20 cycles; assume also that this operation is implemented in an inlined function (in C or assembly, this does not really matter for this discussion). The cost of setting up the data and preparing the

[2]Montgomery representation arithmetic will probably be in place by the time of the Speed conference.

registers at the beginning might take a few cycles (say 4 cycles), and this task is done by the compiler. However, those 4 cycles might become much less if the context of the function call is favorable (that's one of the advantages of inlining). Therefore, some discrepancy is inherently attached to the measurement.

There are basically two ways of timing an operation: either ask the operating system (with the `getrusage()` function) or use the tick counter of the processor (the `rdtsc` assembly instruction on `x86`). The first approach is fine only for very long tasks, since the precision is of the order of the millisecond. The second approach is suitable only for rather short tasks, since any interruption or context switch of the system will perturbate the measure. Also, there is some kind of "Observer effect" for very small operations: a call to `rdtsc` is not serializing, which means that there is no guarantee that the instructions are executed in the order they are written. This is of course not good for our purpose. There is a variant of `rdtsc` that is serializing (or one can add some serializing operation before and calibrate it), but then we really perturbate the operation we are measuring, since there is a high risk of flushing the pipeline.

In our context, we have mostly used the `getrusage` approach for our measures. Since the operations we want to measure are small, we repeat them a large number of times and divide the running time accordingly. On a few tests we have made, the results are not too far from the other approach based on `rdtsc`, and consistent with the running times of the BATs we have built upon the measured operations. But an operation like an addition in $\mathbb{F}_{2^{113}}$ can definitely not be measured in an optimized implementation, since its cost is essentially just the cost of fetching the appropriate data: if it is already in registers, then the operation will be less than 2 cycles or even zero (if the `xor`'s can be inserted between higher latency instructions), but if this operation must be done at a time where all the registers are already occupied, moving data between the stack and the registers can cost a non-negligible time.

4.3 Cost of basic operations for prime fields

For prime fields, we have written $\mathtt{mp}\mathbb{F}_q$ implementations for each machine word size of the modulus (up to nine words) and for the two finite fields that we use in our cryptographic applications. The algorithms we have implemented are by no means original (classical representation and we have used a basic binary extended GCD for the inversion). The costs of basic operations for these fields are given in Table 1 and Table 2. We also give similar benchmarks in NTL and ZEN for comparison. NTL and ZEN do not take advantage of the pseudo-Mersenne form of the modulus. However, in ZEN one can activate a Montgomery representation that speeds-up computations, so we give both timings.

We can see that $\mathtt{mp}\mathbb{F}_q$ is faster than NTL for all sizes. ZEN with Montgomery representation is comparable or slower than NTL except for 1 word primes, where it is sometimes faster than $\mathtt{mp}\mathbb{F}_q$. As one can expect, the difference is more visible for small sizes, and on Opteron, since our assembly code is best suited to this processor. The gain obtained by writing a reduction procedure that is specific to a pseudo-Mersenne modulus is visible on the last two columns.

We conclude with a comment on the usual practice in curve-based cryptography: quite often, to compare the costs of different coordinate systems or addition chains, only the multiplications and squarings are counted, and the additions are said to be negligible. This is clearly not the case, for instance for the field $\mathbb{F}_{2^{255}-19}$ on the Opteron where the `mul/add` ratio is less than 6. The same kind of ratio is observed with the ZEN and NTL libraries, and is even amplified by the fact that an addition or subtraction is much slower than with $\mathtt{mp}\mathbb{F}_q$.

Table 1: Time (in nanoseconds, with 2 significant digits) for basic operations in $\mathbb{F}_p$ on an **AMD Opteron** 250 processor at 2.40 GHz.

$\mathtt{mp}\mathbb{F}_q$:

	1 word	2 words	3 words	4 words	$2^{127}-735$	$2^{255}-19$
add	2	4	5	7	4	8
sub	2	3	5	5	4	9
sqr	67	108	170	230	14	30
mul	66	109	180	240	16	45
inv	420	2600	4600	7500	2600	7400

NTL:

	1 word	2 words	3 words	4 words
add	40	42	36	47
sub	38	40	28	44
sqr	120	150	230	290
mul	120	150	230	290
inv	1600	4400	6600	9200

ZEN/ZENmgy:

	1 word	2 words	3 words	4 words
add	8/11	44/44	44/44	48/49
sub	7/8	64/71	66/70	73/75
sqr	62/90	270/170	420/270	520/320
mul	68/95	300/180	450/270	600/340
inv	1700/2100	3300/4300	4800/5900	6500/7500

4.4 Cost of basic operations for binary fields

The binary fields up to $\mathbb{F}_{2^{255}}$ are implemented in $\mathtt{mp}\mathbb{F}_q$, using a polynomial basis representation. We have chosen defining polynomials with lowest possible Hamming weight. Figure 1 shows the performance of the multiplication, squaring and inversion using $\mathtt{mp}\mathbb{F}_q$ compared to the NTL and ZEN libraries for these fields. The figure also indicates the timings for the "unreduced" multiplication and squaring operations. The graphs on the left side correspond to timings on an AMD Opteron CPU at 2.40 GHz, while the graphs on the right

Table 2: Time (in nanoseconds, with 2 significant digits) for basic operations in $\mathbb{F}_p$ on an **Intel Core2** 6700 processor at 2.66 GHz.

mp$\mathbb{F}_q$:

	1 word	2 words	3 words	4 words	$2^{127} - 735$	$2^{255} - 19$
add	1	2	4	8	3	8
sub	1	4	5	7	3	9
sqr	73	110	180	240	17	40
mul	74	120	190	260	19	53
inv	300	2000	3600	5800	2000	5800

NTL:

	1 word	2 words	3 words	4 words
add	38	45	53	67
sub	38	45	52	64
sqr	110	130	210	270
mul	110	140	210	270
inv	1200	3400	5800	8000

ZEN/ZENmgy:

	1 word	2 words	3 words	4 words
add	6/6	41/41	46/46	57/57
sub	4/4	54/60	60/62	73/78
sqr	52/52	280/120	400/170	550/250
mul	52/60	280/120	400/180	590/260
inv	1000/1000	2500/3000	3800/4300	5000/5900

side correspond to timings on an Intel Core2 CPU at 2.66 GHz.

The algorithms implemented for the different operations are the classical ones described for instance in [8, chap. 11]. So far, the multiplication is done using the schoolbook algorithm, but Karatsuba and Toom-Cook variants have to be measured in comparison. It appears that for all sizes, the best performance for the multiplication is attained by using the SSE-2 instruction set [1]. Using these instructions, it is effectively possible to work in parallel with two 64-bit machine words at a time. The performance gain is most remarkable on the Intel Core2 CPU.

The comparison with the NTL library shows that mp$\mathbb{F}_q$ is faster than NTL except in a few situations. The ZEN library is somewhat slower than NTL in particular for the inversion (we did not investigate where this problem could come from). In our ZEN test program, we have activated the precomputations that could yield speedups, but we have not tried to split the extension in a double extension; in the documentation of ZENfact (a submodule of ZEN), there are examples of such constructions that provide a speedup, but this is not really automatic. We have also skipped the optimization that builds a logarithm table, since this is valid only for tiny fields.

The inversion in $\mathtt{mp}\mathbb{F}_q$ has not been looked at seriously, it merely has the merit of giving correct results. Concerning the multiplication, the relative under-performance of the SSE-2 implementation on the AMD Opteron CPU is probably explained by the different implementation of the SSE-2 pipeline on this particular CPU compared to the Intel Core2. On both CPUs, the large steps around 2^{250} call for further optimization, and will be investigated. For this purpose, an automatic tuning program is being prepared. We mention that NTL has a conspicuous problem for finite fields smaller than $\mathbb{F}_{2^{64}}$. This should probably not be worried about and should be considered as an easy tuning issue.

5 Writing BATs with $\mathtt{mp}\mathbb{F}_q$

We have used $\mathtt{mp}\mathbb{F}_q$ to write efficient software implementation of the Diffie-Hellman key exchange protocol based on curves. We started with the `curve25519` parameters given by Bernstein [4]: this is an elliptic curve in Montgomery form defined over $\mathbb{F}_{2^{255}-19}$, such that both the curve and the twist are secure. We obtain the following timings on our two test machines.

`curve25519`		
	Opteron 2.40 GHz	Core2 2.66 GHz
Time for one scalar mult. in μsecs	128	145
Time for one scalar mult. in cycles	307,000	386,000
Number of scalar mul. per second	7800	6900

We have designed a cryptosystem of genus 2 of the same level of security that we called `surf127eps`. It is based on a genus 2 curve defined over $\mathbb{F}_{2^{127}-735}$ that has complex multiplication by $K = \mathbb{Q}\left(i\sqrt{5+\sqrt{53}}\right)$. The Jacobian of this curve has an order which is 16 times a prime and is suitable for using the Kummer surface formulae of [11] that we have implemented. We obtain the following timings on our two test machines.

`surf127eps`		
	Opteron 2.40 GHz	Core2 2.66 GHz
Time for one scalar mult. in μsecs	116	154
Time for one scalar mult. in cycles	279,000	410,000
Number of scalar mul. per second	8600	6500

We can see that for prime fields, the Opteron behaves better than the Core2. This might be surprising since this Opteron is a 3-year old computer, whereas the Core2 is a brand new architecture; however our skills to optimize assembly code for the Core2 is clearly not the same as for Opteron.

The other observation is that a genus 2 cryptosystem can beat an elliptic one with our implementation, depending on the processor. However, if a general, efficient genus 2 point counting implementation were available, one could construct a Kummer surface with small coefficients, thus saving a lot of operations (as shown by Bernstein [5]). We expect that the situation would be constantly in favour of genus 2.

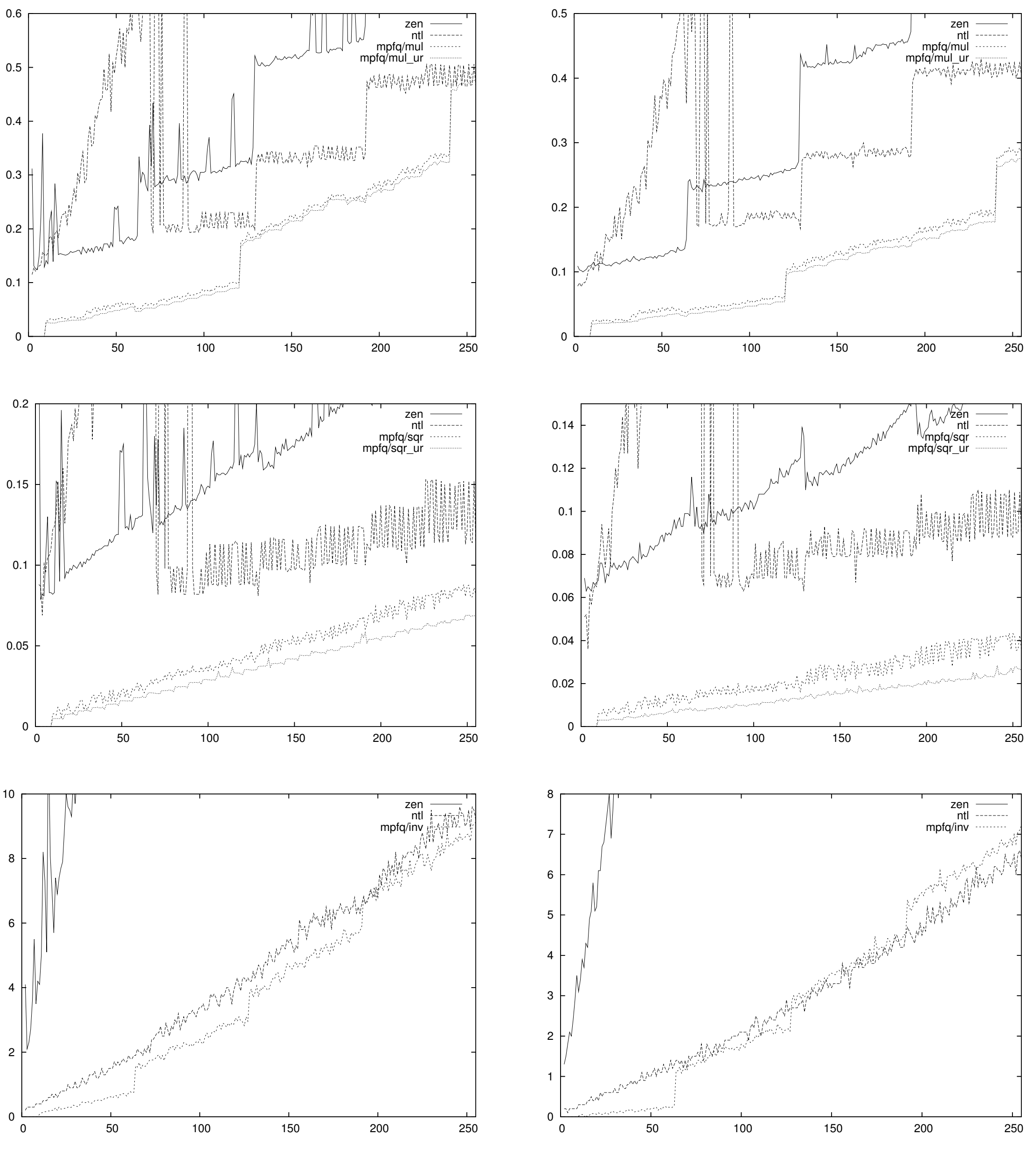


Figure 1: Time (in microseconds) for Multiplication, Squaring, Inversion in $\mathbb{F}_{2^n}$
(Left column is on AMD Opteron 2.40 GHz, right column is on Intel Core 2 2.66 GHz)

In order to test our library and to measure the difference between prime fields and characteristic 2 curve based cryptosystems, we have also implemented the scalar multiplication on elliptic curves in characteristic 2, based on the formulae by Stam [18], over the finite field $\mathbb{F}_{2^{251}}$ and a genus 2 scalar multiplication based on the Kummer surface, over the finite field $\mathbb{F}_{2^{113}}$. This time there is no problem for the point counting in genus 2, thanks to p-adic methods (we have used Magma for these point counting computations).

The results are the following. It should be noted that the performance suffers from the lack of fine-tuning of the multiplication algorithm (in particular, we acknowledge that the multiplication in $\mathbb{F}_{2^{251}}$ is still sub-optimal).

`curve2_251`		
	Opteron 2.40 GHz	Core2 2.66 GHz
Time for one scalar mult. in μsecs	863	506
Time for one scalar mult. in cycles	2,070,000	1,350,000
Number of scalar mul. per second	1100	2000

`surf2_113`		
	Opteron 2.40 GHz	Core2 2.66 GHz
Time for one scalar mult. in μsecs	441	268
Time for one scalar mult. in cycles	1,060,000	713,000
Number of scalar mul. per second	2200	3700

Comparison with other scalar multiplication implementations

The usual problem when comparing timings is that the computers are quickly evolving, so that comparison is difficult. This is particularly true in the present case, where we are concentrating only on 64-bit architectures, whereas almost all implementations reported in the literature rely on 32-bit architecture. This is strange, since the Opteron has been sold for 4 years, now, and the gain of using 64-bit is clear.

Additionally, usually the reported implementations are there to illustrate some improvement in the group law formulae, so that no two papers are really comparable if one is mostly interested in the underlying finite field implementation. Therefore we give the raw data, without trying to scale it to our experiment platform or to the coordinate system we choose.

For each reference, we mention the result that corresponds more or less to the security level we have chosen.

In [2], an implementation of curve arithmetic in characteristic 2 has been written, based on a carefully written set of finite field routines. There timings are given on a 32-bit Power G4 at 1.5 GHz. In genus 1 over $\mathbb{F}_{2^{251}}$, a scalar multiplication takes 3758 microseconds. In genus 2 over $\mathbb{F}_{2^{109}}$, a scalar multiplication takes 1673 microseconds.

In [4] where `curve25519` is described, Bernstein reports an implementation between 620000 and 950000 cycles depending on the processor. All the processors that are considered are 32-bit, and therefore floating point arithmetic used. In [5], he gives a genus 2 implementation (very similar to `surf127eps`) that takes 580000 cycles on a Pentium M.

In [19], an implementation in characteristic 2 gives the following timings on a Pentium 4 at 1.8 GHz: in genus 1 over $\mathbb{F}_{2^{191}}$, a scalar multiplication takes 2780 microseconds and in genus 2 over $\mathbb{F}_{2^{95}}$, it takes 3410 microseconds.

In [3], Avanzi has used his NuMongo library to implement scalar multiplication for curves over prime fields. The timings are for an Athlon 1 GHz: in genus 1 over a 256-bit prime field, it takes 3048 microseconds and in genus 2 over a 128-bit prime field, it takes 3575 microseconds.

6 Future plans

The future directions of $\mathtt{mp}\mathbb{F}_q$ are numerous, given the amount of algorithms that would be worth giving a try. For prime fields, we need to adapt our implementation of Montgomery's REDC algorithm [15] to the $\mathtt{mp}\mathbb{F}_q$ library. We also plan to improve the inversion both on prime and binary fields.

Given the growing interest in pairing-based cryptography, we will probably provide implementations of extension field arithmetic (with specialized code for the most frequently used extension degrees).

It is planned to extend $\mathtt{mp}\mathbb{F}_q$ to handle polynomials and matrices over finite fields and generate optimized source code files for this purpose. This might lead us to consider the case of large polynomials, and include FFT algorithms which are suited to the base fields used.

As for the BATs, we still have room for improvements in the choice of the addition chain. Right now, we have used the most basic binary ladder. We plan to try some of the heuristic algorithms available in the literature for finding better addition chains, for instance the so-called PRAC algorithm by Montgomery [14].

Acknowledgments

Although we are only two authors, we rely heavily on GMP, not only as a dependency library, but also as a source of inspiration for the design of $\mathtt{mp}\mathbb{F}_q$. We have also taken ideas from ZEN, NTL and from software written by R. Harley for the ECDL challenges. We wish to thank Paul Zimmermann and Richard Brent who shared several ideas with us on the topic of muliplication in binary fields.

The genus 2 curve that we use for the BAT called Surf127-735 has been generated by the CM method using tools written by T. Houtmann.

References

[1] Advances Micro Devices. *AMD64 Architecture Programmer's Manual, Volume 4: 128-Bit Media Instructions*, 2005.

[2] R. Avanzi, N. Thériault, and Z. Wang. Rethinking low genus hyperelliptic jacobian arithmetic over binary fields: interplay of field arithmetic and explicit formulae, 2006. Preprint available at `http://www.cacr.math.uwaterloo.ca/techreports/2006/cacr2006-07.pdf`.

[3] R. M. Avanzi. Aspects of hyperelliptic curves over large prime fields in software implementations. In M. Joye and J.-J. Quisquater, editors, *CHES 2004*, volume 3156 of *Lecture Notes in Comput. Sci.*, pages 148–162. Springer–Verlag, 2004. Proc. 6th International Workshop on Cryptographic Hardware and Embedded Systems, Cambridge, MA, USA, August 11-13, 2004.

[4] D. J. Bernstein. Curve25519: new diffie-hellman speed records. In M. Yung, Y. Dodis, A. Kiayias, and T. Malkin, editors, *Public Key Cryptography – PKC 2006*, volume 3958 of *Lecture Notes in Comput. Sci.*, pages 207–228. Springer–Verlag, 2006. Proc. 9th International Conference on Theory and Practice of Public-Key Cryptography, New York, NY, USA, April 24-26, 2006.

[5] D. J. Bernstein. Elliptic vs. hyperelliptic, part 1, 2006. Talk given at ECC 2006. Slides available at `http://cr.yp.to/talks.html#2006.09.20`.

[6] Certicom corp. The Certicom ECC challenges, 1997. Description at `http://www.certicom.com/index.php?action=ecc,ecc_challenge`.

[7] F. Chabaud and R. Lercier. ZEN, user manual, 1996–2007. Homepage at `http://zenfact.sourceforge.net/`.

[8] H. Cohen and G. Frey, editors. *Handbook of elliptic and hyperelliptic curve cryptography.* Chapman & Hall / CRC, 2005.

[9] J.-G. Dumas, T. Gautier, P. Giorgi, J.-L. Roch, and G. Villard. Givaro, une bibliothèque C++ pour le calcul formel, 1987–2007. Homepage at `http://ljk.imag.fr/CASYS/LOGICIELS/givaro/`.

[10] M. Fouquet, P. Gaudry, and R. Harley. Finding secure curves with the Satoh-FGH algorithm and an early-abort strategy. In B. Pfitzmann, editor, *Advances in Cryptology – EUROCRYPT 2001*, volume 2045 of *Lecture Notes in Comput. Sci.*, pages 14–29. Springer-Verlag, 2001.

[11] P. Gaudry. Fast genus 2 arithmetic based on Theta functions. *J. of Mathematical Cryptology*, 2007. To appear. Preprint available at `http://eprint.iacr.org/2005/314`.

[12] T. Granlund. `GMP`, the GNU multiple precision arithmetic library, 1993–2007. Homepage at `http://gmplib.org/`.

[13] R. Harley. The ECDL project. `http://cristal.inria.fr/~harley/ecdl/`, 2000.

[14] P. L. Montgomery. Evaluating recurrences of form $x_{m+n} = f(x_m, x_n, xm - n)$ via Lucas chains, 1983. Preprint available at `ftp.cwi.nl:/pub/pmontgom/Lucas.ps.gz`.

[15] P. L. Montgomery. Modular multiplication without trial division. *Math. Comp.*, 44(170):519–521, Apr. 1985.

[16] M. Scott. MIRACL: Multiprecision integer and rational arithmetic c/c++ library, 1988–2007. Homepage at `http://www.shamus.ie/`.

[17] V. Shoup. NTL: A library for doing number theory, 1990–2007. Homepage at `http://www.shoup.net/ntl/`.

[18] M. Stam. On Montgomery-like representations for elliptic curves over $GF(2^k)$. In Y. G. Desmedt, editor, *Public Key Cryptography – PKC 2003*, volume 2567 of *Lecture Notes in Comput. Sci.*, pages 240–254. Springer–Verlag, 2003.

[19] T. Wollinger, J. Pelzl, and C. Paar. Cantor versus Harley: Optimization and analysis of explicit formulae for hyperelliptic curve cryptosystems. *IEEE Trans. Comput.*, 54:861–872, 2005.

Accelerating SSL using the Vector processors in IBM's Cell Broadband Engine for Sony's Playstation 3™

Neil Costigan[*], Michael Scott
School of Computing,
Dublin City University,
Dublin 9, Ireland.
{neil.costigan,mike}@computing.dcu.ie

Abstract

Recently the major performance chip manufacturers have turned to multi-core technology as a more cost effective alternative to ever increasing clock speeds. Well known examples of multi-core architectures include the Intel Core Duo and AMD Athlon 64 X2 range of chips. IBM have introduced the Cell Broadband Engine (Cell) as their next generation CPU to feed the insatiable appetite modern multimedia and number crunching applications have for processing power.

The Cell is the "Wicked Smart"[1] technology at the heart of Sony's Playstation 3™. The Cell contains a number of specialist synergistic processor units (SPUs) optimised for multimedia processing and offer a rich programming interface to applications that can make use of the vector processing capabilities. The specialised hardware design for gaming will always deliver performance gains compared to a more generic processor for its specific domain. Multi-precision number manipulation for use in cryptography is a considerable distance away from this domain. This paper explores the implementation and performance gains when using the vector processing capabilities for SSL and shows that big improvements are still possible with the hardware designed primarily for other purposes.

1 Why SSL?

Despite huge gains in computing performance and bandwidth, the widespread use of secure communications on the Internet is still essentially limited to SSL connections for password logins or with credit card payments. Despite this SSL implementations are widely distributed and well analysed for security weaknesses making it the de-facto standard for secure communications. The main reason for such limited usage is the perception that encrypted communication protocols such as SSL place too high demands on bandwidth and processing power at the server side of the communication and can interrupt the browsing experience of the client. This paper sets out to show that with the performance of modern multi-core hardware devices it is now possible to enable secure channels for a wider range of network communications.

2 The Cell Broadband Engine

When Sony examined the options for the Playstation 2's successor they realised that traditional clock speed improvements were not going to deliver to next generation demands. They wanted something more than the

[*]Research supported by the Irish Research Council for Science, Engineering and Technology, (IRCSET).

[1]"Wicked Smart" is an advertising slogan used by Sony

traditional CPU if the Playstation brand was going to maintain its lead over its chief competitor Microsoft's XBox^{TM}brand of gaming consoles. In early 2001 they turned to IBM and Toshiba. Together they formed a partnership to deliver a chip that would both provide the power for the next generation of media rich gaming consoles, while also being a scalable, adaptable design that would meet the most demanding computational tasks. The result is a unique architecture combining a traditional central processor and specialised high performance processors similar to those found in graphics cards (GPUs). These processing units are combined across a circular high bandwidth bus to offer a multi-core environment with multiple-instruction sets and enormous processing power. Sony use a subset of the chip inside its Playstation 3 gaming and media console. IBM offer a range of configurations inside a Blade series suitable for server and super-computing use. Central to the Cell Broadband Engine (more commonly referred to as 'Cell') is a 3.2 GHz 64-bit Power Processing Unit (PPU). The PPU is a variant (970) of the G5/PowerPC product line, a RISC driven processor found in IBM's servers and Apple's PowerMac range. This PPU works as the primary processor and as supervisor for the other cores.

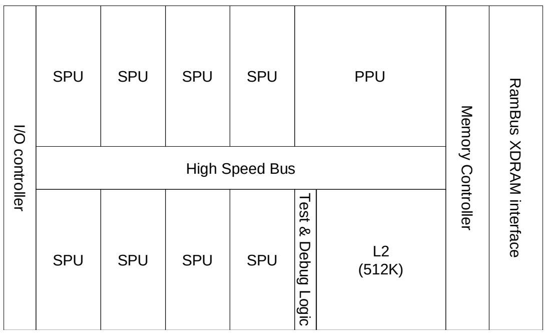


Figure 1: Cell BE Die Layout

The Cell's SPU The real power of the Cell is in the ability to harness the additional Synergistic Processing Units (SPUs). The SPU is a specialist processor with a RISC-like SIMD instruction set and a large (128) array of 128-bit registers. Each SPU has its own local memory store (LS). Currently, and on the Playstation 3, this LS is limited to just 256K. The SPU can access the LS in the same clock cycle as its register operations. The latest SDK (2.0) has a beta software cache implementation. While the SPU does not directly access main memory the central PPU can access each SPU's local memory store. The SPU has no hardware cache so each software application directly manages data transfer to and from each SPU. This leads to a number of interesting programming models.

While the architecture allows for any number of SPUs, a standard Cell, and those currently in production, has 8 SPUs. Interestingly Sony have chosen to utilise just 7 as they can gain much higher production yields if they can discard an SPU that shows a failure during silicon testing. Furthermore Sony restrict access to one SPU for DRM purposes on a Playstation running in Linux mode.

A processor with just 256K, no hardware cache and with no access to I/O doesn't appear to be anything exciting when compared to the PPU or other modern CPUs. The fact that the Cell offers 8 SPUs on one die all designed to operate in parallel makes this design so interesting. Combine this with the ability to work with up to 4 32-bit integer operations in just one clock cycle (referred to as SIMD) that make the SPU so interesting. The SPU also contains 2 instruction pipelines and while the pipelines are not equal, careful management of the order of instructions can lead to huge amounts of data being processed with very few clock cycles and a very low clock cycles per instruction (CPI) ratio.

The large register size is ideal for the number crunching operations required for cryptography. However, the fact that the size of the register is too large for most high level language's basic types, and that most operations work with, at most, 32-bit sub-sections of the quadword register, makes development a little tricky. The programmer accesses the registers through a set of C extensions which operate exclusively on vectors rather than traditional direct memory access. The C extensions (or intrinsics) also offer a degree of code portability with similar CPUs such as the Altivec [10]. It is possible to develop small, dedicated, standalone, SPU applications (spulets). A more interesting, but more complex model, is the capability of the PPU to call SPU applications through a POSIX threads-like library passing data through a rich direct memory access (DMA) library.

To stimulate interest within the development community IBM offer a software development kit and ample documentation [1]. This SDK contains a full range of development tools and code samples including a powerful cycle accurate simulator for the SPU and a vector optimised multi-precision Math library (IBM MPM) [6].

Multi-instruction sets One interesting issue with the different architectures of the PPU and SPU is the need for multi-instruction set binaries. Traditional applications compile individual source modules and then link the results to bind all program data symbols (variable, types functions etc.), but as the SPUs LS memory is physically separate and makes use of wide 128-bit registers, its program code needs to be compiled and linked separately. Both the SPU and PPU use standard ELF binary formats. An application's binary contains 64-bit code for the main PPU but embedded inside this is an object file with the SPU instructions and data ready to be pushed to the SPU on a `createthread()` call from the PPC. The build process involves two separate compilers and two linkers. The SPU ELF binary is passed through an `embedspu` command which builds a wrapper (a CESOF linkable) to the SPU binary marking it with PPU compatible symbols. Finally there is one more link stage which binds all executables together. Figure 2 [3] outlines the build process.

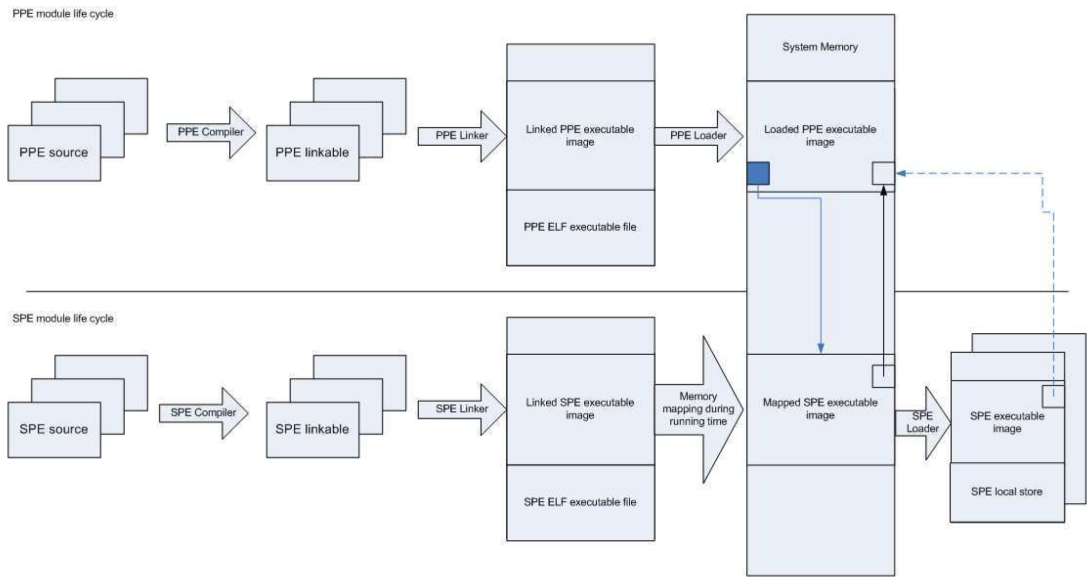


Figure 2: Cell BE build process [3]

For further information on the Cell see IBM's excellent Cell resource centre [5]

Direct Memory Access As mentioned above the PPU can access main memory and has instructions to transfer data between the main memory and its registers. The SPU, on the other hand, works with its own smaller local store and so to access data from the main memory the SPU goes through a *Memory Flow Controller* (MFC) which translates SPU main memory requests over the high speed bus via a set of DMA channel calls. These DMA calls are directional (read or write), blocking or non-blocking, can be issued in parallel, and can be tagged by the programmer to allow for identification management of data.

When communicating either the PPU or an SPU can initiate and manage a DMA transfer. However it is optimal for the SPU to do the 'protocol' management as it can free PPU clock cycles that can occur if, for example, a number of SPUs have blocking calls. When the PPU needs to initiate the transfer the procedure is for the PPU to *push* a pointer to the SPU with a tag and then let the SPU *pull* the data from the pointed reference and informing the PPU, via the tag, that it has done so.

Vector Programming To utilise the full performance of SPU SIMD instructions a developer works with a combination of Vector C extensions with assembly like code. Space is limited so we will look at the following extracts to highlight typical techniques used. We implemented a primitive `MADD()` commonly used in cryptographic libraries which fully utilises the 128-bit register by implementing a 64x64-bit multiply function.

For example the following code fragment is used to fill a quadword with two scalars (in this case standard C 64-bit `unsigned long long`) and to 'splat' across a vector. Splat is a term used when filling a vector with a mask. In a big number context we utilise splats to allow us operate on different elements of a quadword when filling partial products.

```
unsigned long long _a, _b
vector unsigned short AB;
AB=(vector unsigned short)  \
                    spu_insert(_a,(vector unsigned long long)AB,0x1);
AB=(vector unsigned short)  \
                    spu_insert(_b,(vector unsigned long long)AB,0x0);
/* select two bytes */
const vector unsigned char splat_short1= \
            (vector unsigned char)(VEC_SPLAT_U32(0x80800405));
```

Here we utilise a C macro to guarantee all vector multiplies (`spu_mulo()`) are at a 16-bit level to efficiently use the 16x16-bit multiplier in the SPU.

```
#define MULTIPLY(a, b)\
 (spu_extract(spu_mulo((vector unsigned short)spu_promote(a,0),\
            (vector unsigned short)spu_promote(b, 0)),0))
```

Finally an assembly-like example of a speed up technique when adding a 128-bit value to a 64-bit value where we know there is no need to manage an overflow. This technique is used in summing partial products inside the big number multiply.

```
vector unsigned int _out_s, _in_a128, _in_a64;
vector unsigned int _sum, _c0, _t0;

_c0    = spu_genc(_in_a128, _in_a64);    // generate carry bits
_sum   = spu_add(_in_a128, _in_a64);    // add
_t0    = spu_slqwbyte(_c0, 4);       // shift quadword left  4 bytes
_out_s  = spu_add(_sum, _t0);      // add in the carry
```

3 OpenSSL

OpenSSL [7] is an open source toolkit released under under a BSD style license. It evolved out of Eric Young's popular SSLeay and in 1998 passed to a dedicated team of developers. It has since become the de facto open source SSL toolkit. It is the security sub-system of choice for large open source projects such as Apache [8] and MySQL [12] and included in virtually all UNIX distributions including Linux, MacOSX^{TM}, and Solaris^{TM}.

The name 'OpenSSL' is misleading as the toolkit provides a vast array of building blocks and interfaces from cryptographic primitives through big number routines to PKI components such as certificate authorities and OCSP responders. One of the most useful features is the ability to factor out processing intensive operations to specialist hardware through an 'engine' interface. It is through this engine subsystem that we accelerate SSL by using the Cell SPU's vector processing capabilities.

SSL operates in two phases: an initial handshake and a bulk encryption phase. The purpose of the handshake is to swap identification credentials, algorithm capabilities, and negotiate a bulk encryption key. The reason for the key negotiation is that asymmetric cryptography, whilst needed to establish a shared secret, incurs a large computational overhead compared to a symmetric encryption algorithm. By analysing clock cycles, Zhao *et al.* [2] found that 90.4% of the SSL handshake comprises public key operations. Cryptographic operations take, in total, about 95% of the total CPU load.

Since the CPU load will be heaviest at the server side, and since the main computationally load incurred by the server for its part in the handshake is asymmetric decryption, we focus our attempts on speeding up asymmetric decryption.

Isolating the SSL handshake to measure our improvements is a challenging task as there are can be many dependencies (network traffic, HTTP server etc.) on a running machine which make accurate sampling difficult. Fortunately OpenSSL provides the utility `openssl speed` which can measure individual algorithms. Using this utility we can demonstrate improvements to the throughput of the critical algorithms. The SSL protocol supports a range of asymmetric algorithms, (RSA, DSA, ECC etc.). In this paper we focus on RSA but the technique is relevant to all.

OpenSSL engine When taking over computational tasks from OpenSSL two issues which must be considered are

1. How the engine informs the library of the scope of its responsibilities.

2. Marshalling the big number format to and from OpenSSL's internal representation.

To tell OpenSSL exactly what the engine will do the developer provides a static library with a set of defined interfaces with descriptive text to describe the engine. Then, through function pointer replacement, a defining a set of functions which implement the algorithms that the engine intends to provide.

While there are dynamic loading techniques for closed source libraries, at the current OpenSSL version (0.9.8d) the simplest method to integrate the engine is to statically link the engine code and add a call inside `ENGINE_load_built_in_engines()`. This will add the engine as an option to any application using the OpenSSL default engine.

Through this call OpenSSL then loads any engine that conforms to the correct interface at start-up, and subsequently any OpenSSL command that uses the `-engine <id>` option will redirect to the named engine. At the `Engine_init()` stage the calling library passes an OpenSSL data structure containing a set of initialisation variables and an opening via a free additional pointer for the engine to append its own data structure which can be accessed later by subsequent engine functions. The engine is responsible for its own memory management. It is through this `Engine_init()` call that we gather OpenSSL parameters and convert the OpenSSL big number representation to the native Cell IBM Multi-precision Big number format.

4 Development

To recap: we need to build a PPU library (32 or 64-bit) that plugs into a PPU build of OpenSSL. Inside this library we embed an SPU ELF executable which can act upon the 128-bit registers and utilises IBM's MPM library. This SPU ELF executable needs to be under 256K including all code and data. The multi-core environment with the limitations on code and data size requires some unconventional, data centric, programming models which the engineering community are still evolving. The cardinal rule appears to be 'offload as much as one can to the SPUs'. Many data intensive multimedia applications employ a model where data is streamed through a chain of SPUs with each SPU carrying out a specific operation on the data, then calling another SPU with the processed data. Yet another model makes the PPU act as a scheduler pushing data segments and code 'blobs' to any SPU with the PPU managing the operations and data ordering through double buffering. To fit the OpenSSL engine model, we mirror the operation of a similar engine developed by Geoff Thorpe of the OpenSSL core team for the GNU Multi-Precision library (GMP) [9]. To have the SPUs do as much work as possible we chose to overload the `RSA_mod_exp()` function and indicate through control flags that the engine would perform full RSA decryption using the Chinese Remainder Theorem. Figure 3 describes the interaction between the various components. This allows us to potentially parallelise the modular exponentiation calls. We could approach this a number of ways:

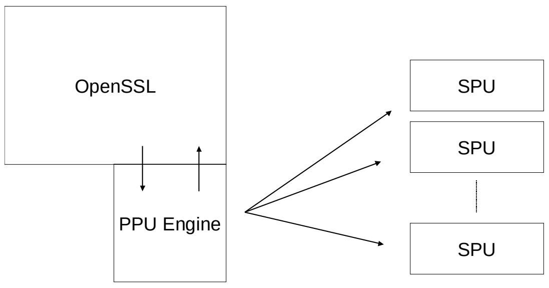


Figure 3: OpenSSL with Engine and SPUs

1. Have the PPU do the RSA/CRT but invoke SPUs to manage the expensive modular exponential (`mod_exp()`). Different SPUs would handle the p and q `mod_exp()`.
2. Have the PPU pass the whole RSA/CRT to an SPU.
3. Have the PPU pass the whole RSA/CRT to an SPU with this SPU passing the the two `mod_exp()` to two other SPUs.
4. Have the PPU pass the whole RSA/CRT to an SPU with this SPU passing one of the two `mod_exp()` to another SPU and, in parallel, handle the other.

There are a number of advantages to each. With (1) the amount of data in the DMA bus is reduced but breaks the guideline of offloading as much computation as possible to an SPU. With (3&4) the latency per SSL connection will be reduced but, as it adds extra DMA data to the bus, the over all maximum throughput will be affected. With (2, 3 & 4) we can double buffer the data transfer, for example passing the p parameter to the bus while the SPU is processing the q `mod_exp()`. The double buffering technique would offer relatively small speed gains. In an attempt to measure the maximum throughput we chose to focus on (2).

To maintain compatibility with OpenSSL and other engine implementations we use notation matching OpenSSL code: dmp_1, the decryption exponent $\mod p-1$, dmq_1, the decryption exponent $\mod q-1$. $iqmp$ is the inverse of $q \mod p$. I_0 is the cyphertext. A decryption exponent d, for a prime p, is a number d, such that $m^{ed} \mod p = m$ or $ed = 1 \mod (p-1)$, where e is the encryption exponent, commonly chosen to be 3 or 65537.

At the RSA initialisation stage OpenSSL passes the parameters ($p, q, dmp_1, dmq_1, iqmp$) to the engine. At this stage we check the parameters, allocate a memory store, fill the store with local copies of the big numbers ready to pass to an SPU, and then pass the memory store pointer back through a thread safe thread local memory store. OpenSSL later makes a call to the main overloaded `RSA_mod_exp()` function with I_0 and the same thread memory store parameter. The overloaded `mod_exp()` extracts the thread local data, calls an SPU thread, DMA transfers the location and size of the memory store to the SPU. It then allocates space for the return data from the SPU.

As mentioned above, the SPU thread when activated could either receive all parameters in a full DMA transfer or, more efficiently, a pointer to the block of big numbers in memory on the Cell's main store. By passing the pointer, the SPU's memory flow controller effectively takes the memory processing away from the main PPU, further improving the performance.

At this stage the SPU thread converts the big number set to the IBM MPM format and carries out the CRT logic. On success it takes the result, pushes it back to the PPU using the DMA tag that the I_0 parameter was sent with, finally cleaning up any memory used by the engine and exiting.

RSA/CRT We implement traditional RSA Decryption using Chinese Remainder Theorem but with a small modification. Because the SPU is restrictive in some respects and as we can't be certain that the parameter p is always greater than q we need to maintain a sequence of calls that ensure the results of any modular exponentiation stay positive.

1. The SPU compiler optimiser is most efficient when there is no branching.
2. The current version (SDK 2.0) of the IBM MPM is intended to work with unsigned numbers.
3. Integer comparison operations (less than, greater than) on negative numbers are undefined.

To overcome these restrictions we assume p is always less than q. A condition OpenSSL guarantees. The modified algorithm is outlined in Algorithm 1.

5 Results

Table 1 lists timings in cycles counts and milliseconds for the time consuming functions of the RSA/CRT implementation. Two totals are presented: sum of these calls and an observed timing for all calls including some initialisation and the DMA receive calls. These timings are made using an engine with just one SPU configured.

We can see that, as expected, the `mpm_mont_mod_exp()`[2] calls represent the bulk of the time consuming operations. A case could be made for a design that offloaded just this call to an SPU. Theoretically (from the results of Table 1) we can expect the SPU to be able to process 14.8 4096-bit decryptions in a second. Interesting (from Table 2) we achieve close to this at 14.1. Obviously there is additional overhead

[2]The generic `mpm_mod_exp()` clocks at 136914856 cycles for a 4096-bit modulus

Algorithm 1 RSA Decryption using Chinese Remainder Theorem modified for the IBM MPM unsigned restrictions. *Note: We follow OpenSSL notation found in all engine implementations*

INPUT: $p, q, I_0, dmq_1, dmp_1, iqmp$
OUTPUT: r_0

```
$r_1 \leftarrow I_0 \mod q$
$m_1 \leftarrow {r_1}^{dmq_1} \mod q$
$r_1 \leftarrow I_0 \mod p$
$r_0 \leftarrow r_1^{dmp_1} \mod p$
$r_0 \leftarrow r_0 - m_1$
if $r_0 < 0$ then
  $r_0 \leftarrow r_0 + p$
end if
$r_1 \leftarrow r_0 \cdot iqmp$
$r_0 \leftarrow r_1 \mod p$
$r_1 \leftarrow r_0 \cdot q$
$r_0 \leftarrow r_1 + m_1$
```

function	calls	cycle count	total cycles	total millisecs	secs	#/sec
big_number_ convert()	7	877	6139	0.00192		
mpm_mod()	4	77731	310924	0.09716		
mpm_mont_mod_exp()	2	93328909	186657818	58.33057		
mpm_mul()	1	22733	22733	0.00710		
mpm_sub()	1	704	704	0.00022		
mpm_add()	1	1116	1116	0.00035		
mpm_madd()	1	39648	39648	0.01239		
total function calls		187039082			0.05845	
Total *includes other calls*		215159632			0.06724	14.87

Table 1: RSA/CRT decryption implemented in IBM MPM function calls with cycle count and time in milliseconds for a 4096-bit key

from DMA and the process queue on the main PPU. Cycle counts are from the latest (2.0) version of the SDK's simulator. Unfortunately the simulator (at this time) cannot measure DMA or PPU latency.

As mentioned previously, to get some sense of the improvements our optimisations have made we use the `openssl speed` command on RSA with the engine off (native OpenSSL on the PPU) and with our engine on utilising the SPU.

Tests are run on a 3.2 GHz Playstation 3 with just **6** SPUs running Yellow Dog Linux 5.0 [14] with kernel version 2.6.16-20061110.ydl.1ps3. A server/blade Cell system would have up to 16 SPUs. We could expect the Playstation Cell to deliver a throughput of up to 89 sign/sec and a blade server to go as high as 237 sign/sec. Our observations (Table 3) see slightly smaller results. As mentioned there are number of factors that could skew our observed numbers, mainly the design of the OpenSSL speed post-processing, DMA overhead and the fact that the PPU is busy managing the multiprocess queue.

We are using the `openssl speed -elapsed` time option instead of the more often quoted CPU user time as on the multi-core processor the CPU timer will just count the CPU time of the driving PPU thread whereas the multi-threaded nature of the SPU based system is better represented by elapsed time.

OpenSSL is configured for 64-bit PPC/G5 ASM [3].

```
openssl speed rsa -elapsed
openssl speed rsa -engine cellspumpm -elapsed
```

RSA	PPU		1 SPU	
key length	sign	sign/sec	sign	sign/sec
1024-bits	0.003435s	291.2	0.005655s	176.8
2048-bits	0.017541s	57.0	0.015636s	64.0
4096-bits	0.109793s	9.1	0.070915s	14.1

Table 2: OpenSSL speed on PPU vs. 1 SPU using IBM-MPM on 3.2GHz Cell/PS3.

From Table 3 we can see that the overhead of the DMA transfer and the big number conversion impact the performance improvements just below the 2048-bit key. The benefits of the 128-bit registers are apparent at 4096-bit level with improvements in the order of 150% (14.1 vs. 9.1).

To see the full impact of the multi-core we need to use the -multi [n] option to the speed command which can (through `fork()`) generate multiple simultaneous RSA operations. We have picked a number (6) of parallel processes to run matching the number of SPUs on the Playstation 3. It is important to note that the -multi option introduces some small processing overhead to the speed command as it uses a fork() invocation whereas the standard calls in single threaded. Again we compare the PPU with an SPU enabled engine.

```
openssl speed rsa -elapsed -multi 6
openssl speed rsa -engine cellspumpm -elapsed -multi 6
```

We see from Table 3 similar overheads impacting the 1024-bit keys. However there is huge improvements in 2048-bit (329.7 vs.71.7) and 4096-bit (83.6 vs 11.2). A 749% increase.

RSA	PPU		6 SPUs	
key length	sign	sign/sec	sign	sign/sec
1024-bits	0.000724s	384.5	0.001906s	524.7
2048-bits	0.002600s	71.7	0.003033s	329.7
4096-bits	0.089455s	11.2	0.011925s	83.9

Table 3: OpenSSL speed on PPU vs. 6 SPUs using IBM-MPM on 3.2GHz Cell / PS3, 6 parallel processes.

While the `openssl speed` utility running on the 6 SPU Cell inside a Playstation 3 gives us a solid basis to develop and measure our improvements, Séan Starke at IBM was kind enough to try our tests in a full 16 SPU dual Cell blade. These results (Table 4) are preliminary but are consistent with the trend from the Playstation 3 results.

[3]Options: bn(64,64) md2(int) rc4(ptr,char) des(idx,risc1,16,long) aes(partial) idea(int) blowfish(idx) compiler: ppu-gcc -DOPENSSL_USE_MPM_SPU -DOPENSSL_THREADS -D_REENTRANT -DDSO_DLFCN -DHAVE_DLFCN_H -m64 -DB_ENDIAN -DTERMIO -O3 -Wall

RSA	2 PPUs		16 SPUs	
key length	sign	sign/sec	sign	sign/sec
1024-bits	0.001270s	787.5	0.001509s	662.7
2048-bits	0.006805s	146.9	0.001664s	601.0
4096-bits	0.043944s	22.8	0.005762s	173.6

Table 4: OpenSSL speed on 2 PPUs vs. 16 SPUs using IBM-MPM on 3.2GHz Cell, 16 parallel processes.

6 Conclusions and Future Work

The numbers speak for themselves: over 700% improvement in performance. With the widespread use of specialised multi-core processors there is no reason to prevent the roll out of always on encryption leading the improvements in privacy for the general user.

We believe that we have pushed the Cell SDK's IBM-MPM library to its limits. The library is intended for general purpose use for diverse applications such as FFTs and scientific computing. It is a excelent demonstration of the power of SPU instrincs 'vector' programming. However, we believe the introduction of an optimised number library more suited to crypto can substantially improve the performance, possibly doubling the figures presented above.

As mentioned the results are based on using generic Montgomery `mpm_mont_mod_exp()` function. This function allows for any size of parameter whereas we know the size of parameters are based on fixed key lengths (1024, 2048 etc.) These fixed lengths can offer further optimisations as they always align on the 128-bit boundaries of the vectors and that the number of partial products to be summed inside any multiplies can be determined allowing for very efficient carry management.

The multiplication inside the `mpm_mont_mod_exp()` needs to be examined in more detail. MPM uses 'row by row' operand scanning to do big number multiplies whereas a 'column by column' product scanning technique used by the Comba [4] method would be more suitable for the large, fixed sized numbers used by crypto. Furthermore, as the number length moves beyond 1024-bit the Comba method can be combined with the Karatsuba technique [11] for further improvement.

OpenSSL uses this Comba/Karatsuba combination at key lengths above 1024-bit irrespective of the architecture. We hope to swap out the IBM MPM library and use a fine tuned version of MIRACL [13] with fixed key sizes on fixed 128-bit alignment, and utilising the Comba/Karatsuba speed ups on longer key lengths.

The threshold key length to optimally use the Karatsuba method depends heavily on the underlying word size and the architecture's instruction set, specifically how fast the multiplier is compared to the addition. We hope to examine this threshold in more detail with the more flexible, fit for purpose, MIRACL library.

The performance figures focus on raw crypto performance. We would like to examine the SSL performance of real word data using a commercial grade SSL/HTTP load testing suite. We also intend to offer support for DSA and ECC algorithms.

7 Acknowledgements

For development tools and background information we turned again and again to the IBM's 'Developer-Works' resource centre and the Cell SDK. We would like to thank the Cell development community particularly Séan Starke and the IBM team. We would also like to thank Augusto Jun Devegili (Unicamp, Brazil), Peter Kehoe (DCU), Noel McCullagh, and Stephen Henson (OpenSSL)[7] for their encouragement, assistance & patience.

References

[1] IBM alphaWorks. Cell Broadband Engine SDK. `http://www.alphaworks.ibm.com/topics/cell`.

[2] Zhao Iyer Srihari Makineni Laxmi Bhuyan. Anatomy and performance of SSL processing. In *Proc. IEEE Int. Symp. Performance Analysis of Systems and Software*, pages 197–206, 2005.

[3] Alex Chunghen Chow. Programming the Cell Broadband Engine. *Embedded Systems Design*, 2006. `http://www.embedded.com/columns/showArticle.jhtml?articleID=188101999`.

[4] P. G. Comba. Exponentiation cryptosystems on the ibm pc. *IBM Syst. J.*, 29(4):526–538, 1990.

[5] IBM DeveloperWorks. Cell Broadband Engine resource center. `http://www-128.ibm.com/developerworks/power/cell/`.

[6] IBM DeveloperWorks. Cell Broadband Engine SDK Libaries Multi-Precision Math Library, 2006. `http://www-128.ibm.com/developerworks/power/cell/`.

[7] S. Henson et al. OpenSSL library. Open source library, 1988. `http://www.openssl.org`.

[8] Apache Foundation. Apache HTTP server project. `http://www.apache.org`.

[9] SWOX / Free Software Foundation. GNU Multiple Precision Arithmetic Library. `http://www.swox.com/gmp/`.

[10] Freescale. Altivec velocity engine. `http://www.freescale.com/altivec`.

[11] Donald E. Knuth. *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[12] MySQL. MySQL open souce database. `http://www.mysql.com`.

[13] M. Scott. MIRACL. `http://www.shamus.ie`.

[14] Terrasoft. Yellow Dog Linux. `http://www.terrasoftsolutions.com/products/ydl/`.

Montgomery Modular Multiplication Algorithm for Multi-Core Systems

Junfeng Fan, Kazuo Sakiyama, and Ingrid Verbauwhede

Katholieke Universiteit Leuven,ESAT/SCD-COSIC,
Kasteelpark Arenberg 10
B-3001 Leuven-Heverlee, Belgium
{Junfeng.Fan,Kazuo.Sakiyama,Ingrid.Verbauwhede}@esat.kuleuven.be

Abstract. This paper presents an efficient software implementation of the Montgomery modular multiplication algorithm on a multi-core system. A prototype of general multi-core systems is designed with GEZEL. We propose a new instruction scheduling method for multi-core systems that can reduce the number of data transfers between different cores. Compared to the implementations on a single-core system, the performance can be improved by a factor of 1.87 and 3.68 when 256-bit modular multiplication being performed on a 2-core and 4-core system, respectively.

Key words: Montgomery Modular Multiplication, Multi-core, Parallel Computation

1 Introduction

Modular multiplication is a fundamental operation in many popular Public Key Cryptography (PKC) algorithms such as RSA [1] and ECC [2, 3]. As the division operation in modular reduction is time-consuming, Montgomery [4] proposed a new algorithm where division is avoided. An integer Z is represented as $Z \cdot R \bmod M$, where M is the modulo and $R = 2^r$ is a radix that is coprime to M. This representation is called Montgomery residue. Multiplication is performed in this residue, and division by M is replaced with division by R. This algorithm can be easily implemented on general purpose processors. However, due to the highly intensive computation, software implementations are offen not fast enough. Many hardware implementations [5–7] were proposed to improve the performance.

The increasing use of multi-core systems have opened another window for improving the performance of software implementations. Processor vendors have published various dual-core [8] and quad-core [9] processors for personal computers. Even for embedded systems several multi-core processors [10, 11] are now available. Therefore, multiple cores in the system can be utilized to perform the intensive computation and the software implementations of the Montgomery algorithm can then be accelerated by parallel computation.

When performing parallel computation, task scheduling is highly dependent on the hardware architecture. If the architecture is based on a super-scalar processor, the task will be automatically partitioned. In this paper, we consider general multi-core systems that do not have this feature. We use a Very Long Instruction Word (VLIW) processor as a prototype. This processor can be configured to have 1, 2, 4, 8 or even more cores. Each core can work separately. To be general, only the very basic instructions are supported. The Montgomery algorithm is partitioned in algorithm level and tasks are mapped to each core. We explore two different scheduling methods to find bottlenecks.

The rest of the paper is organized as follow. Section 2 briefly reviews previous work on the Montgomery algorithm and its parallel implementations. In section 3, we describe the multi-core architecture of our platform. Two instruction scheduling methods are proposed in section 4 and comparison between them is given in section 5. Finally, we show implementation results in section 6 and conclude the paper including future work in section 7.

2 Previous Work

The Montgomery modular multiplication algorithm was designed to avoid division in modular multiplications. Given two n-bit inputs, X and Y, this algorithm gives $Z = X \cdot Y \cdot R^{-1} \bmod M$, where R equals to 2^n and M is the n-bit modulo. A modified Montgomery multiplication algorithm was proposed to avoid the conditional final substraction by choosing a suitable R [12]. Algorithm 1 shows the Montgomery algorithm with the conditional substraction.

Algorithm 1 Radix-2^w Montgomery modular multiplication (FIOS) [13]

Input: integers $M = (M_{s-1}, ..., M_0)_r$, $X = (X_{s-1}, ..., X_0)_r$, $Y = (Y_{s-1}, ..., Y_0)_r$, where $0 \leq X, Y < M$, $r = 2^w$, $s = \lceil \frac{n}{w} \rceil$, $R = r^s$ with $gcd(M, r) = 1$ and $M^{'} = -M^{-1} \bmod r$.
Output: $X \cdot Y \cdot R^{-1} \bmod M$

```
1: $Z = (Z_{s-1}, ..., Z_0)_r \leftarrow 0$
2: for $i = 0$ to $s - 1$ do
3:     $T \leftarrow (Z_0 + X_0 \cdot Y_i) \cdot M^{'} \bmod r$
4:     $Z \leftarrow (Z + X \cdot Y_i + M \cdot T)/r$
5: end for
6: if $Z > M$ then
7:     $Z \leftarrow Z - M$
8: end if
9: return $Z$
```

As shown in Algorithm 1, the operands X, Y and M are divided into w-bit words. In the beginning of each iteration, $X_0 \cdot Y_i$ is calculated to generate T. After the generation of T, the multiplication of $X \cdot Y_i$ and reduction of C are performed together by computing $Z = Z + X \cdot Y_i + M \cdot T$. After that, Z_0 always becomes 0. The division of Z by r is performed by shifting Z one word to the right. After s iterations and one conditional substraction, $Z = X \cdot Y \cdot R^{-1} \bmod M$ is obtained. As Algorithm 1 scans the operands X and M from Least Significant Bit (LSB) to Most Significant Bit (MSB) simultaneously, it is also called Finely Integrated

Operand Scanning (FIOS). It is possible to perform $Z = Z + X \cdot Y_i$ first, and then $Z = Z + M \cdot T$. This modified algorithm is called Coarsely Integrated Operand Scanning (CIOS) [13] and is presented in Algorithm 2.

Algorithm 2 Radix-2^w Montgomery modular multiplication (CIOS) [13]

Input: integers $M = (M_{s-1}, ..., M_0)_r$, $X = (X_{s-1}, ..., X_0)_r$, $Y = (Y_{s-1}, ..., Y_0)_r$, where $0 \leq X, Y < M$, $r = 2^w$,$s = \lceil \frac{n}{w} \rceil$,$R = r^s$ with $gcd(M, r) = 1$ and $M' = -M^{-1}$ mod r.
Output: $X \cdot Y \cdot R^{-1}$ mod$M$

```
 1: $Z = (Z_{s-1}, ..., Z_0)_r \leftarrow 0$
 2: for $i = 0$ to $s - 1$ do
 3:     $T \leftarrow (Z_0 + X_0 \cdot Y_i) \cdot M'$ mod $r$
 4:     $Z \leftarrow (Z + X \cdot Y_i)$
 5:     $Z \leftarrow (Z + M \cdot T)/r$
 6: end for
 7: if $Z > M$ then
 8:     $Z \leftarrow Z - M$
 9: end if
10: return $Z$
```

So far, many task scheduling methods have been proposed. Kaihara *et al.* [14] designed a bipartite multiplier, where the modular multiplication was divided into two separated tasks. Besides, systolic array [15] is deployed in hardware implementations. Various implementations [16–18] of systolic array were proposed to improve the performance. However, most of the scheduling methods are targeting a fast hardware implementation, data transfers between PEs are almost free since they can be performed with a hardwired communication. In general purpose multi-core systems, different cores exchange data via shared memory. Thus, frequent data transfers can make a heavy overhead. Therefore, those scheduling methods need to be modified to fit software implementations.

In this paper we propose a new scheduling method, which can efficiently reduce the data transfers between different cores. This method is highly scalable and can achieve high performance.

3 Our Design Platform

It doesn't make sense to fix the hardware architecture (i.e. the number of cores) and explore the best software algorithm for the fixed hardware configuration. The hardware/software co-design with a multi-core system, the main focus of this paper, needs an environment to get a quick and correct evaluation of cost and performance for various hardware configurations and software programs. Thus, we use a simulation environment, called GEZEL [19], which allows us to estimate immediate system performance in a cycle-accurate manner before synthesizing the entire design.

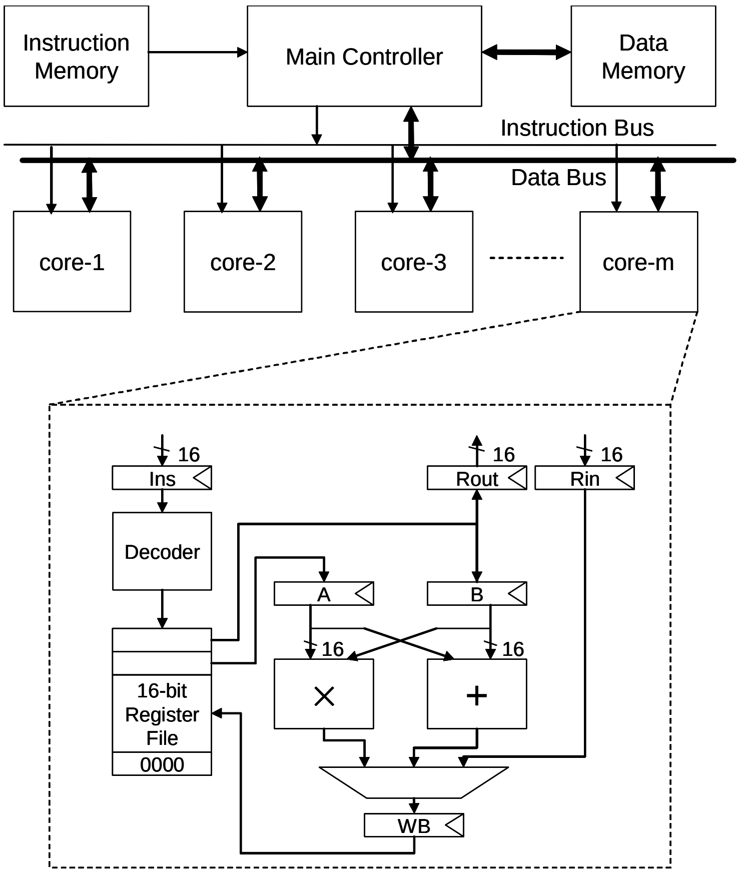


Fig. 1. Platform architecture. ($w = 16$).

General multi-core systems can have various architectures and corresponding memory organizations. For instance, they may contain symmetrical cores with a shared memory, or a master CPU with several slave CPUs/DSPs connected to the system bus. To be as general as possible, we use a VLIW architecture processor. The purpose of this prototype processor is to explore different algorithms on multi-core systems.

As shown in Figure 1, this platform consists of a main controller, a data memory, an instruction memory and several cores. Only the main controller can access the instruction memory and the data memory. The main controller fetches instructions from the instruction memory and dispatches them to all cores in parallel via the instruction bus. Each core executes arithmetic instructions in parallel, and stores the results in its register file. The data memory has only one read/write port, therefore, a single data memory access is allowed in each cycle.

We denote w as the operation size of w-bit core. A 16-bit ($w = 16$) core is also shown in Figure 1. It is a highly simplified Load/Store CPU. It has a instruction decoder, a register file with 16 general 16-bit registers and one status register. The Arithmetic Logic Unit(ALU) includes one 16-bit multiplier and one 16-bit adder. It also has an output register to store the data that will be written to the data memory, and an input register to buffer the data from the data memory.

Table 1. Instruction sets for one core.

Opcode 4-bit	**Operand 1** 4-bit	**Operand 2** 4-bit	**Operand 3** 4-bit	**Description**
`Nop`				No operation
`Load`	`Ri`	`#Addr`		Load the data from location `Addr` of the data memory into register `Ri`
`Store`	`Ri`	`#Addr`		Store the data of register `Ri` to location `Addr` or the data memory
`Mul`	`Ri`	`Rj`	`Rk`	`{R(i+1),Ri} = Rj· Rk`
`Add`	`Ri`	`Rj`	`Rk`	`{Ca,Ri} = Rj + Rk`, `Ca` is the carry out and is stored in the status register
`Adc`	`Ri`	`Rj`	`Rk`	`{Ca,Ri} = Rj + Rk + Ca`
`Sub`	`Ri`	`Rj`	`Rk`	`Ri = Rj - Rk`

Both of them are 16-bit. One 32-bit Write Back (WB) register is also used to store data from the ALU. The bit-length of both data-path and registers are doubled if it is configured as a 32-bit ($w = 32$) core.

The cores here support a simple Load/Store Instruction Set Architecture (ISA). As shown in Table 1, this simplified ISA has only 7 general instructions. Here `#Addr` denotes memory address. Instructions for each core are of 16-bit long. All the arithmetic operations are performed among data stored in the local register file. When data needs to be moved from one core to another, it is first stored to the data memory, then it is loaded by the destination core. Cores in this platform support a 4-stage instruction pipelining.

4 Instruction Scheduling

Before we schedule the instructions, the data dependency is analyzed. The main dependency in the Montgomery algorithm is due to the carries of additions. Taking FIOS shown in Algorithm 1 as an example, in each iteration, Z_j is replaced by $(Z_j + (X \cdot Y_i)_j + (M \cdot T)_j + Ca)$, where Ca is the carry. The data dependency in one iteration is shown in Figure 2. Obviously, $X_j \cdot Y_i$, for any $0 \le i, j \le s-1$, is only dependent on the operands X and Y. We can also calculate $M_j \cdot T$ immediately after the generation of T. The products with the same weight of Z_j and the carry from Z_{j-1} are accumulated to Z_j, generating a new Z_j and 2-bit carries. As a result, Z_j can only be generated after carry from Z_{j-1} is ready.

As shown in Figure 2, we need to add Z_j with four w-bit data and 2-bit carries. In hardware implementations, cascaded Carry Save Adders (CSAs) can be used to construct a 6-to-2 CSA. The carry can also be saved in a 2-bit register or transferred to another PE. However, in general purpose processors these special features are not available. Normally only general adders with a

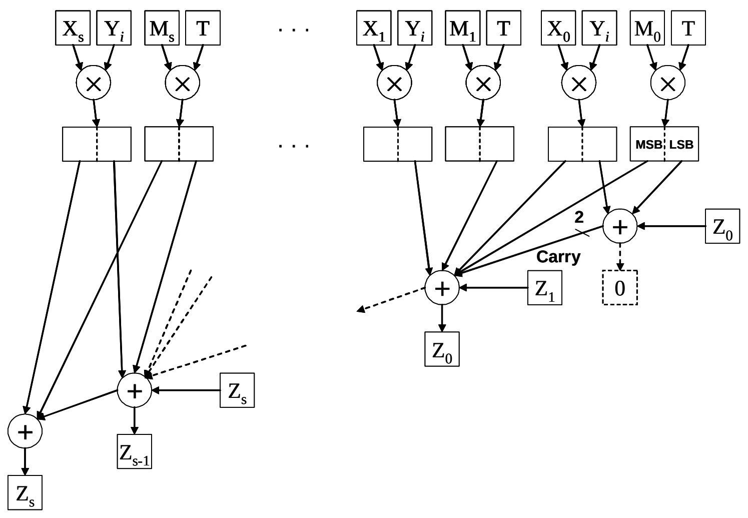


Fig. 2. Data dependency of FIOS Montgomery algorithm.

fixed length are used. The carry is saved in the status register after an `Add` instruction. In order to keep the 1-bit carry for future use, one instruction is needed to copy it from the status register to a general register. It will be very inefficient to use carries generated by another core, since it needs to be stored to register file first, and then transferred via the data memory.

Therefore, it will be desirable to partition the algorithm so that carry is only used in the core where it was generated. In [6], Tenca and Koç proposed an iteration-based scheduling method. In this method each Processing Element (PE) performs one iteration of the loop in Algorithm 1. This method is attractive because carries are only used in the local PE. Note that this method was originally designed for a hardware implementation. Here we map this algorithm to general purpose multi-core systems. Figure 3 shows the scheduling method, denoted as method-I, for 256-bit Montgomery multiplication for a 4-core system. As $n = 256$ and $w = 16$, sixteen iterations are needed. Core-1 performs the first iteration and generates Z_0 to Z_{15} one bye one. Each word is transferred to core-2 as soon as it is generated. Core-2 then performs the second iteration and then transfers Z_0 to Z_{15} to core-3. After 4 iterations $Z = (Z_{15}, ..., Z_0)$ is transferred back to core-1 from core-4 and the 5^{th} iteration begins. As in total 16 iterations are required, each core needs to perform 4 iterations. After a conditional substraction, the result is obtained.

Though the method-I can avoid carry transfers between cores, transferring $(Z_{s-1}...Z_0)$ causes a heavy overhead. In Figure 3 the transfers of $(Z_{s-1}...Z_0)$ are denoted as arrows. For each iteration $s = \lceil \frac{n}{w} \rceil$ arrows are required to transfer Z. Since one modular multiplication contains s iterations, $s(s-1)$ arrows are needed during the whole loop. Let N_{arrow} be the number of arrows, then N_{arrow} is $s(s-1)$. In Figure 3 we have $s = 16$, therefore $N_{arrow} = 240$.

Note that in order to generate T, only Z_0 must be ready at the end of each iteration, while $(Z_{s-1}...Z_1)$ can be generated later. Based on this observation, a new scheduling method is proposed and is shown in Figure 4. In this method,

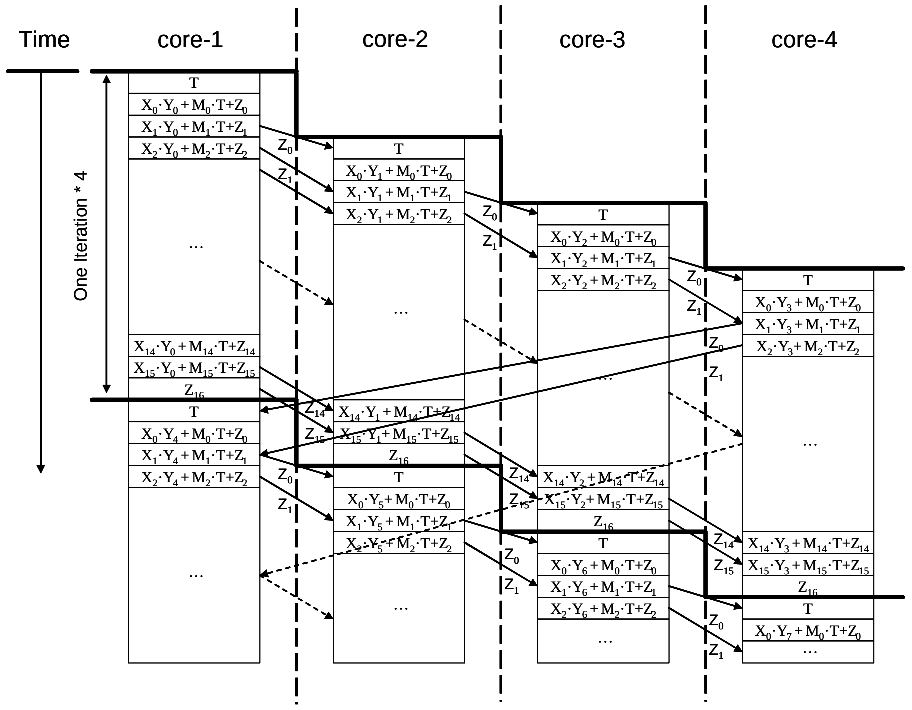


Fig. 3. Instruction scheduling method-I: One iteration is performed with one core. ($n = 256$, $w = 16$, $s = 16$, $N_{arrow} = 240$).

each iteration in Algorithm 1 is performed by multiple cores. Figure 4 shows this scheduling method for a 4-core system. Here we still choose $n = 256$, $w = 16$ and $s = \lceil \frac{n}{w} \rceil = 16$. During the whole loop $(Z_3, .., Z_0)$ is generated and stored in core-1, $(Z_7, .., Z_4)$ in core-2, $(Z_{11}, .., Z_8)$ in core-3 and $(Z_{15}, .., Z_{12})$ in core-4. Carry is only used in the local core. At the end of each iteration, Z_4 is sent to core-1, Z_8 is sent to core-2 and Z_{12} is sent to core-3. After sixteen iterations and a conditional substraction, $Z = X \cdot Y \cdot R^{-1} \bmod M$ is generated and stored separately in four cores. Z can be written to the data memory or can be used by another modular multiplication.

The method-II needs significantly less data transfers between cores than the method-I. T is always generated in core-1, and then distributed to other cores. On a p-core system, $p - 1$ arrows are needed to transfer T in each iteration. To shift Z to the right, $p - 1$ word is transferred, making $p - 1$ arrows in each iteration. As a result, the number of arrows for one modular multiplication is $2(p - 1)s$. When $s = 16$ and $p = 4$, the number of arrows, $N_{arrow} = 96$.

Each arrow in Figure 3 and Figure 4 causes one store and one load operation. The comparison of memory accesses caused by data transfers is presented in Table 2. Here $N_{load-tr}$ and $N_{store-tr}$ are the number of load and store operations caused by data transfers, respectively. $N_{total-tr}$ is the sum of them. Note that in Figure 4 the multiple arrows starting from T cause only one store operation. For the method-II $N_{total-tr}$ is $3ps - 2s$, while $2s^2 - s$ in the method-I. As p is always

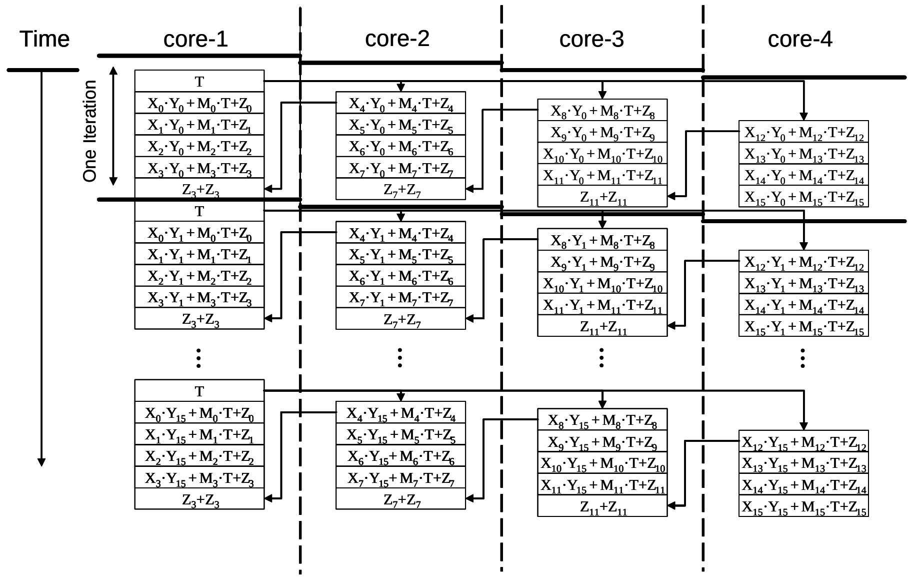


Fig. 4. Instruction scheduling method-II: One iteration can be performed with several cores. ($n = 256$, $w = 16$, $s = 16$, $N_{arrow} = 96$).

much smaller than s, the number of memory accesses caused by data transfers in the method-II is much smaller than that of the method-I.

5 Performance Comparison

Compared to the method-I, the method-II has two major advantages. First, operands and intermediate data are distributed in the register file of each core, thus less registers are required in each core. Second, less data transfers reduce memory accesses, as a result, a single-port data memory can support more cores before becoming the bottleneck. Detailed performance comparison between the method-I and the method-II is given below.

Let N_{mul} be the number of multiplications, N_{add} the number of additions and $N_{load-opr}$ the number of operand load operations. As shown in Table 2, we use $N_{load-tr}$ and $N_{store-tr}$ to denote the number of load and store operations caused by transferring intermediate data, respectively. The total number of memory accesses is denoted as N_{total}. Suppose that Algorithm 1 is implemented, N_{add} is always $4s^2 + s$, and N_{mul} is $2s^2 + s$ regardless the value of p.

Table 2. Number of data memory accesses caused by data transfers.

Scheduling Methods	N_{arrow}	$N_{load-tr}$	$N_{store-tr}$	$N_{total-tr}$
Method-I	$s(s-1)$	$s^2 - s$	s^2	$2s^2 - s$
Method-II	$2(p-1)s$	$2(p-1)s$	ps	$3ps - 2s$

Although N_{mul} and N_{add} are constant, N_{load} and N_{store} vary from different scheduling methods or different number of cores. Since the size of register files has a great influence on the number of memory accesses, it must be taken into account. If the register file is large enough, the operands, X, Y and M can stay in the registers during the whole loop. If not, they may need to be reloaded in each iteration, thus N_{total} becomes larger. Let S_{rf} be the number of entries of w-bit registers in each core's register file. Table 3 shows how the number of load and store operations changes for different size of register files.

Suppose Algorithm 1 is implemented on a w-bit processor. First, if S_{rf} is larger than $3s$, then X, M and Y only need to be loaded to the registers in the beginning of the loop, making $N_{load-opr} = 3s$. Since Z is generated and always stay in the registers in the whole loop, both $N_{load-tr}$ and $N_{store-tr}$ are 0. For $2s < S_{rf} \leq 3s$, only Z and X can be stored in the registers. The $N_{load-opr}$ increases to $s^2 + 2s$. For $s < S_{rf} \leq 2s$, only Z can be stored in the registers, thus $N_{load-opr}$ becomes $2s^2 + s$. For $S_{rf} \leq s$, X,M, Y_i and Z will be loaded from the data memory in each iteration, making $N_{load-opr} = 2s^2 + s$. Z also has to be sent to the data memory in each iteration, which leads $N_{load-tr} = N_{store-tr} = s^2$.

Now suppose that this processor has p general purpose cores. If the method-I is used, each core needs to use X, M and Y_i in each iteration. For $S_{rf} > 2s$, X and M can stay in the registers during the whole loop. In order to load X, M and Y_i it takes $2ps + s$ cycles. In order to transfer Z from one core to another s load operations and s store operations are required. Thus, for s iterations we need $N_{load-tr} = s(s-1)$ and $N_{store-tr} = s^2$ in total. For $s < S_{rf} \leq 2s$, only X can stay in registers during the whole loop, while M and Y_i are reloaded in each iteration. As a result, $N_{load-opr}$ increases to $s^2 + ps + s$. For $S_{rf} \leq s$, X, Y, M and Z need to be reloaded in each iteration, making $N_{load-opr} = 2s^2 + s$.

Table 3. The number of cycles required for one Montgomery multiplication for various Register File size (S_{rf}).

Processor type	S_{rf}	$N_{load-opr}$	$N_{load-tr}$	$N_{store-tr}$	N_{total}
Single-core	$S_{rf} > 3s$	$3s$	0	0	$3s$
	$2s < S_{rf} \leq 3s$	$s^2 + 2s$	0	0	$s^2 + 2s$
	$s < S_{rf} \leq 2s$	$2s^2 + s$	0	0	$2s^2 + s$
	$S_{rf} \leq s$	$2s^2 + s$	$s(s-1)$ *	s^2 *	$4s^2$
Multi-core Method-I	$S_{rf} > 2s$	$2ps + s$	$s(s-1)$	s^2	$2s^2 + 2ps$
	$s < S_{rf} \leq 2s$	$s^2 + ps + s$	$s(s-1)$	s^2	$3s^2 + ps$
	$S_{rf} \leq s$	$2s^2 + s$	$s(s-1)$	s^2	$4s^2$
Multi-core Method-II	$S_{rf} > \frac{3s}{p}$	$2s + ps$	$2(p-1)s$	ps	$5ps$
	$\frac{2s}{p} < S_{rf} \leq \frac{3s}{p}$	$s^2 + ps + s$	$2(p-1)s$	ps	$s^2 + 4ps - s$
	$\frac{s}{p} < S_{rf} \leq \frac{2s}{p}$	$2s^2 + ps$	$2(p-1)s$	ps	$2s^2 + 4ps - 2s$
	$S_{rf} \leq \frac{s}{p}$	$2s^2 + s$	$s^2 + (2p-3)s$ *	$s^2 + s$ *	$4s^2 + 2ps - s$

*Including store and load operations caused by calculating intermediate data.

For the method-II, the memory accesses are less. Since each core keeps a part of X, M and Z, the required register size of each core is less than that of the method-I. To keep X, M and Z in the registers during the whole loop, S_{rf} needs to be larger than $\frac{3s}{p}$. In order to load X,Y and M into registers it takes $2s + ps$ cycles, namely $N_{load-opr} = 2s+ps$. In order to distribute T, one store and $(p-1)$ load operations are needed in each iteration. In order to shift Z one word to the right, $(p-1)$ words of Z must be stored to the data memory and loaded in each iteration. As a result, we need $N_{load-tr} = 2(p-1)s$ and $N_{store-tr} = ps$. For $\frac{2s}{p} < S_{rf} \leq \frac{3s}{p}$, we only keep X and Z in the registers, while load M in each iteration. The $N_{load-opr}$ increases to $s^2 + ps + s$. For $\frac{s}{p} < S_{rf} \leq \frac{2s}{p}$, only Z is stored in registers, thus $N_{load-opr} = 2s^2 + ps$. For the case of $S_{rf} \leq \frac{s}{p}$, all the operands and intermediate data have to be reloaded in each iteration, $N_{load-opr}$ becomes $2s^2 + s$. $N_{load-tr}$ and $N_{store-tr}$ are also increased rapidly. The number of memory accesses in each case is shown in Table 3.

As shown in Table 3, in the case that s and p are fixed, N_{total} changes as various size of register files are used. Note that for the method-II S_{rf} decreases as p increases. To reduce N_{total} for a large s, one can increase S_{rf} and p. However, increasing p doesn't help in the method-I. For example, when $S_{rf} = 16$, $p = 4$ and $s = 64$, N_{total} is 16384 and 16832 for the method-I and method-II, respectively. When $p = 8$, N_{total} is reduced to 10112 for the method-II, while is still 16384 for the method-I. Figure 5 illustrates the comparison.

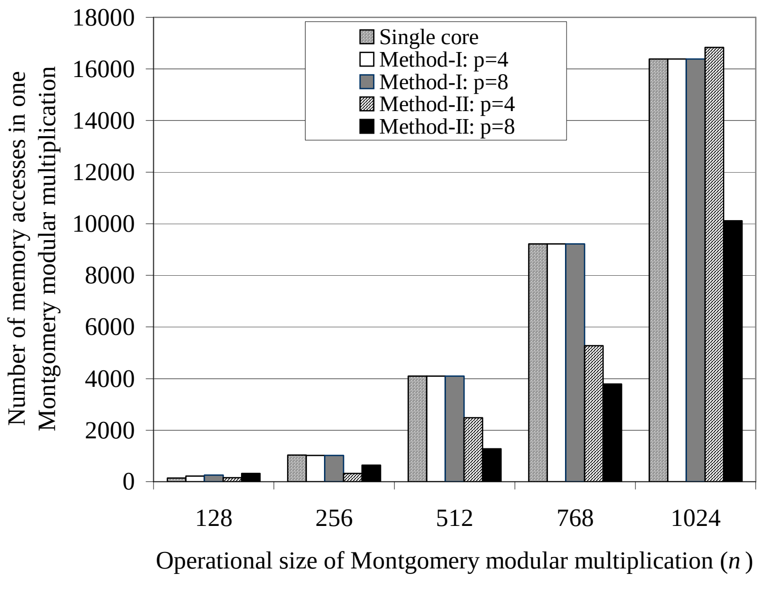


Fig. 5. Number of data memory accesses for various operand bit-length. ($w = 16$, $S_{rf} = 16$).

When the number of cores reaches a specific value, the memory access becomes the bottleneck. Because our proposed architecture uses a single-ported shared memory, load and store can not be operated in parallel. As a result, the cycles needed by one modular multiplication are no smaller than $N_{load}+N_{store}$. Let $N(s,p)$ be the number of cycles that needed by one n-bit multiplication on a p-core system. Then as p increases, there is a point where $N(s,p) = N_{load}+N_{store}$. After reaching this point, increasing p doesn't improve the performance any more. Because the method-I needs more load and store instructions than the method-II, it reaches this point before the method-II as p increases.

6 Results

The multi-core platform proposed in section 3 is implemented with GEZEL. The GEZEL code is automatically converted to synthesizable VHDL codes. The software program of Montgomery modular multiplication is stored in the instruction memory. The operands, X, Y and M, are stored in the data memory.

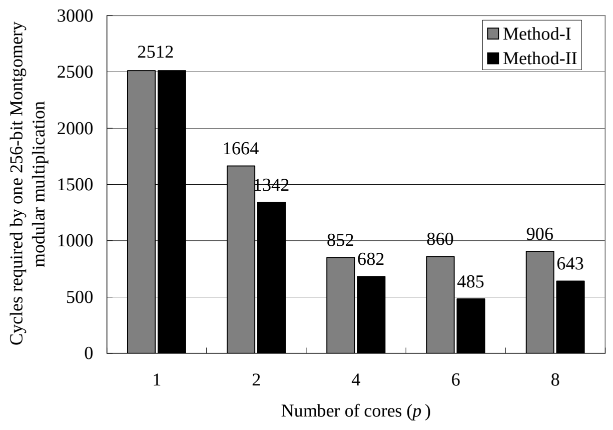


Fig. 6. Performance of 256-bit Montgomery modular multiplication on a multi-core system. ($n = 256$, $w = 16$, $S_{rf} = 16$).

We implemented both method-I and method-II on the platform with various hardware configurations. The results are presented in Figure 6. The results of the method-II show a better performance and a higher scalability in the number of cores compared to that of the method-I. If a single core is used, 2512 cycles are needed to finish one 256-bit Montgomery multiplication. When using 2 cores, 1664 and 1342 cycles are required for the method-I and the method-II, respectively. If 4 cores are used, only 852 cycles are required for the method-I, while 682 cycles are required for the method-II.

On the other hand, when employing more than 4 cores, the performance of the method-I is deteriorated because the number of the memory accesses becomes the bottleneck. For the method-II, the best performance is obtained when $p = 6$ as shown in Figure 6.

Table 4. Performance comparison of modular multiplication.

Reference	**Description**	**Platform**	**Area** (Slices)	**Freq.** (MHz)	**256-bit time**(μs)	**1024-bit time**(μs)
This work (method-I)	2-cores 2 16x16 mults	Xilinx XC2VP30	1102	125	13.3	213.0
	4-cores 4 16x16 mults	Xilinx XC2VP30	2029	125	6.8	131.0
	2-cores 2 32x32 mults	Xilinx XC2VP30	1822	93	4.5	71.6
	4-cores 4 32x32 mults	Xilinx XC2VP30	3173	93	2.6	44.0
This work (method-II)	2-cores 2 16x16 mults	Xilinx XC2VP30	1102	125	10.7	189
	4-cores 4 16x16 mults	Xilinx XC2VP30	2029	125	5.5	134.7
	2-cores 2 32x32 mults	Xilinx XC2VP30	1822	93	3.7	64.0
	4-cores 4 32x32 mults	Xilinx XC2VP30	3173	93	2.2	33.0
Tenca & Koç [6]	Software implementation	ARM processor	-	80	43	570
Cohen *et al.* [20]	Software implementation	UltraSPARC GMP library	-	143	14.6†	–
Itoh *et al.* [21]	Software implementation	DSP TMS320C6201	-	200	2.68‡	–
Brown *et al.* [22]	Software implementation	Pentium II	-	400	1.57§	–
Sakiyama *et al.* [23]	CSAs based Dual-Field	Xilinx XC2VP30	4836	110.4	0.80	–
Kelley *et al.* [24]	4-PEs 8 16x16 mults	Xilinx XC2V2000-6	360*	135	0.68	8.3
Mentens [7]	34 16x16 mults	Xilinx XC2VP30	1927	73	0.27	–
Mentens [7]	130 16x16 mults	Xilinx XC2VP30	7244	64	0.31	1.07

* Author's estimation from the original paper.
† 224-bit normal modular multiplication.
‡ 239-bit Montgomery modular multiplication.
§ Using fixed modulo for fast reduction.

For the purpose of checking the maximum frequency, the platform is implemented on Xilinx Virtex-II PRO (XC2VP30) FPGA. A maximum frequency of 125 MHz could be achieved if 16-bit cores ($w = 16$) are used. For 32-bit cores ($w = 32$), a maximum frequency of 93 MHz can be obtained. The instruction memory and the data memory are implemented in the block RAM on the FPGA board. The number of slices here only includes main controller and cores. The performance comparison between our software implementations and the state-of-the-art implementations is summarized in Table 4.

As shown in Table 4, our software implementation can achieve high speed and good scalability. Taking the method-II as an example, when using two 32-bit cores, the 256-bit modular multiplication is almost 10 times faster than the implementation on the ARM processor [6] and almost 4 times faster than the implementation on the UltraSPARC processor [20]. When using four 32-bit cores, the implementation of 256-bit modular multiplication is as fast as the implementation on TI's DSP (TMS320C6201) [21], which can issue eight 32-bit instructions in parallel. The implementation in [22] is fast, however it only supports fixed modulus. Compared to the state-of-the-art hardware implementations [23, 24, 7], software implementations are still much slower. However, hardware implementations add area and complexity to the whole system, and have far less flexibility than software implementations.

7 Conclusions

In this paper, we introduced an efficient software implementation of the Montgomery multiplication algorithm on a multi-core system. A prototype of general multi-core systems is implemented. Our newly proposed scheduling method could reduce the number of data transfers between different cores. As a result, the performance of 256-bit Montgomery multiplication was improved by a factor of 1.87 and 3.68 when using 2-core and 4-core systems, respectively.

Our future work includes a hardware implementation based on our proposed parallel-processing algorithm with a special data-path that can perform multiple arithmetic operations. A software implementation of this algorithm on commercial multi-core processors is also in progress. We believe that the scheduling method proposed in this paper can achieve high flexibility and high-performance in both software and hardware implementations.

Acknowledgments

Junfeng Fan and Kazuo Sakiyama are funded by a research grant of the Katholieke Universiteit Leuven and FWO projects (G.0450.04, G.0475.05). This work was supported in part by the IAP Programme P6/26 BCRYPT of the Belgian State (Belgian Science Policy), by the EU IST FP6 projects (SESOC and ECRYPT), by the K. U. Leuven, and by the IBBT-QoE project of the IBBT.

References

1. R. L. Rivest, A. Shamir and L. M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120-126, 1978.

2. N. Koblitz. Elliptic curve cryptosystem. *Math. Comp.*, 48:203-209, 1987.
3. V. Miller. Uses of elliptic curves in cryptography. In H. C. Williams, editor, *Advances in Cryptology: Proceedings of CRYPTO'85*, number 218 in LNCS, pages 417-426. Springer-Verlag, 1985.
4. P. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*,44:519-521,1985.
5. 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.
6. A. Tenca and Ç. K. Koç. A scalable architecture for modular multiplication based on Montgomery's algorithm. *IEEE Transactions on Computers*, 52(9):1215-1221, September 2003.
7. N. Mentens, Secure and efficient coprocessor design for cryptographic applications on FPGAs. PhD Thesis, June, 2007.
8. http://download.intel.com/products/processor/xeon/dcprodbrief.pdf
9. http://download.intel.com/products/processor/xeon/dc53kprodbrief.pdf
10. Y. Kanno, H. Mizuno, Y. Yasu, K. Hirose, Y. Shimazaki, T. Hoshi, Y. Miyairi, T. Ishii, T. Yamada, T. Irita, T. Hattori, K. Yanagisawa, and N. Irie. Hierarchical Power Distribution with 20 Power Domains in 90-nm Low-Power Multi-CPU Processor. *ISSCC Dig. Tech. Papers*, pages 540-541, Feburary 2006.
11. http://www.arm.com/products/CPUs/ARM11MPCoreMultiprocessor.html
12. C. D. Walter. Montgomery's exponentiation needs no final subtraction. *Electronic letters*, 35(21):1831-1832, October 1999.
13. Ç. K. Koç, T. Acar and B. S. Kaliski. Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro*,16:26-33,1996.
14. M. E. Kaihara and N. Takagi. Bipartite modular multiplication. *Proceedings of Cryptographic Hardware and Embedded Systems - CHES 2005*, number 3659 in Lecture notes in Computer Science, pages 201-210, September 2005. Springer-Verlag.
15. K. Iwamura, T. Matsumoto, and H. Imai. High-speed implementation methods for RSA scheme. In R. A. Rueppel, editor, *Advances in Cryptology: Proceedings of EUROCRYPT 92*, number 658 in Lecture Notes in Computer Science, pages 221-238. Springer-Verlag, 1992.
16. T. Blum and C. Paar. Montgomery modular exponentiation on reconfigurable hardware. *In Proceedings of 14th IEEE Symposium on Computer Arithmetic*, pages 70C77, Adelaide, Australia, April 14-16 1999.
17. L. Batina and G. Muurling. Montgomery in practice: How to do it more efficiently in hardware. In B. Preneel, editor, *Proceedings of RSA 2002 Cryptographers Track*, number 2271 in Lecture Notes in Computer Science, pages 40-52, San Jose, USA, February 18-22 2002. Springer-Verlag.
18. S. H. Tang, K. S. Tsui and P. H. W. Leong. Modular exponentiation using parallel multipliers. *Proceedings of the 2003 IEEE International Conference on Field Programmable Technology (FPT)*, Tokyo, 52-59. 2003
19. P. Schaumont and I. Verbauwhede, Interactive cosimulation with partial evaluation. *Proceedings of Design Automation and Test in Europe (DATE 2004)* pp. 642-647, 2004.
20. H. Cohen, A. Miyaji and T. Ono. Efficient elliptic curve exponentiation using mixed coordinates. *Asiacrypt'98*, LNCS 1514, pp. 51-65, Springer-Verlag, 1998.
21. K. Itoh, M. Takenaka, N. Torii, S. Temma, and Y. Kurihara: Fast implementation of public-key cryptography on a DSP TMS320C6201. *Proceedings of Cryptographic Hardware and Embedded Systems - CHES'99*, LNCS 1717, pp. 61-72, Springer-Verlag, 1999.

22. M. Brown, D. Hankerson, J. López and A. Menezes. Software implementation of the NIST elliptic curves over prime fields. *Topics in Cryptology, CT-RSA 2001*, LNCS 2020, pp. 250-265, Springer-Verlag, 2001.
23. K. Sakiyama, B. Preneel and I. Verbauwhede. A fast dual-field modular arithmetic logic unit and its hardware implementation. *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS 2006)*, pages 787-790, 2006.
24. K. Kelley and D. Harris. Parallelized very high radix scalable Montgomery multipliers. *Conference on Signals, Systems and Computers*, pages 1196-1200, 2005.

Cell SPEED

Dag Arne Osvik

Ecole Polytechnique Fédérale de Lausanne, Switzerland

The Cell processor used in the PlayStation3 provides the potential for computations at high speed and low cost. In this talk I will discuss how to design your programs to benefit from this new architecture.

An Efficient General Purpose Elliptic Curve Cryptography Module for Ubiquitous Sensor Networks

Leif Uhsadel, Axel Poschmann, and Christof Paar

Horst Görtz Institute for IT Security
Communication Security Group (COSY)
Ruhr-Universität Bochum, Germany
Universitätsstrasse 150
44780 Bochum, Germany
{uhsadel, poschmann, cpaar}@crypto.rub.de
www.crypto.rub.de

Abstract. In this article we present the fastest known implementation of a modular multiplication for a 160-bit standard compliant elliptic curve (secp160r1) for 8-bit micro-controller which are typically used in ubiquitous sensor networks (USN). The major part (77%) of the processing time for an elliptic curve operation such as ECDSA or EC Diffie-Hellman is spent on modular multiplication. We present an optimized arithmetic algorithm which significantly speeds up ECC schemes. The reduced processing time also yields a significantly lower energy consumption of ECC schemes. We show that a 160-bit modular multiplication can be performed in 0.37 *ms* on an 8-bit AVR processor clocked at 8 MHz. This brings the vision of asymmetric cryptography in the field of USNs with all its benefits for key-distribution and authentication a step closer to reality.

Keywords: ubiquitous sensor network, elliptic curve cryptography, secp160r1, 8-bit micro-controller, Micaz

1 Introduction

The terms *ubiquitous* and *pervasive computing* designate the penetration of our everyday life with intelligent devices. *Ubiquitous sensor networks* (USN) will play a fundamental role to enable this vision. USNs consist of many tiny and smart devices, referred to as nodes, which typically combine an 8-bit processor with memory, sensors, radio unit and power supply. The foreseen applications for USNs range from medical scenarios to agricultural, military and environmental monitoring. Since much data may be very critical (e.g., for the health of human beings in medical scenarios or safety critical monitoring) security mechanisms are required to ensure integrity, confidentiality and authenticity of the data.

USNs face major security problems because the communication is wirelessly and the devices are often easy to access. Therefore, an adversary can easily eavesdrop on communication or simply steal a node. Since sensor nodes are usually not tamper-resistant, an adversary can often read out any content that is stored on the node. Furthermore, the devices are very constrained in terms of memory, computing power, and energy supply. Since battery powered devices have a limited amount of energy, the major metric in the area of USNs is energy consumption. The lifetime of a USN is directly proportional to its energy efficiency, i.e., the less energy is consumed by applications the longer the batteries will last.

Symmetric algorithms are generally preferable to asymmetric algorithms in the field of USNs because they are more efficient in terms of energy consumption and memory requirements. However, when symmetric algorithms are used, two problems arise: (1) key distribution and (2) number of stored keys. When individual keys are used in a USN with n nodes, each node has to store $(n-1)$ keys. This has good resiliency properties but obviously scales badly and is especially unsuitable for large USNs. Moreover, perfect forward secrecy is not given after a node's key have been compromised. When one single symmetric key is used, memory requirement is greatly reduced, but at the same time this is not resilient anymore. To cope with this problem many probabilistic key distribution schemes for symmetric algorithms have been proposed [EG02, CPS03, DDHV]. In general these approaches either need pre-distributed keys, which means a higher configuration effort before deployment, or they produce much traffic, which results in higher energy consumption. Therefore, asymmetric algorithms are very valuable for key establishment and authentication in USN.

Asymmetric cryptography is often considered as being too demanding for constrained devices such as sensor nodes with an 8-bit micro-controller. However, there exist several protocols for asymmetric cryptographic algorithms for USNs. In [WKC$^+$04] Watro et al. describe public-key based protocols for USNs. In particular, they present authentication and key-agreement protocols based on RSA. The so-called TinyPK was implemented in NesC for MicaZ 8-bit micro-pocessors. However, one RSA exponentiation with a 1024-bit key needs 14.5 seconds, which is arguably not acceptable in many applications. RSA needs much longer key lengths compared to elliptic curve cryptography to achieve the same security level (1024 bit vs. 160-bit) [Res00]. Considering the limited amount of memory, computing power and energy of a typical 8-bit sensor node, it seems that ECC is a much better choice for public-key cryptography for USN than RSA. Since TinyPK is based on the more demanding RSA algorithm and was implemented in NesC, it is not surprising that this is more than one order of magnitude slower than the fastest known implementation of a point multiplication for ECC in assembly. In [GPW$^+$04] Gura et al. describe a point multiplication on a 160-bit standard curve within 0.81 seconds. The majority (77%) of the clock cycles was required by the modular multiplication. However, the source code of this implementation is not publicly available, it is rather intellectual property of Sun Microsystems. Therefore, these impressive results are not usable for the scientific community. Alternatively there is the TinyECC implementation [LN06], which may be used free of charge. TinyECC is a free software package for TinyOS that supports all SECG recommended 128-bit, 160-bit and 192-bit elliptic curve domain parameters. However, it is slower and needs more memory than the equivalent of SUN Microsystems. Therefore, our goal was to implement a prime field arithmetic for an ECC scheme for 8-bit micro-processors, which is free for use and at the same time faster than the aforementioned implementation of SUN. The source code is available on request.

The remainder of this work is organized as follows: In Section 2 we give an introduction to elliptic curve cryptography and constraints of the target devices. Subsequently, in Section 3 our implementation of the modular multiplication for a 160-bit standard elliptic curve is described. The results of our implementation are presented in Section 4. Finally, this paper is concluded in Section 5.

2 Preliminary Assumptions and Introduction to Elliptic Curve Cryptography

In this section, we first state the constraints of the target micro-processor. Subsequently we introduce the mathematical background of ECC. Finally, we state the implementation issues that arise when trying to implement ECC for constrained devices.

2.1 Constrained Devices

For the envisioned applications of USNs, up to tens of thousands of smart, but battery powered devices are required, which communicate wirelessly. In order to lower costs, these devices will be very constrained in terms of memory capacity, computing power and energy supply. Nowadays, the de-facto standard sensor nodes for researchers are the so-called Mica motes [xbo,HC02]. They comprise an 8-bit RISC ATMEL AVR ATmega128L [Atm] micro-processor, 4 KB configuration EEPROM memory, 512 KB data Flash memory, 128 KB program Flash memory, various sensors, ZigBee radio interface, and two standard AA batteries. Ideally these batteries should last for several months up to years. Therefore, a small power consumption is a crucial requirement for any application running on these nodes. Sending and receiving of messages is by far the most energy consuming task on the nodes [HSW$^+$00], therefore the traffic should be minimized wherever possible. Furthermore, the energy consumption of an application is mainly determined by its execution time. Therefore, a rule-of-thumb is: the shorter the processing time of an algorithm, the lower its energy consumption.

2.2 Introduction to Elliptic Curve Cryptography

Compared with symmetric algorithms the asymmetric algorithms work very slow. In particular on low-power processors they are felt as not practical and are used only rarely or not at all. For this purpose special algorithms were developed, but more research is needed to investigate their security before they can be used to protect sensitive data. Elliptic curves represent a special case. The advantage of the Elliptic Curve Cryptography (ECC) is that on one hand it is meanwhile quite well investigated and thus considered secure while on the other hand just a very short bit length is needed as compared to other asymmetric systems. In order to reach a security level, which is equivalent to an RSA key with a length of 1024-Bit, already 160 bits are sufficient with elliptic curves [Res00]. This is a ratio of 6.4 and will significantly reduce the consumed energy for key establishment.

Let E be an elliptic curve defined over a field K as shown in figure 1, then a set of points can be created by a *chord-and-tangent rule* (extended addition). If P and Q are two different points, which are part of the set, that intersect the elliptic curve in a straight line, there will be a third intersection on the straight line with the curve. The reflection on the x axis of the latter is called R and represents the sum of P and Q. Doubling works the same, but the straight line is given by the tangent of the curve in the according point. This set of points defined by the extended addition extended by the point ∞ forms an Abelian group.

$P + P$ is referred to as $2P$. Accordingly is $P + .. + P = kP$. For every point P there exists a point Q with $P = kQ$, if P is not the identity and the order of the elliptic curve is prime. Finding the appropriate k for a given set (Q, P) is considered to be hard and called the *elliptic curve discrete logarithm problem* (ECDLP). Most ECC protocols rely on the ECDLP.

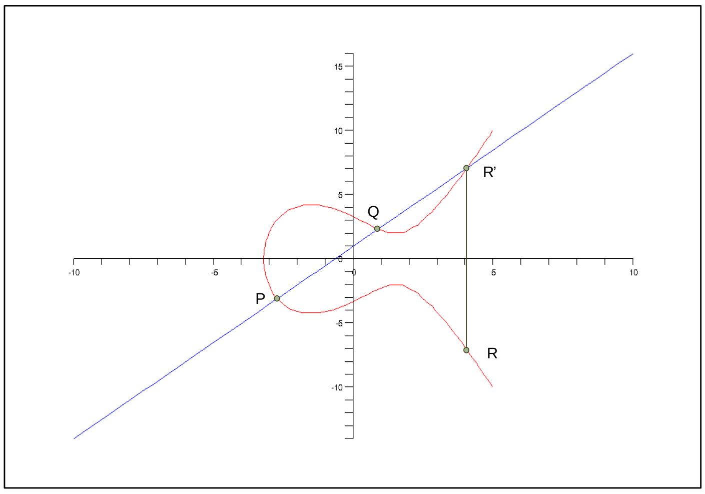


Fig. 1. Elliptic Curve, Parameters: a=-7 and b=11

There are various algorithms for the extended addition on an elliptic curve for different coordinates and different underlying fields. They can be optimized according to the used protocol and hardware. A good overview is given by [HMV04] and [Mic01]. Regardless which algorithm is used, they are all based on the arithmetic of the underlying field. Especially the multiplication in the field comes at great cost in time and energy. An efficient field arithmetic is therefore the base for an efficient implementation of an elliptic curve cryptographic system.

As prime fields are promising candidates for software implementations, we rely in the following on elliptic curves of the form

$$E/K : y^2 = x^3 + ax + b, char(K) \neq 2, 3 \tag{1}$$

2.3 Elliptic Curve Cryptography Implementation Issues

The basis for an efficient cryptographic system based on elliptic curves is a very efficient prime field arithmetic. As shown in Figure 2, a cryptographic system based on elliptic curves can be divided into three layers. The highest level actually represents the application layer. Protocols implemented here are for example ECDSA [HMV04] or EC ElGamal [HMV04]. Optimizations in this layer vary strongly, depending on the application (signature, coding etc.) and have to be partly or completely redone for each application. The underlying layer is the arithmetic of the elliptic curve. Most protocols are based on the multiplication of a point on the elliptic curve with an integer $(k * P)$. However, optimizations at this level usually also strongly depend on the protocol layer. Optimizations in the underlying prime field arithmetics layer will always improve the performance of the whole ECC-System, because they are layer independent. More than 77% of the computing time can be applied here. Therefore, a very efficient prime field arithmetic is crucial for ECC based systems on constrained devices and time critical systems.

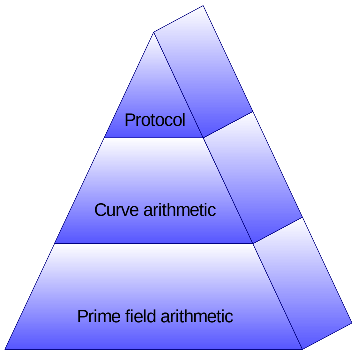


Fig. 2. Three Layers of an ECC-system

3 Implementation of Modular Multiplication

In this section, we first state criterias for an efficient implementation of an ECC system. Subsequently we will present details of our implementation of the modular multiplication, on which ECC system are based.

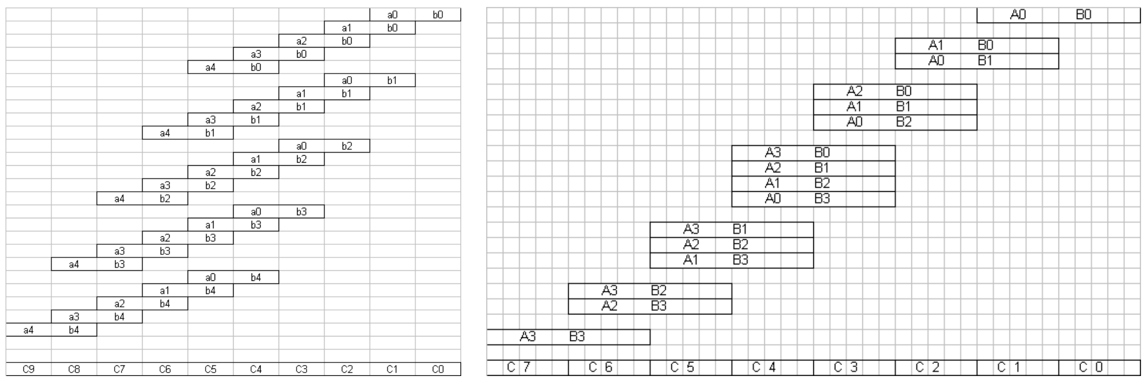
3.1 Criteria for an Efficient ECC Implementation

Since optimizations in the prime fields arithmetic, contrary to other optimizations, will always improve the performance of the ECC system, the main attention goes here. Further optimization should be done depending on the application and the selected EC domain parameters. Prime field arithmetic should provide the operations multiply, add, subtract, halve and reduction. Operations with the most potential for optimization are the multiplication and the reduction. Starting point for the implementation is to choose a curve. For compatibility reasons it should be a standardized curve and for security reasons it should have at least 160-bit in length. To keep computations fast the bit length should be as short as possible. The curve "secp160r1" standardized by *Standards for Efficient Cryptography (SEC2)* [Cer00] was chosen for our implementation. It has two advantages that can be used to speed up prime field arithmetic reduction and to speed up curve arithmetic double and add. Because its underlying prime field is based on a pseudo Mersenne prime the reduction in the prime field can be done by several shifts and adds [Sol99] which is much faster than any other known algorithm on constrained devices. The curve parameter $a = -3$ can be used to reduce the effort of point doubling and point addition when using Jacobian projective coordinates [HMV04].

To adapt the algorithms in the best possible way to the hardware the prime field arithmetic is completely implemented in assembly. As mentioned before the reduction can be implemented very efficiently if pseudo Mersenne primes are used. Addition and subtraction can be done without special optimization. The highest cost of computation lies in the 160-bit multiplication of the prime field. When choosing an algorithm for this multiplication it is important to consider the hardwares characteristic, such as processor word-size and number of general purpose registers. The ATmega128L is able to perform an 8 bit multiplication in two cycles. Loading one 8-bit word from SRAM to registers also requires two cycles. Basically two different approaches are possible: reduce the number of multiplication or reduce SRAM usage. The first attempt would be to implement

Karatsuba [MVPV96] and the second some kind of *improved schoolbook* algorithm. The hybrid multiplication [GPW$^+$04] is a memory optimized variant of the schoolbook algorithm. A special characteristic of the algorithm is that the computational cost rises linearly with smaller numbers of registers and processor word size. It also is much easier to implement than Karatsuba and hence much easier to port to different platforms. For these reasons the hybrid multiplication was chosen.

When doing a multiplication using the schoolbook algorithm the multiplication is divided in several parts that are accumulated to get the final result. The summands can be sorted in two ways before the addition. Adding them from left-to-right or right-to-left[1] it is called row wise multiplication, see Figure 3(a). Sorting them by bit length is called column wise multiplication, see



(a) Row Wise Multiplication (b) Column Wise Multiplication

Fig. 3. Row Wise and Column Wise Multiplication

Figure 3(b). The hybrid multiplication algorithm [GPW$^+$04] combines both methods: the summands that are used in the column wise way are calculated by using the row wise method, see Figure 4. The number of partial products per row is called *column width* (d). According to [GPW$^+$04] the optimal column width is:

$$d = \max\{i \mid 1 \leq i \geq n, r \geq 3i + \lceil \log_2 (n/i)/k \rceil\}, \tag{2}$$

where n is the operand size, r are the available registers and k is the bitlength.

3.2 Implementation of the Modular Multiplication

According to Formula 2 the optimal d is 10 using all registers of the micro controller. In the first approach this parameter was used. The implementation benchmark showed that the implementation was about 50% slower than the benchmarks of SUN Microsystems in [GPW$^+$04]. This overhead was mainly caused by handling carry bits. Let's have a look at the theoretic minimum effort of the algorithm. The core of the row wise part is the elemental 8-bit multiplication of the CPU followed

[1] This is what is taught in school when learning the multiplication the first time - probably giving the algorithm its name

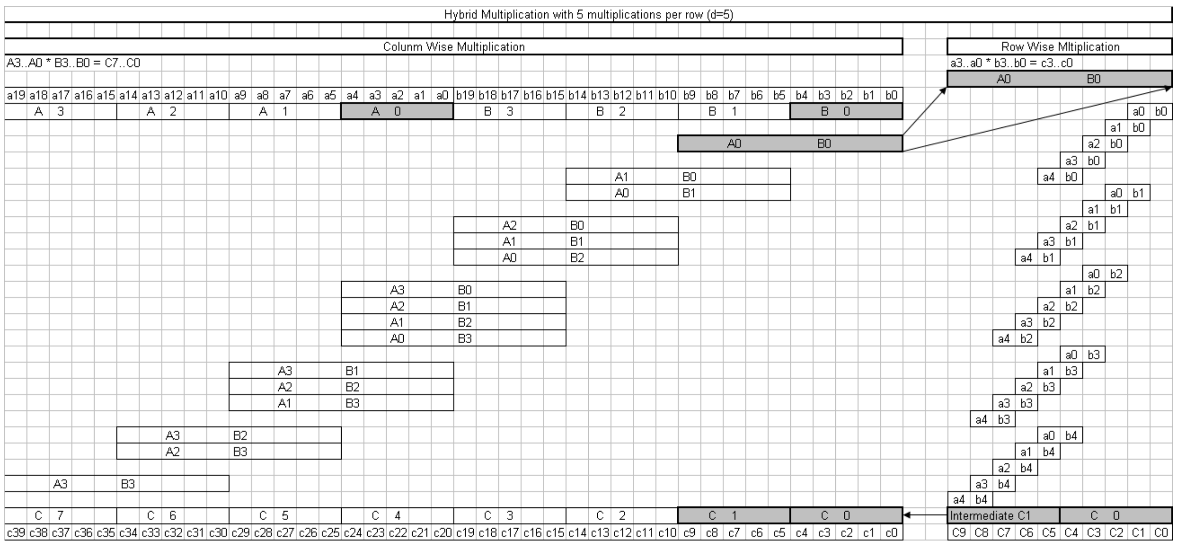
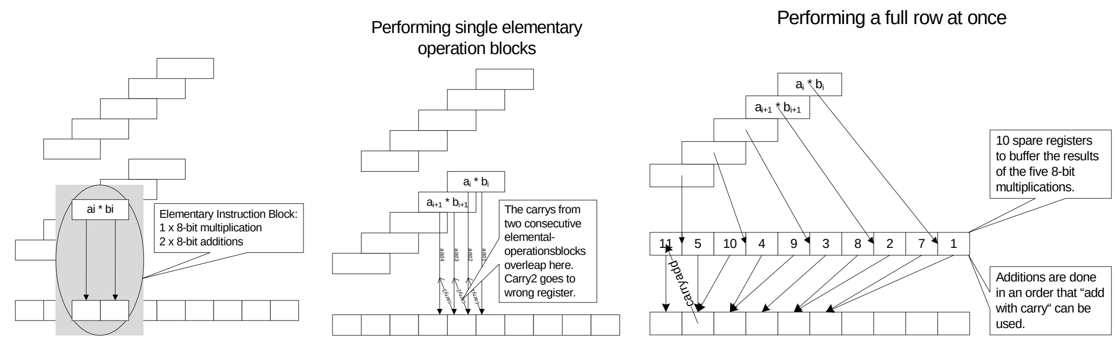


Fig. 4. 160-bit Hybrid Multiplication on ATMega128L with 5 multiplications per row

by two additions to add the product to an intermediate result. These three operations are performed in the inner loop and will be referenced as the *elementary instruction block* in the remainder as illustrated in Figure 5(a). When using 160-bit operands this is done exact 400 times regardless of d. One multiplication and two additions equal 4 cycles. This means 1600 cycles in total plus the effort to get the operands from SRAM and write them back. This effort depends on the parameter d which depends on the machine's hardware. For the theoretically best d ($d = 10$) on our target device the memory load and store effort would be 80 data loads and 40 stores consuming 240 cycles in total. For $d = 5$ the data load effort would double to 160 cycles while data store effort remains at 440 consuming 400 cycles in total. In summary, the theoretic optimum is 1840 cycles for d equal to 10 or 2000 cycles for d equal to 5. However, our first implementation needed about 4500 cycles, even though we used the – theoretically – optimal column width d of size 10.

We found that surprisingly, the major part of the overhead was caused by carry handling rather than handling pointers or other arbitrary effort. The elementary instruction block is one 8-bit multiplication followed by two additions as mentioned before. Since the additions are targeted to an intermediate result which is in general not zero the addition produces a carry bit in the general case. When the next iteration starts the elementary 8-bit multiplication will overwrite the carry flag in the CPU. Hence the carry has to be stored and restored in each elementary instruction block, which would result, in at least two additional cycles per elementary instruction block (= 3 cycles) or an overhead of at least 66.66% only for carry handling! Even if an efficient carry store and restore would be available, the operation "add with carry" would add the carry to the wrong register, as can be seen in Figure 5(b). The best solution found to solve both problems takes three cycles per iteration of each elementary instruction block, which is an overhead of 75% in the elementary instrucion block. Any other possible solution found needed more spare registers. In our second implementation the column width was chosen equal to 5 ($d = 5$), therefore doubling the number of memory loads. In other words, we trade at least 80 additional cycles for the sake of more


ai * bi
Elementary Instruction Block:
1 x 8-bit multiplication
2 x 8-bit additions
Performing single elementary operation blocks
ai * bi
ai+1 * bi+1
The carrys from two consecutive elemental-operationsblocks overleap here. Carry2 goes to wrong register.
Performing a full row at once
ai * bi
ai+1 * bi+1
10 spare registers to buffer the results of the five 8-bit multiplications.
11 5 10 4 9 3 8 2 7 1
carryadd
Additions are done in an order that "add with carry" can be used.

(a) Elementary Instruction Block (b) Column Wise (c) Row Wise at Once

Fig. 5. Core Operations in Detail

spare registers. Storing and restoring the carry bit after each 8-bit multiplication is not efficient. A solution in which the carry can be handled by the "add with carry" command is required. The number of consecutive elementary instruction blocks performed in the row wise part is set by the parameter d. In this case five iterations are done in a row. The spare registers can be used as buffer to safe the five 16-bit products of the five 8-bit multiplications. After five multiplications eleven additions are performed in the order shown in Figure 5(c)[2]. In this way the overhead for handling the carry bits is six cycles for five elementary instruction blocks. That means for one elementary instruction block the carry handling overhead is reduced to 1.2 cycles. This reduces the overhead of the elementary instruction block's effort from 75% in the first implementation to 30%. Taking one cyle as the theoretical minimum overhead this is close to the optimum. The advantage in saving both time and energy is enormous since the elementary instruction block is repeated 400 times.

4 Results

The basic requirement for a fast and thus energy efficient implementation of ECC is a very fast multiplication in the prime field. The fastest known implementation was implemented by SUN Microsystems. In [GPW+04] they provide a benchmark for the same micro-controler that we used, hence a direct comparison is possible. A 160-bit multiplication from SUN Microsystems' implementation requires 3106 cycles, which is at a clock rate of 8 MHz equivalent to 0.39 ms.

The implementation presented in this work needs 2913 cycles for a 160-bit multiplication, which is equivalent to 0.36 ms. In fact, this represents a time saving of 7.2%. To the best of our knowledge this is the fastest implementation world wide of a modular multiplication of a 160-bit standardized elliptic curve for an 8-bit micro-controller.

In Table 1 we present a detailed list of instructions used by our and SUN Microsystems' implementation as published in [GPW+04]. A third column contains the theoretical minimum amount

[2] In Figure 5(c) addition number six is represented by the – carryadd – . We call this carry add "secure" since it cannot produce another carry. This is due to the fact that 0xFF * 0xFF = 0xFE01. Hence, adding a carry to the first byte of 0xFE01 results in 0xFF01 and does not produce another carry.

of the appropriate instruction, as required by the hybrid multiplication with a column width of five on the ATMega128L micro-controller. However, this number cannot be achieved, but is mentioned to show the limit and the overhead. Each row represents an instruction or a set of instructions, which are very similar. The first row represents the 8-bit addition with and without carry. In the next row the number of 8-bit multiplications can be seen. In the following row all used data loads are combined. Thereafter the used commands to write back to SRAM are listed. The underlying row shows all 8-bit and 16-bit register moves. Finally all other instructions are combined. In this row only the number of used cycles is given while the number of instructions is missing, because different instructions may consume different number of cycles to be executed.

As one can see, the main differences between our implementation and SUN Microsystems' lie in the number of used additions and data loads. Note that data loads require two cycles contrary to the addition, which only requires one cycle. In SUN Microsystems' implementation the number of data loads is close to the minimum number of 160 data loads for a column width of 5. The additional data loads in our implementation result from pointer handling. Pointers have to be restored from SRAM very often, because the carry handling needs all spare registers. The time saving results from the improved carry handling reducing the number of needed additions close to the minimum. The overall improvement is 7.2%.

		This work		**SUN Microsystems**		**Theoretical Minimum**	
Instruction	**#C/I**	**Instructions**	**Cycles**	**Instructions**	**Cycles**	**Instructions**	**Cycles**
add/adc	1	986	986	1360	1360	800	800
mul	2	400	800	400	800	400	800
ld/lds	2	238	476	167	334	160	320
st/sts	2	40	80	40	80	40	80
mov/movw	1	355	355	335	335		
other			184		197		
Sum			**2913**		**3106**		**2000**

Table 1. Overview of instructions used

5 Conclusion and Future Work

We presented the fastest implementation of a modular multiplication for a 160-bit standardized elliptic curve for 8-bit micro-processors in Section 3 and compared the results in Section 4. We also highlighted the criteria for efficient implementations of ECC schemes for 8-bit micro-controllers and pointed out the problems that arise when implementing

Since modular multiplications take up the major part of the computing time of point multiplications over an elliptic curve, our results can be used to significantly increase the efficiency of point multiplications over an elliptic curve. Many ECC schemes such as EC ElGamal or ECDSA are based on modular multiplication and will therefore directly benefit from our results. Our results bring the vision of asymmetric cryptography in the field of USNs with all its benefits for key-distribution and authentication a step closer to reality.

Next steps are the efficient implementation of point multiplication over the elliptic curve and some ECC schemes such as EC ElGamal and ECDSA. Furthermore an integration into existing ECC modules for TinyOS is thinkable.

References

[Atm] Atmel. 8-bit Microcontroller with 128K Bytes In-System Programmable Flash. http://www.atmel.com/.

[Cer00] Certicom Research. SEC 2: Recommended Elliptic Curve Domain Parameters. Standards for Efficient Cryptography Version 1.0, September 2000.

[CPS03] H. Chan, A. Perrig, and D. Song. Random Key Predistribution Schemes for Sensor Networks. In *Proceedings of the IEEE Security and Privacy Symposium 2003*, 2003.

[DDHV] W. Du, J. Deng, Y. Han, and P. Varshney. A Pairwise Key Pre-distribution Scheme for Wireless Sensor Networks. In *CCS '03: Proceedings of the 10th ACM Conference on Computer and Communications Security.*

[EG02] L. Eschenauer and V. Gligor. A Key Management Scheme for Distributed Sensor Networks. In *CCS '02: Proceedings of the 9th ACM Conference on Computer and Communications Security*, New York, NY, USA, 2002. ACM Press.

[GPW$^+$04] N. Gura, A. Patel, A. Wander, H. Eberle, and S.C. Shantz. Comparing Elliptic Curve Cryptography and RSA on 8-bit CPUs. *Proceedings of Workshop on Cryptographic Hardware and Embedded Systems (CHES 2004), 6th International Workshop, pages 119–132*, 2004.

[HC02] JL Hill and DE Culler. Mica: a Wireless Platform for Deeply Embedded Networks. *Micro, IEEE*, 22(6):12–24, 2002.

[HMV04] Darrel Hankerson, Alfred J. Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography.* Springer, New York [u.a.], 2004.

[HSW$^+$00] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System Architecture Directions for Networked Sensors. *SIGOPS Oper. Syst. Rev.*, 34(5):93–104, 2000.

[LN06] An Liu and Peng Ning. TinyECC: Elliptic Curve Cryptography for Sensor Networks. available for download at `http://discovery.csc.ncsu.edu/software/TinyECC`, September 2006.

[Mic01] Michael Brown and Darrel Hankerson and Julio López and Alfred Menezes. Software Implementation of the NIST Elliptic Curves Over Prime Fields. *Lecture Notes in Computer Science*, 2020:250ff, 2001.

[MVPV96] A.J. Menezes, O. Van, C. Paul, and S.A. Vanstone. *Handbook of Applied Cryptography.* CRC Pr I Llc, 1996.

[Res00] Certicom Research. SEC 1: Elliptic Curve Cryptography, Version 1.0, September 2000.

[Sol99] J. Solinas. Generalized Mersenne Numbers. *Technical report CORR-39, Dept. of C&O, University of Waterloo, 1999. Available from http://www.cacr.math.uwaterloo.ca*, 1999.

[WKC$^+$04] Ronald Watro, Derrick Kong, Sue Fen Cuti, Charles Gardiner, Charles Lynn, and Peter Kruus. TinyPK: Securing Sensor Networks with Public Key Technology. In *SASN '04: Proceedings of the 2nd ACM Workshop on Security of Ad Hoc and Sensor Networks*, pages 59–64, New York, NY, USA, 2004. ACM Press.

[xbo] Crossbow Technology, Inc. http://www.xbow.com.

Optimal Irreducible Polynomials for GF(2^m) Arithmetic

Michael Scott

Dublin City University
Ballymun
Dublin
Ireland.
mike@computing.dcu.ie

Abstract. The irreducible polynomials recommended for use by multiple standards documents are in fact far from optimal on many platforms. Specifically they are suboptimal in terms of performance, for the computation of field square roots and in the application of the "almost inverse" field inversion algorithm. In this paper we question the need for the standardisation of irreducible polynomials in the first place, and derive the "best" polynomials to use depending on the underlying processor architecture. Surprisingly it turns out that a trinomial polynomial is in many cases not necessarily the best choice. Finally we make some specific recommendations for some particular types of architecture.
Keywords: Irreducible polynomials. Arithmetic in $\mathbb{F}_{2^m}$.

1 Introduction

The main application of $\mathbb{F}_{2^m}$ arithmetic (where m is an odd prime) is in elliptic curve cryptography [11], although recently it has also found constructive application in pairing-based cryptography, specifically in the implementation of the η_T pairing, which is one of the fastest known [5]. Clearly we would like the underlying $\mathbb{F}_{2^m}$ arithmetic to be implemented as efficiently as possible.

Elements in $\mathbb{F}_{2^m}$ can be represented as a multi-precision string of bits, each bit representing a coefficient in a polynomial of degree at most $(m-1)$, a so-called *polynomial basis representation.* The field is specified in conjunction with an *irreducible polynomial*, and multiplication of field elements is performed modulo this irreducible polynomial.

Addition of field polynomials is a simple coefficient-by-coefficient XOR operation. By grouping coefficients into blocks, each block the size of one computer word of w bits, the addition can be carried out using $\lceil m/w \rceil$ word XOR operations, an operation supported by all computer architectures. This can be thought of as the same as multiprecision integer addition, without the carries.

Multiplication can be performed using the standard school-boy long multiplication algorithm, where in the calculation of each partial product carries are simply ignored. The carry-free nature of this type of arithmetic can be an advantage (implementing multiplication using Karatsuba's algorithm for example is much simpler and faster), and a disadvantage (most architectures do not support carry-free word multiplications).

Squaring is much faster than multiplication – it can be achieved by simply inserting a zero between each coefficient. Multiplication or squaring both result in a polynomial of degree at most $2m - 2$, and it is this polynomial which must be reduced modulo the irreducible polynomial. And it is this reduction process which is the focus of this paper.

For security reasons the parameter m is chosen as a prime number. Certain fields are recommended for use in elliptic curve cryptography in popular standards. For example Certicom in their "Standards for Efficient Cryptography" (available from www.secg.org) recommend $m \in \{113, 131, 163, 193, 233, 239, 283, 409, 571\}$, and it is these fields that we will focus on here. They also insist on specific irreducible polynomials for each case ("... must use [these] reduction polynomials ... to encourage interoperability"), although interestingly they offer a choice of two polynomials for the case of $m = 239$, on the grounds that both have been commonly used in practise.

Unfortunately the recommended irreducible polynomials have several disadvantages

– Few of them are optimal in terms of the actual implementation of the reduction algorithm on popular architectures.
– Few are suitable for the calculation of field square roots, which is required for elliptic curve point halving operations [4], and for the η_T pairing [5].
– Many are unsuitable for use with the almost inverse algorithm for field inversion [11].

The reason for the standardisation of irreducible polynomials is that the represention of field elements depends upon it. Therefore it is important for example in a communications protocol in which field elements are transmitted from one party to another, that all participants are aware of the irreducible polynomial in use by the others. However it is a simple matter to include the parameters that define the irreducible polynomial as part of the domain information which each participant needs to aware of anyway. In this case the irreducible polynomial might be chosen to facilitate the participant with the lowest computational power, and hence in most need of an optimal choice of polynomial. Alternatively, one can change from one polynomial representation to another should that be necessary as described in [13], section A.7. The change of basis algorithm is a little awkward, and requires a precomputed $m \times m$ bit conversion matrix. It could represent a significant overhead for a poorly-resourced processor, and so should only be expected of the more powerful protocol participants.

The fact that standard elliptic curve domain parameters are described in terms of the standard polynomials presents a barrier to converting them to a more suitable polynomial, and most implementors just stick with the standard polynomials for implementation as well as for representation.

2 Irreducible polynomials

For the field $\mathbb{F}_{2^m}$ it is in practise always possible to chooose as an irreducible polynomial either a trinomial

$$x^m + x^a + 1$$

or a pentanomial

$$x^m + x^a + x^b + x^c + 1$$

In both cases the optimal reduction algorithm is linear and fast compared to the worse-than-linear multiplication algorithm. For efficiency the reduction algorithm of choice [11] requires that $m - a \geq w$.

In [8] it is stated that "performance reasons impose that irreducible polynomials have the shortest number of non-zero terms", and it appears to be a commonly held view that trinomials are better than pentanomials. As we shall see this is not necessarily the case.

The standard polynomials are selected according to the following criteria (with the exception of one of the $m = 239$ polynomials). If a trinomial exists, the trinomial with the smallest value for a is chosen. Otherwise the pentanomial is chosen with the smallest a, then the smallest b given a, and then the smallest c given a and b. Given these rules it is relatively easy to find the standard polynomial for any value of m. They are also most likely to fulfil the condition that $m-a \geq w$. There is also a widespread belief that such polynomials are also in some sense optimal. For example Ahmadi and Menezes [3] suggest a construction where the middle terms are "all of relatively low degree and are close to each other, which in turn facilitates efficient multiplication of polynomials modulo $f(x)$".

However there does not seem to be any basis for this suggestion for software implementations, although it may have merit in hardware. For example, as we shall see, the standard polynomial for $\mathbb{F}_{2^{283}}$ is far from being optimal, despite adhering to these recommendations.

Recently there have emerged applications in which it is important that square rooting should be fast, specifially point halving algorithms for fast elliptic curve point multiplication [4], [11], and the η_T pairing [5]. Fast square rooting requires that a (and b and c for a pentanomial) must all be odd [10]. Such polynomials are easy to find, but unfortunately most of the standard polynomials are not of this form [4].

It has been known for some time that the Schroeppel et al. "almost inverse" algorithm for field inversions is more efficient if $a \geq w$ for a trinomial, and $c \geq w$ for a pentanomial [14] (although in [10] a couple of strategies due to Knudsen and Schroeppel are described which can to an extent circumvent this restriction). Many of the standard irreducible polynomials do not satisfy this condition, although in practise it may be that this algorithm is not the algorithm of choice [11]. Nonetheless the authors of [14] complain that "Unfortunately, most of the field polynomials specified in ANSI X9.62 do have terms of low degree. This increases the timings of the almost inverse algorithm by up to 30%. Therefore, we conclude that the choice of polynomials in ANSI X9.62 is rather unfortunate, and may be revised if that is practically feasible.”.

A third, and hitherto largely neglected, issue is that of performance. Although the reduction algorithm is always fast compared with multiplication, it applies also to squarings. One of the advantages of using methods based on $\mathbb{F}_{2^m}$ fields is that squarings are potentially so fast. But if the reduction algorithm is slow this advantage will be offset.

3 An Example

For the field $\mathbb{F}_{2^{163}}$ the standard irreducible polynomial is $x^{163} + x^7 + x^6 + x^3 + 1$. The reduction algorithm as described by Hankerson et al. in section 2.3.5 of [11] for a 32-bit processor is illustrated in Algorithm 1.

Algorithm 1 Fast reduction modulo $f(x) = x^{163} + x^7 + x^6 + x^3 + 1$

```
INPUT: A binary polynomial represented as 10 32-bit words $g[.]$
OUTPUT: $g[.]$ reduced modulo $f(x)$, represented as 6 32-bit words
 1: for $i \leftarrow 10$ downto 6 do
 2:     $t \leftarrow g[i]$
 3:     $g[i] \leftarrow 0$
 4:     $g[i-6] \leftarrow g[i-6] \oplus (t \ll 29)$
 5:     $g[i-5] \leftarrow g[i-5] \oplus (t \ll 4) \oplus (t \ll 3) \oplus t \oplus (t \gg 3)$
 6:     $g[i-4] \leftarrow g[i-4] \oplus (t \gg 28) \oplus (t \gg 29)$
 7: end for
 8: $t \leftarrow g[5] \gg 3$
 9: $g[0] \leftarrow g[0] \oplus t$
10: $t \leftarrow (t \ll 3)$
11: $g[1] \leftarrow g[1] \oplus (t \gg 28) \oplus (t \gg 29)$
12: $g[0] \leftarrow g[0] \oplus t \oplus (t \ll 4) \oplus (t \ll 3)$
13: $g[5] \leftarrow g[5] \oplus t$
```

After the loop there is some tidying-up to be done to deal with the most significant word of the result. Concentrating on the time-critical loop, unrolling it (and omitting for clarity the rest of the algorithm) we get Algorithm 2.

Note that this requires 30 word shift operations 35 word XORs, which can be considered as 5 times the 6 shifts and 7 XORs required by each iteration of the loop.

Now consider a different irreducible polynomial, $f(x) = x^{163} + x^{99} + x^{97} + x^3 + 1$. The equivalent algorithm is shown in Algorithm 3.

In both cases the number of memory reads and writes are the same. However it is clear from inspection alone that this last algorithm will be faster. In fact it requires only 20 word shifts and 30 XORs, or 5 times the 4 shifts and 6 XORs required by each iteration of the loop. To understand the reasons for

Algorithm 2 Loop-unrolled reduction modulo $f(x) = x^{163} + x^7 + x^6 + x^3 + 1$

```
INPUT: A binary polynomial represented as 10 32-bit words $g[.]$
OUTPUT: $g[.]$ partially reduced modulo $f(x)$, represented as 6 32-bit words
 1: $g_{10} \leftarrow g[10],\ g_9 \leftarrow g[9],\ g_8 \leftarrow g[8],\ g_7 \leftarrow g[7],\ g_6 \leftarrow g[6]$
 2: $g[10] \leftarrow g[9] \leftarrow g[8] \leftarrow g[7] \leftarrow g[6] \leftarrow 0$
 3: $g_6 \leftarrow g_6 \oplus (g_{10} \gg 28) \oplus (g_{10} \gg 29)$
 4: $g[5] \leftarrow g[5] \oplus (g_{10} \gg 3) \oplus (g_{10} \ll 4) \oplus (g_{10} \ll 3) \oplus g_{10} \oplus (g_9 \gg 28) \oplus (g_9 \gg 29)$
 5: $g[4] \leftarrow g[4] \oplus (g_{10} \ll 29) \oplus (g_9 \gg 3) \oplus (g_9 \ll 4) \oplus (g_9 \ll 3) \oplus g_9 \oplus (g_8 \gg 28) \oplus (g_8 \gg 29)$
 6: $g[3] \leftarrow g[3] \oplus (g_9 \ll 29) \oplus (g_8 \gg 3) \oplus (g_8 \ll 4) \oplus (g_8 \ll 3) \oplus g_8 \oplus (g_7 \gg 28) \oplus (g_7 \gg 29)$
 7: $g[2] \leftarrow g[2] \oplus (g_8 \ll 29) \oplus (g_7 \gg 3) \oplus (g_7 \ll 4) \oplus (g_7 \ll 3) \oplus g_7 \oplus (g_6 \gg 28) \oplus (g_6 \gg 29)$
 8: $g[1] \leftarrow g[1] \oplus (g_7 \ll 29) \oplus (g_6 \gg 3) \oplus (g_6 \ll 4) \oplus (g_6 \ll 3) \oplus g_6$
 9: $g[0] \leftarrow g[0] \oplus (g_6 \ll 29)$
```

Algorithm 3 Loop-unrolled reduction modulo $f(x) = x^{163} + x^{99} + x^{97} + x^3 + 1$

```
INPUT: A binary polynomial represented as 10 32-bit words $g[.]$
OUTPUT: $g[.]$ partially reduced modulo $f(x)$, represented as 6 32-bit words
 1: $g_{10} \leftarrow g[10],\ g_9 \leftarrow g[9],\ g_8 \leftarrow g[8],\ g_7 \leftarrow g[7],\ g_6 \leftarrow g[6]$
 2: $g[10] \leftarrow g[9] \leftarrow g[8] \leftarrow g[7] \leftarrow g[6] \leftarrow 0$
 3: $g_8 \leftarrow g_8 \oplus g_{10} \oplus (g_{10} \gg 2)$
 4: $g_7 \leftarrow g_7 \oplus (g_{10} \ll 30) \oplus g_9 \oplus (g_9 \gg 2)$
 5: $g_6 \leftarrow g_6 \oplus (g_9 \ll 30) \oplus g_8 \oplus (g_8 \gg 2)$
 6: $g[5] \leftarrow g[5] \oplus (g_{10} \gg 3) \oplus g_{10} \oplus (g_8 \ll 30) \oplus g_7 \oplus (g_7 \gg 2)$
 7: $g[4] \leftarrow g[4] \oplus (g_{10} \ll 29) \oplus (g_9 \gg 3) \oplus g_9 \oplus (g_7 \ll 30) \oplus g_6 \oplus (g_6 \gg 2)$
 8: $g[3] \leftarrow g[3] \oplus (g_9 \ll 29) \oplus (g_8 \gg 3) \oplus g_8 \oplus (g_6 \ll 30)$
 9: $g[2] \leftarrow g[2] \oplus (g_8 \ll 29) \oplus (g_7 \gg 3) \oplus g_7$
10: $g[1] \leftarrow g[1] \oplus (g_7 \ll 29) \oplus (g_6 \gg 3) \oplus g_6$
11: $g[0] \leftarrow g[0] \oplus (g_6 \ll 29)$
```

the speed up, consider the constants that arise in the shift operations. These are the numbers $m \bmod w$, $w - (m \bmod w)$, $m - a \bmod w$, $w - (m - a \bmod w)$, $m - b \bmod w$, $w - (m - b \bmod w)$, $m - c \bmod w$ and $w - (m - c \bmod w)$. So for the standard polynomial these values in pairs are (3, 29), (28, 4), (29, 3), and (0, 32) respectively. For the fortuitous values of 0 and 32 that arise in this case, the former implies no shifting, and the latter, a shift by the word length, results in a zero. This piece of luck results in a saving of 1 XOR and 2 shifts per loop iteration.

Our selection of irreducible polynomial is based on the idea of "making our own luck" in the choice of irreducible polynomial. In our case the shift values are (3, 29), (0, 32), (30, 2), and (0, 32) and this explains the savings achieved. Given that m is odd our chances of being lucky like this can only improve if we insist that a, b and c are also all chosen to be odd.

We note that this same idea is also alluded to in recent papers by Ahmadi et al [1], section 4.2, and by Hankerson and Rodríguez-Henríquez [12], who also independently suggested the use of the polynomial $x^{163} + x^{99} + x^{97} + x^3 + 1$.

4 Further analysis

We define a trinomial where $m - a \equiv 0 \bmod w$ as a *lucky* trinomial (LT), and a trinomial for which this condition does not hold as an *ordinary* trinomial (OT). We define a pentanomial where $m - a \equiv m - b \equiv m - c \equiv 0 \bmod w$ as a *lucky* pentanomial (LP). A pentanomial for which two out of three of these values is congruent to zero is called a *fortunate* pentanomial (FP), the rest are *ordinary* pentanomials (OP). In general, as we will see, a lucky trinomial beats a lucky pentanomial, which beats an ordinary trinomial, which in turn beats a fortunate pentanomial. Whereas a lucky trinomial is relatively rare, a lucky pentanomial can often be found, and hence in many cases the pentanomial is superior to a trinomial.

Is an irreducible lucky pentanomial always possible? The answer is no – an irreducible lucky pentanomial is not possible for any wordlength w which is a multiple of 4 if $m \equiv \pm 3 \bmod 8$, as a direct consequence of a theorem of Buhler [6], [2]. However under the same circumstances useful irreducible trinomials do not exist either [7], and so we must be satisfied with a fortunate pentanomial (if we can find one).

When lucky pentanomials do exist for $m \equiv \pm 1 \bmod 8$, we get the added bonus on a small eight bit processor that the shifts by $m \bmod w$ and $w - m \bmod w$, are in fact shifts by 1-bit and 7-bits (or visa versa), which are likely to be efficient.

Note that these definitions are word-length dependant. So whereas a lucky trinomial might exist for one wordlength, it may not be so lucky for a larger wordlength.

Finally we should point out that there is another way to be lucky. It is possible that for example $m - a \equiv m - b \bmod w$. These means that the same shifted values can be re-used, with some savings.

5 Some real-world architectures

Of course the significance of all these potential savings depends on the particular computer architecture. In this paper we consider four representative real-world examples.

– A 32-bit Pentium or MIPS type of processor, which has fast 1-cycle XOR and shift instructions. We ignore the fact that these architectures often support multiple pipelines – we will assume that execution time is simply preportional to the total number of such instructions (but given the complexity of this architecture and its many variations, we recognise that this may be a rather reckless assumption).
– A 32-bit ARM processor. This architecture supports a “barrel shifter”, and an XOR instruction for example can at no extra cost shift one of its operands by any number of bits. Therefore shifts are effectively free on this architecture.
– A Texas Instruments ultra-low power msp430 16-bit processor, as used in wireless sensor networks. This architecture supports a 1 cycle XOR instruction, but has only a 1-bit shift instruction. Therefore shifts by multiple bits require multiple instructions. Fortunately the instruction set does allow a 1-cycle byte swap within a register, and simple masking instructions, which means that for example an 8-bit left shift of a 16-bit register can be accomplished in 2 instructions.
– An Atmel Atmega128 low power 8-bit processor, another favourite for applications in wireless sensor networks. Again a 1-cycle 1-bit shift operation is supported, along with a 1-cycle nibble swap within a register, as well as register masking.

To find the optimal irreducible polynomial in any particular circumstance, we cost each XOR and shift operation appropriately. An XOR always has a cost of 1. A shift may have a cost of zero (for the ARM), or a larger cost. In the case of the msp430 processor the costs are given in Table 1.

On this architecture right shifts can be slightly more expensive than left shifts, as all rotates and right shifts are through the carry flag, and this must be cleared to obtain a logical right shift as required here. However this only applies to the first of a sequence of right shifts. Note that if the algorithm needs a left-shift by 3, and also a left shift of the same value by 4 within a single iteration, then we assume a compiler will be smart enough to shift once by 3 and follow that by a further shift by 1 bit. A shift left by 15 bits is best achieved by a rotate right followed by a masking. Similar methods can be deployed for the Atmel 8-bit processor, and the costs of shifts for this processor are given in Table 2.

6 Results

For each candidate irreducible trinomial and pentanomial we calculate the associated costs and we select the cheapest, depending on the cost function that

Table 1. Shift cost in clock cycles for msp430 16-bit processor

Shift size	Left shift	Right shift
1	1	2
2	2	3
3	3	4
4	4	5
5	5	6
6	6	7
7	5	5
8	2	2
9	3	3
10	4	4
11	5	5
12	6	6
13	5	5
14	4	4
15	3	3

Table 2. Shift cost in clock cycles for Atmega128 8-bit processor

Shift size	Left shift	Right shift
1	1	1
2	2	2
3	3	3
4	2	2
5	3	3
6	4	4
7	3	3

applies for that architecture. Recall that an irreducible polynomial which is good for a 16-bit processor may not be so good for a 32-bit processor, for example $(m - a)$ may be a multiple of 16, but not of 32. In some cases the outcome is not clear-cut, as it would depend for example on whether or not there would be an intention to use the "almost inverse" algorithm. In the case of more than one solution with the same cost, we favour the solution that is friendly for the "almost inverse" algorithm. We do however insist that all exponents in the irreducible polynomial are odd, given the recently realised significance of field square roots. In any case, in no instances were optimal polynomials found with any even exponents.

Some notable outcomes

- In some cases a pentanomial may be cheaper than a trinomial. In fact for the msp430 arhitecture a trinomial is never optimal for the fields considered here. However some algorithms (other than the reduction algorithm) may be more efficient with a trinomial, so there may still be reasons for preferring a trinomial if one should exist. For the Atmel 8-bit processor, trinomials make something of a come-back, as for the first time we find some lucky trinomials.
- The "folklore" requirement that for a pentanomial irreducible polynomial the middle terms (a, b and c) being close to one another and small leads to a more efficient algorithm, does not appear to have any validity.

Consider for example the field $\mathbb{F}_{2^{233}}$. This supports the trinomial $x^{233}+x^{159}+1$. However for the Pentium cost model the pentanomial $x^{233}+x^{201}+x^{105}+x^9+1$ is superior. Note that (233-201), (233-105) and (233-9) are all multiples of 32. Whereas the trinomial requires 4 XORS and 4 shifts per loop iteration, the pentanomial costs 5 XORS and only 2 shifts. However for the ARM model the trinomial is still superior as shifts are free. For the msp430 an optimal polynomial is $x^{233} + x^{185} + x^{121} + x^{105} + 1$. In this case we do find a solution which also accomodates the "almost inverse" algorithm. Note that in no cases are the middle terms particularly close to one another.

Avanzi [4] suggests choosing the square-root friendly irreducible polynomials with odd exponents and the least degree sediment, that is with the smallest size of a. However there seems to be no compelling reason for doing so. For the $m = 163$ case the cost as we calculate it is even greater with this choice than that for the standard polynomial.

7 Redundant Trinomials

In a recent paper Doche [9], building on earlier work by Brent and Zimmerman [7], has suggested the use of redundant trinomials in place of pentanomials. However our surprising observation that pentanomials can actually be faster than trinomials rather undermines the basis of their results. Certainly for applications in cryptography the premise [7] that pentanomials are "considerably more expensive in applications" is not supported. Nevertheless there may be cases

Table 3. Optimal irreducible polynomials for Pentium-type 32-bit processor

m	Optimal polynomial	Cost of this polynomial	standard polynomial	Type
113	$x^{113}+x^{15}+1$	8	8	OT
131	$x^{131}+x^{97}+x^{65}+x^{3}+1$	11	13	OP
163	$x^{163}+x^{99}+x^{97}+x^{3}+1$	10	13	FP
193	$x^{193}+x^{73}+1$	8	8	OT
233	$x^{233}+x^{201}+x^{105}+x^{9}+1$	7	8	LP
239	$x^{239}+x^{207}+x^{111}+x^{47}+1$	7	8	LP
283	$x^{283}+x^{249}+x^{219}+x^{27}+1$	10	16	FP
409	$x^{409}+x^{377}+x^{185}+x^{57}+1$	7	8	LP
571	$x^{571}+x^{507}+x^{475}+x^{417}+1$	10	16	FP

Table 4. Optimal irreducible polynomials for ARM-type 32-bit processor

m	Optimal polynomial	Cost of this polynomial	standard polynomial	Type
113	$x^{113}+x^{15}+1$	4	4	OT
131	$x^{131}+x^{99}+x^{97}+x^{95}+1$	7	7	OP
163	$x^{163}+x^{99}+x^{97}+x^{3}+1$	6	7	FP
193	$x^{193}+x^{73}+1$	4	4	OT
233	$x^{233}+x^{159}+1$	4	4	OT
239	$x^{239}+x^{203}+1$	4	4	OT
283	$x^{283}+x^{249}+x^{219}+x^{27}+1$	6	8	FP
409	$x^{409}+x^{87}+1$	4	4	OT
571	$x^{571}+x^{507}+x^{475}+x^{417}+1$	6	8	FP

Table 5. Optimal irreducible polynomials for msp430 16-bit processor

m	Optimal polynomial	Cost of this polynomial	standard polynomial	Type
113	$x^{113}+x^{97}+x^{65}+x^{33}+1$	10	13	LP
131	$x^{131}+x^{115}+x^{81}+x^{67}+1$	16	22	FP
163	$x^{163}+x^{131}+x^{129}+x^{115}+1$	16	22	FP
193	$x^{193}+x^{145}+x^{129}+x^{113}+1$	10	12	LP
233	$x^{233}+x^{185}+x^{121}+x^{105}+1$	13	16	LP
239	$x^{239}+x^{207}+x^{111}+x^{47}+1$	9	13	LP
283	$x^{283}+x^{225}+x^{203}+x^{107}+1$	17	33	FP
409	$x^{409}+x^{377}+x^{185}+x^{57}+1$	13	19	LP
571	$x^{571}+x^{507}+x^{475}+x^{417}+1$	17	31	FP

Table 6. Optimal irreducible polynomials for Atmega128 8-bit processor

m	Optimal polynomial	Cost of this polynomial	standard polynomial	Type
113	$x^{113} + x^{9} + 1$	7	7	LT
131	$x^{131} + x^{115} + x^{81} + x^{67} + 1$	13	15	FP
163	$x^{163} + x^{131} + x^{129} + x^{115} + 1$	13	17	FP
193	$x^{193} + x^{73} + 1$	7	11	LT
233	$x^{233} + x^{185} + x^{121} + x^{105} + 1$	9	12	LP
239	$x^{239} + x^{207} + x^{111} + x^{47} + 1$	9	12	LP
283	$x^{283} + x^{249} + x^{219} + x^{27} + 1$	13	20	FP
409	$x^{409} + x^{377} + x^{185} + x^{57} + 1$	9	11	LP
571	$x^{571} + x^{507} + x^{475} + x^{417} + 1$	13	21	FP

where a redundant trinomial (or indeed pentanomial) may be superior to the irreducible polynomials suggested here. For example, for the trickly $m = 163$ case the redundant trinomial $x^{171} + x^{101} + 1$ is superior in most cases (not however for the 8-bit processor, where the length of the representation of field elements would increase by one byte, probably cancelling any potential gains).

In practice any small improvement possible with redundant trinomials may be more than offset by the extra complications involved in field element comparisons, inversions and by the requirement for possibly one or more extra computer words to be added to the field representation [9]. Note our extra condition that m and a be odd further constrains the choice of redundant trinomials, so many of the solutions presented in [9] are unsuitable.

8 Conclusions

In the light of recent developments the irreducible polynomials as recommended in some standards are in urgent need of an overhaul. In fact we would argue that the choice of irreducible polynomial should be left to the implementor, and that issues that arise from the use of different irreducible polynomials by communicating parties should be dealt with in some other way other than through standardisation.

We have derived polynomials that, when used for modular reduction, are in all cases at least as fast (and often much faster) than those suggested in the standards. We have also described a simple methodology for determining the best polynomial to use in any given circumstance, using a simple and easy to develop costing model. All of our suggested polynomials support a very fast field square root operation, and there seems to be no good reason not to use a square-root friendly polynomial in all cases.

Alternatively, based on the method described here, it might be possible to come up with good compromise irreducible polynomials, which while not necessarily being optimal in all cases, would nevertheless be an improvement on the current standards. As can be seen from the tables above, all agree on the optimal irreducible polynomial for the case $m = 571$, and there are other examples of majority agreement for other values of m.

9 Acknowledgments

Thanks are due to Darrel Hankerson and an anonymous referee of an early draft who pointed out to me the work of Brent and Zimmerman [7] and Doche [9].

References

1. O. Ahmadi, D. Hankerson, and A. Menezes. Software implementation of arithmetic in f_{3^m}. CACR Technical Reports, 2007. `http://www.cacr.math.uwaterloo.ca/techreports/2005/cacr2007-15.pdf`.
2. O. Ahmadi and A. Menezes. Irreducible polynomials of maximum weight. CACR Technical Reports, 2005. `http://www.cacr.math.uwaterloo.ca/techreports/2005/cacr2005-01.pdf`.
3. O. Ahmadi and A. Menezes. On the number of trace-one elements in polynomial bases for GF(2^m). *Designs, Codes and Cryptography*, 37:493–507, 2005.
4. R. Avanzi. A note on square roots in binary fields. Cryptology ePrint Archive, Report 2007/103, 2007. `http://eprint.iacr.org/2007/103`.
5. P.S.L.M. Barreto, S. Galbraith, C. OhEigeartaigh, and M. Scott. Efficient pairing computation on supersingular abelian varieties. *Designs, Codes and Cryptography*, 42:239–271, 2007. `http://eprint.iacr.org/2004/375`.
6. A. W. Bluher. A Swan-like theorem. *Finite Fields Appl.*, 12:128–138, 2006.
7. R. Brent and P. Zimmerman. Algorithms for finding almost irreducible and almost primitive trinomials. Proceedings of a conference in honour of Professor H.C. Williams, 2003. `http://web.comlab.ox.ac.uk/oucl/work/richard.brent/pd/rpb212.pdf`.
8. M. Ciet, J. J. Quisquater, and F. Sica. A short note on irreducible trinomials in binary fields. Proceedings of the 23rd Symposium on Information Theory in the Benelux, 2002. `citeseer.ist.psu.edu/559928.html`.
9. C. Doche. Redundant trinomials for finite fields of characteristic 2. In *ACISP 2005*, volume 3574 of *Lecture Notes in Computer Science*, pages 122–133. Springer-Verlag, 2005.
10. K. Fong, D. Hankerson, J. López, and A. Menezes. Field inversion and point halving revisited. Technical report CORR 2003-18, University of Waterloo, 2002.
11. D. Hankerson, A. Menezes, and S. Vanstone. *Guide to Elliptic Curves Cryptography*. Springer, 2004.
12. D. Hankerson and F. Rodríguez-Henríquez. Parallel formulation of scalar multiplication on koblitz curves. CACR Technical Reports, 2007. `http://www.cacr.math.uwaterloo.ca/techreports/2005/cacr2007-18.pdf`.
13. IEEE Computer Society, New York, USA. *IEEE Standard Specifications for Public-Key Cryptography – IEEE Std 1363:2000*, 2000. `http://grouper.ieee.org/groups/1363`.
14. P. De Win, S. Mister, B. Preneel, and M. Wiener. On the performance of signature schemes based on elliptic curves. In *ANTS, 3rd International Symposium*, volume 1423 of *Lecture Notes in Computer Science*, pages 252–266. Springer-Verlag, 1998.

Computer Aided Cryptographic Engineering

Daniel Page

University of Bristol, UK

In developing cryptographic software, most of us are implicitly dependent on tools that translate our programs into an executable form; experience shows that the translation process can heavily influence features such as efficiency and security. Maximising these features while simultaneously minimising programmer effort is clearly desirable. As a result, an interesting research question is how best to design domain-specific languages and development tools that support the description and implementation of cryptographic software. The aim of this talk will be to highlight existing development tools (which are perhaps unknown but useful) and to overview new research at the University of Bristol into a cryptographic-aware language and compiler.