Elliptic Curve Cryptography on Modern Processor Architectures

Neil Costigan

B.Sc., M.Sc.

A thesis submitted for the degree of

Ph.D.

to the

Dublin City University

Faculty of Engineering and Computing

School of Computing

Supervisor: Prof. Michael Scott

June, 2009

Declaration

I, Neil Costigan, hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of Ph.D. is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

Name: Neil Costigan.
ID: 52165809.
Date: June, 2009.

The original work in this thesis is as follows:

1. Chapter 2 is my own work derived from my MSc. thesis.

2. Chapter 4 is joint work with Scott. This was part of a C class project at The Irish Centre for High-End Computing (ICHEC) 2007. Presented at ICHEC Seminar 2008 Frontiers in Computational Science.

3. Chapter 3 is joint work with Scott and Abdulwahab. Presented at the Workshop on Cryptographic Hardware and Embedded Systems 2006 (CHES 2006) Yokohama, Japan.

4. Chapter 5 is my own work.

5. Chapter 6 is joint work with Scott. Presented at the Workshop on Software Performance Enhancement for Encryption and Decryption 2007 (SPEED 2007) Amsterdam, the Netherlands. Updated and presented at Special-purpose Hardware for Attacking Cryptographic Systems (SHARCS 2007), and to appear in the Proceedings of the 9th International Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA 2008) Trondheim, Norway.

6. Chapter 7 is my own work.

7. Chapter 8 was joint work with Schwabe. To appear in the Proceedings of AfricaCrypt 2009.

Acknowledgements

I would like to thank my supervisor Mike Scott for giving me the opportunity to join his team. Mike, giving the best advice ever to a PhD student, told me to "follow your nose" not realising it would land me in Sweden with a new baby.

Mum, Dad, Paul, Carolyn, Joe, David, & Ann-Katrine Mattsson for all their support.

My colleagues at the School of Computing especially Noel, Augusto, Claire, Hego, & Wesam.

Gerry & Caroline for helping so much with the final push.

CryptoJedi himself; my co-author Peter Schwabe.

IRCSET who sponsored my research work.

The staff and students of Luleå Technical University Sweden, especially Matt Thurley, for their help during my time as *Guestdocktorand*.

Residents & friends of 12 Middle Mountjoy St. Dublin for good good times.

Most of all to Vicki & Klara-Jo for putting up with me.

Abstract

Elliptic Curve Cryptography (ECC) has been adopted by the US National Security Agency (NSA) in Suite "B" as part of its "Cryptographic Modernisation Program ". Additionally, it has been favoured by an entire host of mobile devices due to its superior performance characteristics. ECC is also the building block on which the exciting field of pairing/identity based cryptography is based. This widespread use means that there is potentially a lot to be gained by researching efficient implementations on modern processors such as IBM's Cell Broadband Engine and Philip's next generation smart card cores. ECC operations can be thought of as a pyramid of building blocks, from instructions on a core, modular operations on a finite field, point addition & doubling, elliptic curve scalar multiplication to application level protocols.

In this thesis we examine an implementation of these components for ECC focusing on a range of optimising techniques for the Cell's SPU and the MIPS smart card. We show significant performance improvements that can be achieved through of adoption of ECC.

Contents

List of Algorithms

In the landscape of extinction,
precision is next to godliness.

Samuel Beckett

1 Preface

In this thesis we review a selection of the numerous public key cryptographic algorithms and demonstrate how they can be optimised by capitalising on improvements in modern processor design.

In Chapter 2, we commence by performing a review of public key cryptography. We continue by reviewing the mathematics behind the best-known techniques. In Chapter 3 we discuss design considerations when implementing cryptographic pairings inside a modern smart card. In Chapter 4 we outline work undertaken to identify pairing friendly curves using super-computing resources. In Chapter 5 we review the design of the Cell Broadband Engine. In Chapter 6 we outline performance gains for SSL achieved by utilising a specialist multi-core processor. Finally in Chapters 7 and 8 we describe techniques used to achieve speed records when using Elliptic Curve Cryptography on a synergistic processor.

2

Crypto 101

This chapter outlines the background to the mathematics of cryptography, details a number of the standard algorithms, multi-precision arithmetic, and evolves to a description of Elliptic Curve Cryptography.

2.1 Cryptographic concepts

As a background to cryptography and an introduction to security protocols, we would like to provide an historic example of an algorithm called the Caesar Cipher attributed to Julius Caesar the 1st Roman Emperor from 61BC to 44BC. The algorithm is a very simple but effective example of the principles involved in the encryption of messages.

In the Caesar cipher the algorithm is a simple symbol swap. All the letters of the alphabet A through W are substituted with the character three places after it in sequence, with X, Y and Z been represented by A, B, C. Hence A is represented by D, N by Q etc. For example the message "ATTACK GAUL " can be encoded to "DWWDFN JDXO". A simple backward step of subtracting 3 places lets one arrive at the original message. This is a crude cipher on which the success of keeping the content secret depends on the casual observer having no knowledge of the algorithm (the simple alphabet switch) and the offset (3). Simple improvements include having a jumbled up alphabet (A=B, B=Z, C=N etc.) as a look-up table with both sides knowing the new lookup table and possibly having an

increment on the offset in some formula also agreed by the participants. The fundamentals of modern cryptography build on these simple ideas of message, algorithm and key.

Messages can be transformed to numbers via simple ASCII (American Standard Code for Information Interchange) representation of the characters making up the message. This is where each character/letter of the message is represented by a well known number (a=97, b=98, z= 122 etc.) and in computers these numbers are represent by binary (1 or 0) bits.

Operations which transform these ASCII numbers are equivalent to transforming the original message they represent. There are other methods of cryptography but the mathematical methods described below are believed to be the strongest. Modern cryptography relies upon the mathematics of making transformation operations, which are hard to invert, even when one knows the transformation used. For example the well known RSA method which is based on the difficulty of integer factorisation, or the El Gamal method which is based on the difficulty of the discrete logarithm problem.

2.2 Some building blocks used in Modern Cryptography.

2.2.1 Modular arithmetic

Modular arithmetic deals with a set of integers where if N is positive then the numbers modulo N are the set of numbers $\forall\ i \mid 0 \leq i < N$. If two numbers have the same remainder when divided by the modulo N then we say they are congruent modulo N .

An everyday example of modular arithmetic is the set of hours on a clock

$0, 1, 2, 3, 4, 6, 7, 8, 9, 10, 11 \mod 12$. So if it is two o'clock and we add three hours it is five o'clock as it will also be in fifteen plus two hours

$(15 + 2 \mod 12 = 5)$.

Two of the most popular public key algorithms use modular exponentiation as their underlying mathematical process.

2.2.2 Prime numbers

Webster's New Collegiate Dictionary defines a prime as follows:

```
Prime \'prim\ n [ME, fr.  MF, fem.  of prin first, L primus;
akin to L prior] 1 :  first in time :  ORIGINAL 2 a :
having no factor except itself and one <3 is a ~ number> b :
having no common factor except one <12 and 25 are relatively ~> 3 a :
first in rank, authority or significance :  PRINCIPAL b :
having the highest quality or value <~ television time>
```

Simply put, a prime is a number that has exactly two positive integer factors, 1 and itself.

Eratosthenes (275-194 B.C., Greece) devised a 'sieve' to discover prime numbers. Using this method to find all the prime numbers first write down all the positive whole numbers.

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	*etc.*

Then you take away all the number that are multiples of 2 , then all the numbers that are multiples of 3, then 4, 5, 6, 7 and so on (numbers known as composite numbers).

The numbers that are left are prime numbers.

	2	3		5		7			
11		13				17		19	
		23						29	
31							37		*etc.*

Prime numbers have fascinated mathematicians for centuries. The problem is no one has yet determined a method to predict the sequence of prime numbers. The only known method is brute force (exhaustive search) i.e going through all possibilities. Picking a number large enough brings us to primes that are usable in cryptography.

2.2.3 Chinese Remainder Theorem

The ancient (4th century AD) Chinese mathematician Sun-Tsu solved the problem of identifying those integers x that leave remainders 2, 3 and 2 when divided by 3, 5, and 7. One

solution is $x = 23$, and all solutions are of the form $23 + 105k$ for arbitrary integers k [26]. The *Chinese Remainder Theorem* (CRT) expressed in terms of integers modulo m.

$$\begin{aligned} x &\equiv r_1 (\text{mod } m_1) \\ x &\equiv r_2 (\text{mod } m_2) \\ x &\equiv r_3 (\text{mod } m_3) \\ &... \\ x &\equiv r_n (\text{mod } m_n) \end{aligned}$$

then there is a unique solution X, for x lying between 0 and $m_1\ m_2\ m_3\ ...\ m_n$, and the general solution is congruent to X (mod $m_1\ m_2\ m_3\ ...\ m_n$).

Let

$$P = \prod_{i=1}^{n} m_i$$

and, for all $i \in \mathbb{N}$ $(1 \leq i \leq n)$, let y_i be an integer that satisfies

$$y_i \cdot \frac{P}{m_i} \equiv 1 \pmod{m_i}$$

Then one solution of these congruences is

$$x_0 = \sum_{i=1}^{n} a_i y_i \cdot \frac{P}{m_i}$$

Any $x \in \mathbb{Z}$ satisfies the set of congruences if and only if it satisfies

$$x \equiv x_0 \pmod{P}$$

Used by the ancient Chinese to count large numbers of troops, one use of the CRT is to do arithmetic on large numbers by choosing a set of moduli $m_1\ m_2\ m_3\ ...\ m_n$ and then treating each number as a set of remainders $r_1\ r_2\ r_3\ ...\ r_n$ rather than a sequence of digits. Subsequently one does the arithmetic and recovers the solutions by using the CRT [97].

2.2.4 Groups

A group is a set of numbers with an operator. This is a relatively simple group to illustrate the properties useful for cryptography. The numbers 1 to 4 and an operator being multiply $\times$.

$$[\times, 1, 2, 3, 4 \mod 5]$$

The group is closed when the answer is always within the group

$$(3 \times 2) = 6 \mod 5 = 1$$

The group has an identity which is a number which when applied by the operator does not change the element.

$$2 \times 1 = 2$$

It is associative

$$(2 \times 3) \times 4 = 2 \times (3 \times 4) \mod 5$$

Every element has an inverse

$$(3 \times 2) = 6 = 1 \mod 5$$

A group is cyclic if it has a member that when subjected to repeated applications of the operator will give every member of the set

$$\begin{aligned} 2 &= 2 \mod 5 = 2 \\ 2 \times 2 &= 4 \mod 5 = 4 \\ 2 \times 2 \times 2 &= 8 \mod 5 = 3 \\ 2 \times 2 \times 2 \times 2 &= 16 \mod 5 = 1 \end{aligned}$$

In order to identify how many times to apply the operator to get 3, one must simply keep applying it -a method known as exhaustive search.

$$2 \times 2 \times 2 = 3 \mod 5$$

This is relatively simple since this group has 4 members. The groups that we use have approximately 2^{512} elements. So the approach outlined would be too time consuming. Therefore the determination of x given 2^x in the group is very difficult.

The point here is that applying the operation N times is easy. Finding out how many times the operation was applied is computationally too expensive. This is a one way function.

$$y = g^x \bmod p$$

The formula can be used to generate elements of the group over the field $\mathbb{F}_q$. In this context g can be described as the generator of the group. The number of elements in the group will be a divisor of $p-1$. In our example 2 is a generator of the group of order 4 over $\mathbb{F}_5$.

2.3 Types of cryptography

There are two main types of mathematical cryptography

- Symmetric or *secret* key
- Asymmetric or *public* key

2.3.1 Symmetric Key

Symmetric key is a relatively easy concept to understand. Essentially one party (the encryptor) uses a secret key to a mathematical function which encrypts the plaintext message to a secure form. The message is passed to the decrypter who uses *the same key* to apply

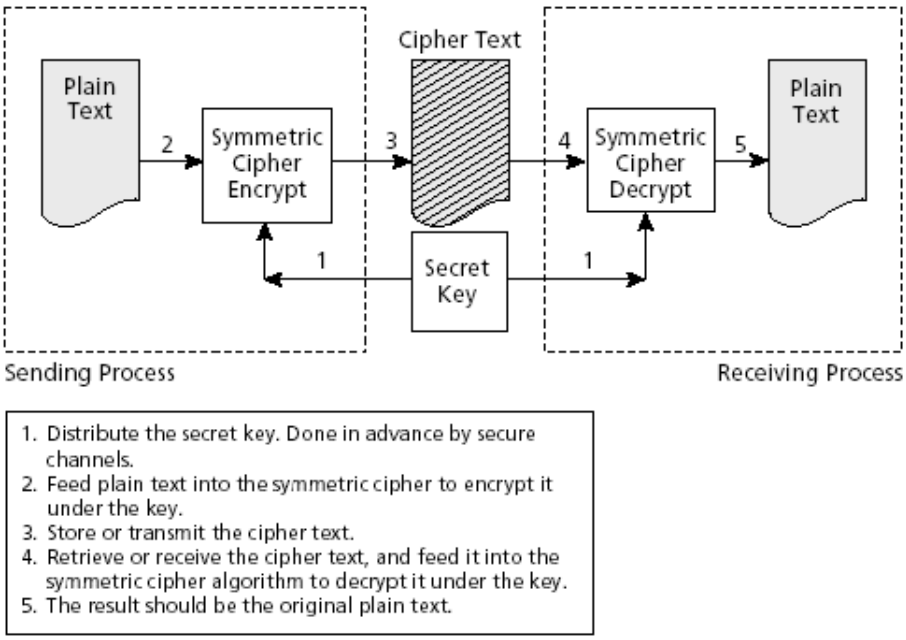

Figure 2.1: Symmetric Key *(source [33])*

an inverse mathematical function which decrypts the message returning it to its original plaintext. The principal issue in symmetric cryptography is the secure transport of the key between the parties. Examples include DES, IDEA, & AES.

2.3.2 Asymmetric Key

Background

Asymmetric key cryptography is a more complex concept to grasp but is generally more useful for application level security. The key is broken up into two parts known as the *public* key and the *private* key. The public key is used to encrypt the message and the private key to decrypt. Asymmetric is popularly known as *public key cryptography.*

The original discovery of methods suitable for public key cryptography have traditionally been attributed to Diffie, Hellman & Merke in 1977, more commonly known as the Diffie-Hellman [32] method. However in 1997 history has corrected itself with the declassification of papers from the British secret services which lay claim to the fact the James Ellis probably

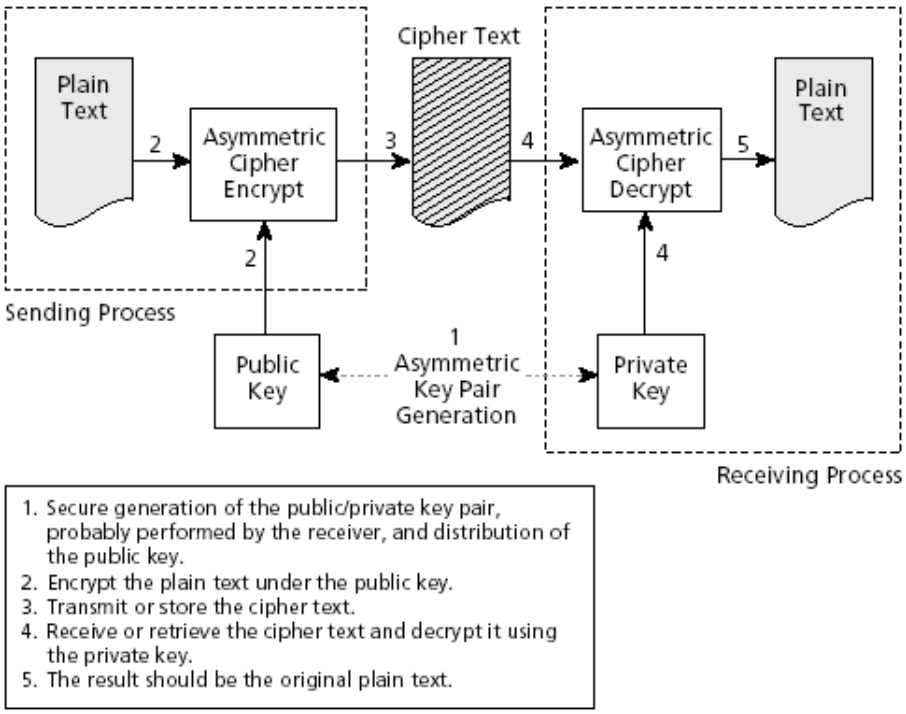

Figure 2.2: Asymmetric Key *(source [33])*

invented and that Clifford Cocks probably discovered a method for public key cryptography long before Diffie et al. But, suitable to persons working inside the intelligence services, they kept the claim to themselves. GCHQ (now CESG [18]) have since backed up their claims with documentary evidence [35]. The American equivalent (NSA) have claimed an even earlier idea but evidence is scant. Background to this interesting story is in Wired magazine [62] where Diffie (a colorful character in his own right) visits a retired Ellis and brings him to a pub in England attempting to prise the real story out of him. Ellis is too humble (and clever !) and leaves Diffie none the wiser but doubtful of his own place in history.

Technical

Asymmetric key protocols are commonly based on the one-way function.

$$y = g^x \mod p \text{ where } p \text{ is a prime.}$$

Assume g and p are public. Then given x finding y is easy. However given y to find x is assumed to be very hard. This is based upon the fact that certain problems are intractable. The issues with the above equation (also written as $x = \log_g(y)$) is known as the discrete logarithm problem (DLP). Based on this we can build asymmetric key encryption. The function looks simple and it would seem that with a simple try every x or brute force attack would yield y. However, for acceptable levels of protection to make such an attack unfeasible it is required to use a p with at least 1024 bits and the exponent x with at least 160 bits.

The size of p is referred to as the *field size*, and x as the *group order size*. The field size is so much larger than the order size as index-calculus methods exist for solving the discrete logarithm problem, which require a relatively large field size to resist. There are faster methods than brute force search (for example so called "square root methods" like the Pollard rho and Pollard Lambda algorithms), but they are still computationally infeasible for numbers of cryptographic size.

As an alternative to the one way function a "hard problem" exists on an Elliptic Curve.

$$y = x^3 + Ax + B \mod p$$

Take a point on the curve $P(x, y)$. Then assume P is used as a public group generator.

$$Y = xP$$

This also represents a strong one-way function. If we know x then calculating Y is easy. However given Y finding x is difficult. This is basically the same Discrete Logarithm problem. The advantage of using Elliptic Curves is that index calculus methods used to attack the modular equation are not known and hence both the field and order sizes can be as low as 160 bits for practical security. This lower bit size reduces computation overhead and allows for efficient use in restricted devices such as mobile phones or smart cards.

The advantage of all this asymmetric cryptography lies with the property that possessing the public key provides little clue to the private key allowing the public key to be freely published to allow anyone to encrypt to the holder of the private key. This unique

property overcomes the limitations of symmetric cryptography which requires prior "key swapping" before use. However this advantage comes with a performance penalty. Asymmetric algorithms are significantly more computationally expensive than symmetric and are not suitable to encrypt large amounts of data taking too long a time for real-time data communications such as a Virtual Private Network (VPN) or an email system.

The solution is to use another more efficient algorithm such as a symmetric algorithm to perform the bulk data encryption and use the public key methods to encrypt the key(s) and transport them with (usually by simply attaching them to) the encrypted data. This hybrid solution is used by most common protocols such as SSL, SSH, IpSec and S/Mime.

For practical purposes a combination of the two methods provides both practical performance together with ease of use.

2.4 Commonly used asymmetric algorithms

2.4.1 RSA

Background

Discovered in 1977 and named after its inventors, Ron **R**ivest [78], Adi **S**hamir and Leonard **A**dleman, RSA [76] encryption is based on the difficulty of the integer factorisation problem, and transforms the message M into the number C

$$C = M^e \mod N$$

The numbers e and N are the two public numbers created and published. They are your public key. As before the message M can be simply the digital value of a block of ASCII characters.

The formula states: multiply the Message M by itself e times, then divide the result by the number N and save only the remainder. The remainder that we have called C is the encrypted representation of the message.

Example Application

Alice publishes the public key numbers $e = 29$ and $N = 77$. Bob wants to send Alice the message "I have it". In decimal ASCII the message is 73321049711810132105116. Break this number string into smaller blocks less then N as per the following 73 32 10 49 71 18 10 13 21 05 11 6

To encrypt these blocks, apply the formula

$$C = M^e (\text{mod } N)$$

to each block.

Technical Overview

Here we present a description due to Mao [63]

As before we deal with Bob attempting to send a message to Alice.

Key Set-up.

Alice creates her public and private key pair thus

1. Choose two large random prime numbers p and q such that $\mid p \mid < \mid q \mid$
2. Compute $N = pq$
3. Compute $\phi(N) = (p-1)(q-1)$
4. Choose a random integer $e < \phi(N)$ such that $\gcd(e, \phi(N)) = 1$ and compute the integer d such that $ed \equiv 1 (\text{mod } (\phi(N)))$
5. Publicise (N, e) as her public key, discarding p, q and $\phi(N)$ and keeping d as her private key. e can be small but d must be impossible to guess.

To encrypt to Alice

To send a message $M < N$ to Alice, the sender Bob creates a ciphertext C by

$$C \leftarrow M^e (\text{mod } N)$$

For Alice to decrypt

To read the ciphertext C from Bob, Alice computes

$$M \leftarrow C^d (\text{mod } N)$$

Efficient decryption

The Chinese remainder theorem 2.2.3 can be used to yield efficient algorithms for RSA decryption and can be more efficient (in terms of bit operations) than working modulo N.

2.4.2 ECC

Background

Another form of "hard problem" that mathematicians have found useful to the field of cryptography is that of Elliptic Curves.

The discovery of the use of Elliptic Curves for public key cryptography can be attributed independently to Neil Koblitz [58] and Victor Miller [68] who both made discoveries in 1985.

Technical Overview

Note: Curves can be defined in Affine (2 dimensions) or Projective (3 dimensions co-ordinates) - The equations we present are in Affine co-ordinates.

To follow general cryptographic convention we use curves of the form

$$y^2 = x^3 + a \times x + b \bmod p$$

This means we are only allowed to use the integers from zero to $p - 1$ as input.

For example let us take an equation with $p = 11$, $a = 4$ and $b = 7$.

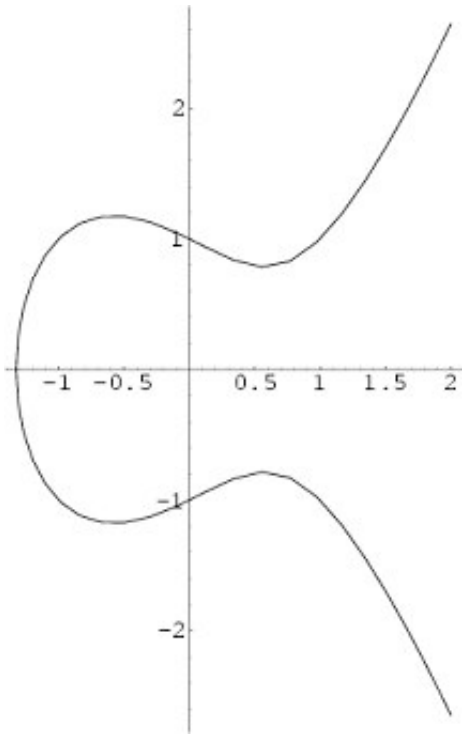

Figure 2.3: Elliptic Curve

$$y^2 = x^3 + 4x + 7 \bmod 11$$

The group of points of interest are those with (x, y) coordinates which satisfy this equation, plus the point at infinity (denoted by O).

To add two points on a curve, it is not possible to simply add the coordinates to find a point which still satisfies the curve equation.

However there are a set of rules which one can apply for curves of this type.

The rules are (see Smart [91])

- Rule 1: $O + O = O$
- Rule 2: $(x_1, y_1) + O = (x_1, y_1)$
- Rule 3: $(x_1, y_1) + (x_1, -y_1) = O$

- Rule 4: if $x_1 \neq x_2, (x_1, y_1) + (x_2, y_2) = (x_3, y_3)$ where

$$\begin{aligned} x_3 &= (\beta^2 - x_1 - x_2) \bmod p \\ y_3 &= (\beta(x_1 - x_3) - y_1) \bmod p \\ \beta &= ((y_2 - y_1)/(x_2 - x_1)) \bmod p. \end{aligned}$$

- Rule 5: if $y_1 \neq 0, (x_1, y_1) + (x_1, y_1) = 2(x_1, y_1) = (x_3, y_3)$ where

$$\begin{aligned} x_3 &= (\beta_2 - 2x_1) \bmod p \\ y_3 &= (\beta(x_1 - x_3) - y_1) \bmod p \\ \beta &= ((3x_1^2 + a)/(2y_1)) \bmod \end{aligned}$$

We recommend Nigel Smart's "Introduction to Cryptography" [91], Wenbo Mao's "Modern Cryptography" [63] Menezes "Elliptic Curve Cryptography" [66].

Applying these rules, two points on the curve may be added to yield a third point also on the curve. For example in the figure above point $P + Q = (P + Q)$ on the curve.

Note 1: : Should we detect, in our calculations, a divide by zero we can stop and say the result is the point at infinity (O).

Note 2: The point at infinity (O) acts like zero in regular addition : Add a point to the point at infinity we get the original point. This is the additive identity for the group.

Note 3. Prime Modulus. The group is cyclic, it has a generator function such that when this function is applied to any member of the group, it will only result in another member of the group. If applied to all members of the group then it will produce another set containing all members of the original group. Interestingly the output sequence of the results of the application of the generator is random. See [96].

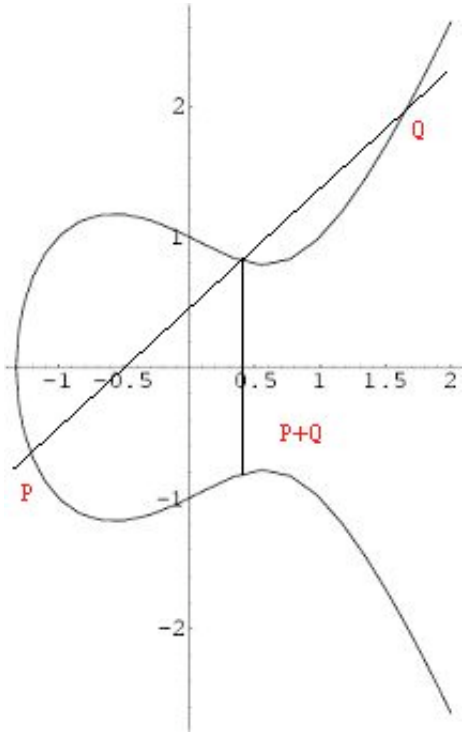

Figure 2.4: Point Addition . *(source [64])*

Note 4: Multiplication

Take a point P an (x, y) point and multiply it by an integer d. dP is $d \times P$ and we can break down $d \times P$ to $(P + P + ...P)$ i.e. P added to itself d times. Then apply the rules of addition. In the illustrated figure $3P = P + P + P$.

This integer d is called a *scalar* as opposed to the coordinate (point) P.

What does all this mean to cryptography ?

We take an elliptic curve (i.e. modulus p and parameters a and b) and a point on this curve P.

Then, take a scalar d and find dP to get another point on the curve Q.

We keep d secret. We can use the curve (p, a, b) and points P, Q as the public key. The challenge for any attacker is to find d. No-one has found a sub-exponential algorithm to be able to compute d. This is another manifestation of the discrete logarithm problem.

If the modulus p is large enough (200 bits or so) then it would take today's supercom-

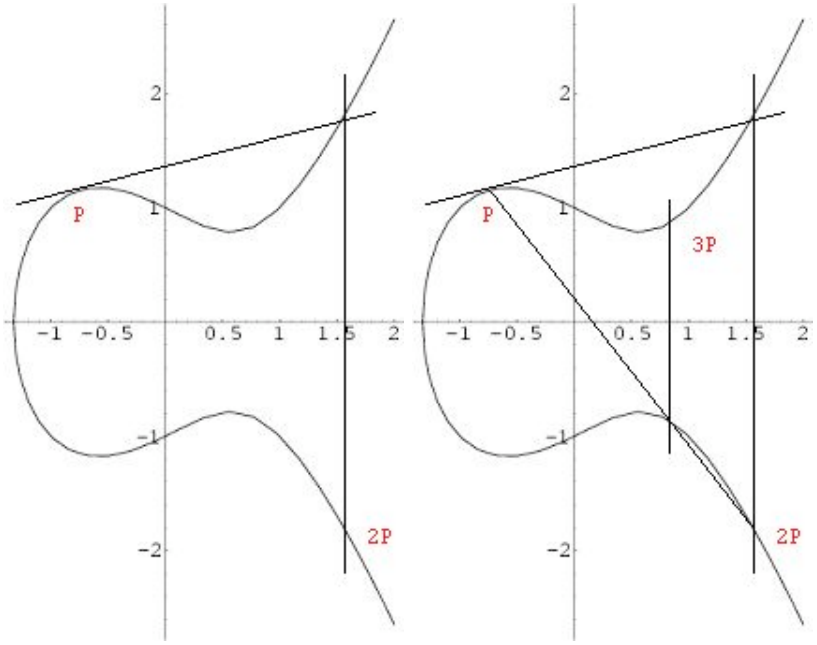

Figure 2.5: $2 \times P$, $3 \times P$*source [64])*

puters several thousands of years. This is the basis for the application to cryptography.

Scalar multiplication on an elliptic curve is relatively easy, but the inverse, which is extremely hard.

To illustrate we will consider Key agreement using Elliptic Curve Diffie-Hellman.

Diffie-Hellman is a technique to allow unauthenticated key agreement using exponentiation. The security rests on the intractability of the Computational Diffie-Hellman problem and the Discrete Logarithm Problem.

- Alice calculates her curve and makes p, a, b and a point P public.
- She generates some random d_a and keeps this secret.
- Alice sends Bob Q_a which is equal to $d_a P$.
- Bob gets Alice's public components and generates his own random d_b.
- He calculates Q_b by computing $d_b P$.
- Bob then computes a secret value $S = d_b Q_a$.
- Since Q_a is just $d_a P$ what Bob has computed is $S = d_b d_a P$.

- He sends Alice Q_b and Alice uses this to compute her secret value $S = d_a Q_b$.
- Since Q_b is $d_b P$, what Alice has done is to compute $S = d_a d_b P$ this is the same as Bob computed.

So Alice and Bob can "secretly" get to the same point on the curve S, by using this protocol. A simple method to extract a key is just to ignore the y coordinate and take the x coordinate as a number. This derived number can be used as a key. Bob can use this secret value to make an AES encryption key. Alice can use the method outlined above to get the same encryption key. So what Bob encrypts, Alice can decrypt. The Attacker Eve intercepting this exchange just knows p, a, b, P, Q_a, and Q_b. The only way for Eve to determine S is to get either d_a or d_b which Alice and Bob are holding secret.

To get d calculate one of the d's by using the fact that she knows either

$Q_a = d_a P$ and she knows Q_a and P

OR

$Q_b = d_b P$ and she knows Q_b and P.

This is exactly the discrete logarithm problem of ECC as outlined above.

ECC key size and RSA key size

The main stumbling block to the wide spread usage of public key cryptography is the computational overhead of traditional PKC based on RSA. ECC supports equivalent security levels with less computational overhead. ECC has smaller key sizes and signatures. ECC is often used in mobile, embedded, and sensor networks for its power characteristics. For more material on ECC the reader is referred to [45] and [13]. The US standards body (NIST) has issued guidelines for equivalent key length usage

2.4.3 Pairings

Joux [55] and Sakai, Ohgishi & Kasahara [82] independently proposed using properties of pairing-mapping functions applied to cryptography to establish ID-based PKI. Initially the area of pairings had been discounted as holding no promise for cryptographic applications.

ECC key size	RSA key size	Key size	AES key size
bits	bits	ratio	bits
163	1024	1:6	
256	3072	1:12	128
384	7680	1:20	192
512	15360	1:30	256

Table 2.1: NIST guidelines for public key sizes for AES

It wasn't until the Joux publication [55] that countered this claim that this whole area was opened to researchers culminating with Boneh & Franklin's much publicised paper in 2001.

Let $\mathbb{G}_1$ and $\mathbb{G}_2$ denote two groups of prime order q, where $\mathbb{G}_1$, with an additive notation, denotes the group of points on an elliptic curve; and $\mathbb{G}_2$, with a multiplicative notation, denotes a subgroup of the multiplicative group of a finite field.

Multiplicative groups will be represented here as $\mathbb{Z}_n^*$, which is the set of positive integers less than n and relatively prime to n under multiplication modulo n. An integer is relatively prime to another if their only common positive divisor is 1. For example, 8 and 15 , though not prime numbers, are relatively prime.

A pairing is a computable bilinear map between these two groups. Two pairings have been studied for cryptographic use. They are the **Weil**[1] **pairing** and the **Tate pairing** . For the purposes of discussion, we let $\hat{e}$ denote a general bilinear map, i.e. $\hat{e}$:$\mathbb{G}_1 \times \mathbb{G}_1 \rightarrow \mathbb{G}_2$, which can be either a modified Weil pairing or a Tate pairing.

In this notation the Diffie-Hellman (DH) solution described above is a tuple in $\mathbb{G}_1$ as $(P, xP, yP, zP) \rightarrow \mathbb{G}_1$ for some x, y, z (chosen at random) $\rightarrow \mathbb{Z}_q*$

satisfying $z = xy \mod q$.

Properties of Pairings

Bilinear: If $P, P_1, P_2, Q, Q_1, Q_2 \in \mathbb{G}_1$ and $a \in \mathbb{Z}_q^*$, then $\hat{e}(P_1 + P_2, Q) = \hat{e}(P_1, Q).\hat{e}(P_2, Q)$, and $\hat{e}(P, Q_1 + Q_2) = \hat{e}(P, Q_1).\hat{e}(P, Q_2)$.

Non-degenerate: There exists a $P \in G_1$ such that $\hat{e}(P, P) \neq 1$.

[1]Pronounced "Vay". Andre Weil in the 1940's.

Computable: If $P, Q \in G_1$, one can compute $\hat{e}(P, Q)$ in polynomial time.

Since pairings were discovered, new protocols for identity-based encryption [14], [81], short signatures [15] and identity-based signcryption [65]. We do not attempt to provide a complete history here, but instead refer the interested reader to the Pairing-based Crypto Lounge [7].

2.4.4 Pairing-friendly elliptic curves

When it comes to the selection of elliptic curves suitable for pairing-based cryptography, one is currently limited to either the supersingular curves or certain special non-supersingular curves of prime characteristic. A basic requirement is that the selected elliptic curve should have a small *embedding degree*, or *security multiplier*, denoted as k. In this chapter, it will be assumed that k is even.

If the curve was defined over a finite field of size q, G is mapped to a subgroup of a finite field of size q^k for some integer k. The smallest such integer k is called the embedding degree.

Therefore, for cryptographic purposes a pairing-friendly elliptic curve over a finite field consists of the finite set of points (including a point at infinity) on a curve which can be described by one of

$$
\begin{aligned}
E(\mathbb{F}_{p^m}): &\qquad y^2 = x^3 + Ax + B \\
E(\mathbb{F}_{2^m}): &\qquad y^2 + y = x^3 + x + b \\
E(\mathbb{F}_{3^m}): &\qquad y^2 = x^3 - x + b
\end{aligned}
$$

In the first case, the curve can be either supersingular, with an embedding degree of $k = 2$, or nonsupersingular with $m = 1$ and any finite embedding degree [13]. In the second case, the curve is supersingular and has a maximum embedding degree of $k = 4$, where $b = 0, 1$. In the third case, the curve is also supersingular with a maximum embedding degree of $k = 6$, and where $b = \pm 1$.

As is common in elliptic curve cryptography over $E(\mathbb{F}_q)$, one wants to work with a group of points of prime order r, where $r \mid q+1-t$ the total number of points on the curve (denoted $\#E$), and where t is the trace of the Frobenius, with $|t| \leqslant 2\sqrt{q}$ (the Hasse condition) [66]. These points then form a prime order cyclic abelian group. This group size needs to be large enough to avoid various generic attacks on the elliptic curve discrete logarithm problem and therefore, at a minimum, r should be 160-bits. The embedding degree k is related to this group of points on the elliptic curve by the condition that k is the smallest positive integer such that $r \mid (q^k - 1)$. A further security requirement for these elliptic curves is that $\mathbb{F}_{q^k}$, where $q = p, 2^m$ or 3^m, should be an extension field of sufficient size to prevent an index calculus attack on the discrete logarithm problem in that field. So, at a minimum, $k.\lg(q)$ should be 1024 bits.

We therefore have the interesting constraints that r can, at most, be approximately as big as q (due to the Hasse condition), with $\lg(r)$ a minimum of 160, and that $k.\lg(q)$ should then be at least 1024. One obvious feasible solution would be to choose $\lg(r) \approx 170$, $r = q+1-t$, and $k = 6$ so that $6.\lg(q) \approx 1024$. This explains the early popularity of curves of characteristic 3 with $k = 6$. This also has the advantage of keeping the size of the elliptic curve as small as those required for standard ECC while still attaining the minimum levels of index calculus security. However, another valid and popular choice would be to use a supersingular [14] or non-supersingular curve [85] over $\mathbb{F}_p$, with $\lg(r) = 160$, $\lg(p) = 512$ and $k = 2$.

In the case of fields of low characteristic, the security situation is rather unclear. As first pointed out by Coppersmith [25], the discrete logarithm problem in $\mathbb{F}_{2^m}$ is somewhat easier than it is over a prime characteristic field. According to the current record holder [95], who was able to calculate discrete logarithms for $m = 607$, it would require $m \approx 1200$ to obtain a greater level of security than 1024-bit RSA. Interpolating into the tables provided by Lenstra [60] would suggest that 1300 bits would be sufficient. Page, Smart and Vercauten [75] have since observed that the record for prime field discrete logarithms is 398 bits [61], $607/398 = 1.53$.

A pairing is denoted as $e(P, Q)$, where P is taken as a point of order r, usually on $E(\mathbb{F}_q)$,

and Q is a point on $E(\mathbb{F}_{q^k})$ linearly independent of P. The pairing evaluates naturally as an element of order r in $\mathbb{F}_{q^k}$. Its most important cryptographic property is its *bilinearity*

$$e(aP, bQ) = e(P, Q)^{ab}$$

If Q should be linearly dependent on P and $P \in E(\mathbb{F}_q)$, then the pairing is degenerate and $e(P, Q) = 1$, and thus, for example, $e(P, P) = 1$. On a supersingular curve, it is usual to exploit the existence of a distortion map $\psi(.)$ that maps a point from $E(\mathbb{F}_q)$ to a linearly independent point on $E(\mathbb{F}_{q^k})$. Now both P and Q can be linearly dependent points from the same group of order r on $E(\mathbb{F}_q)$ and the distorted pairing can be calculated as $\hat{e}(P, Q) = e(P, \psi(Q))$. This pairing has the additional, and sometimes useful, property that $\hat{e}(P, Q) = \hat{e}(Q, P)$, which is implied by the condition that $\hat{e}(P, P) \neq 1$.

2.5 Identity Based Encryption (IBE)

In 1984, Shamir [88] (the 'S' in RSA) proposed the first identity-based signature scheme and outlined a solution to this key distribution / certificate management problem calling it Identity Based Encryption (IBE). However, he didn't have an implementation.

The idea was that if any string can be a public key, then one could use an identifier of the recipient to be the public key in which case one doesn't need to locate the public key associated with an identity. The identity is the key. Furthermore, one can use the "identifier string" combined with something like a date/time and encrypt messages to be read *into the future*.

Our friends Alice and Bob are to communicate in this system. Bob can simply use Alice's email address (alice@wonderland.com) and the public parameters of a trusted third party as the public key to encrypt the message. When Alice receives the message, she contacts the trusted third party (KDC key distribution center), validates herself and receives the private key associated with her identity.

Shamir's proposed IBE remained an elusive "holy grail" for cryptographers until Cocks [22] working at the British secret service GCHQ discovered a method relying on quadratic

residues. Unfortunately this method is impractical for widespread use because of the bandwidth overheads. It has recently come to light that two Japanese researchers Sakai and Kasahara [81] also made a significant discovery relevant to IBE using pairings but due to language barriers their implementation wasn't widely known.

In 2001, Boneh and Franklin [14] announced a more viable method using the Weil pairing (See. 2.4.3) This method, while demonstrable, still lacked a pragmatic implementation which could be considered widely usable. Optimizations of the pairing mathematics were proposed by Barreto-Kim-Lynn-Scott [2] in 2002 which moved the processing overheads close to that of the widely used RSA algorithm. These optimizations involved

- Point tripling for super-singular elliptic curves over $\mathbb{F}_{3^m}$.
- Removal of irrelevant operations from conventional algorithms.

Background

An IBE system involves, using the language of Boneh & Franklin's seminal paper [2] a set of four algorithms.

Setup: A key generator (KDC) which runs a 'setup' algorithm to generate global system parameters and a *master-key* which the KDC keep safe. The whole security of the system relies on the safekeeping of this master-key in a device sich as a hardware security module.

Extract: The KDC runs an extract algorithm inputting the user's identity (or any bit string) and using the master-key from the setup. The output is the users *private-key* associated with the users identity. Its important that the private-key is transported to the user in a safe manner and that the KDC has made a full examination of the user credentials before issuing a key corresponding to those credentials.

Encrypt: A probabilistic algorithm. Any user encrypts using the global system parameters and public key *ID*. The output is the ciphertext.

[2] Extended abstract in [14]

Decrypt: This process takes the ciphertext from the encrypt function, global system parameters and the private key issued by the KDC. The output is the corresponding plaintext.

Boneh & Franklin's paper provides a random oracle security proof for their IBE method. Hence, it is secure against an adaptive chosen ciphertext attack assuming the hardness of the so-called Bilinear Diffie Hellman problem.

Technical

The following description is from Mao [63].

Set-up

1. Generate two groups $\mathbb{G}_1, \mathbb{G}_2$ of prime order q and a mapping-in-pair $e : {\mathbb{G}_1}^2 \to \mathbb{G}_2$. choose P and element in $\mathbb{G}_1$.

2. Pick $s \in_{\cup} \mathbb{Z}_q$ and set $P_{pub} \leftarrow [s]P$; s is the *master key.*

3. Using a strong hash algorithm $F : \{0,1\}^* \to \mathbb{G}_1$. to map the identity string ID to an element in $\mathbb{G}_1$.

4. Specify another suitable hash algorithm $H : \mathbb{G}_2 \to \{0,1\}^n$.

The KDC keeps s as the system master-key and publishes the parameters.

$$(\mathbb{G}_1, \mathbb{G}_2, e, n, P, P_{pub}, F, H)$$

Private Key Generation (= Extract)

Let ID denote an authenticated and validated user's identity (it can be any string)

1. Compute $Q_{ID} \leftarrow F(ID)$. This is an element in $\mathbb{G}_1$ and is the users public key.

2. Set the users private key d_{ID} as $[s]Q_{ID}$.

Encryption

To send an encrypted message, obtain the system parameters $(\mathbb{G}_1, \mathbb{G}_2, e, n, P, P_{pub}, F, H)$. Using them, compute $Q_{ID} = F(ID)$. To encrypt $M \in \{0,1\}^n$ pick $r \in_U \mathbb{Z}_q$ and compute

$g_{ID} \leftarrow e(Q_{ID}, [r]P_{pub}) \in \mathbb{G}_2$,

The ciphertext is $C \leftarrow ([r]P, M \bigoplus H(g_{ID}))$.

Decryption

To decrypt C using ID's private key d_{ID} as $[s]Q_{ID}$, compute $M = V \bigoplus H(e(d_{ID}, U))$.

Other "hard problems"

As mentioned above the **Diffie-Hellman (DH)** tuple in $\mathbb{G}_1$ is a tuple $(P, xP, yP, zP) \in \mathbb{G}_1^4$ for some x, y, z chosen at random from $\mathbb{Z}_q$ satisfying $z = xy \ mod \ q$.

Computational Diffie-Hellman (CDH) problem: Given the first three elements in a DH tuple, compute the remaining element. The CDH assumption: no algorithm exists running in expected polynomial time which can solve the CDH problem with non-negligible probability.

Decision Diffie-Hellman (DDH) problem: Given a tuple $(P, xP, yP, zP) \in \mathbb{G}_1^4$ for some x, y, z chosen at random from Zq, decide if it is a valid DH tuple. If a pairing can be calculated then this can be solved in polynomial time by verifying the equation $\hat{e}(xP, yP) = \hat{e}(P, zP)$.. Note that this is contrast to the situation in the simple finite field where the DDH problem is also complex. This is also difficult on an elliptic curve if a pairing cannot be calculated; that is if the elliptic curve is not pairing-friendly.

Bilinear Diffie-Hellman (BDH) problem: Let P be a generator of G_1. The BDH problem in $\mathbb{G}_1, \mathbb{G}_2, \hat{e}$ is given $(P, xP, yP, zP) \in \mathbb{G}_1^4$ for some x, y, z chosen at random from Zq, compute $W = \hat{e}(P, P)^{xyz} \in \mathbb{G}_2$.

The following description of multi-precision algorithms is primarily derived from the Handbook of Elliptic and Hyperelliptic Cryptography ([23] Chapters 10 and 11), and the Handbook of Applied Cryptography ([67] Chapter 14). However, it is equally well presented in Crandall ([27] Chapter 9), Smart ([91] Chapter 11 Section 11.5), and Rodríguez-Henríquez et al ([79] Chapter 5).

2.6 Integer representation

The fundamental mathematical layer of most cryptographic systems is the integer ring $\mathbb{Z}$ ([23] Chapter 10). On top of the integer ring, it is possible to build finite fields then elliptic curves and other sets. Efficient elliptic curve implementations therefore rely on implementing efficient integer arithmetic. General purpose computers can only operate on relatively small integers. In order to facilitate the large integers required by cryptography, we need to build special representation (multi-precision) on top of the basic integer types. Then we need to use efficient algorithms for arithmetic using these multi-precision integers.

Let $b \geq 2$ be an integer called the *base* or the *radix*. Every integer $u > 0$ can be written in a unique way as the sum

$$u = u_{n-1}b^{n-1} + ... + u_1 b + u_0$$

provided $0 \leq u_i < b$ and $u_{n-1} \neq 0$. This is what is known as the b *representation* of u and is denoted by $(u_{n-1}...u_0)_b$. The u_i's are the digits of u.

This representation is generally present in high-level programming languages as a data structure wrapping an array of base-type integers. However, no standards exist for these data structures so small differences in subtle areas, such as ordering or padding, cause interoperability problems. See Chapter 6 Section 6.2 and Chapter 7 Section 7.1.2 for further discussion on multi-precision toolkits and internal representation.

2.6.1 Elliptic-Curve Diffie-Hellman key exchange (ECDH)

Let $\mathbb{F}$ be a finite field and $E/\mathbb{F}$ an elliptic curve defined over $\mathbb{F}$. Let $E(\mathbb{F})$ denote the group of $\mathbb{F}$-rational points on E. For any $P \in E(\mathbb{F})$ and $k \in \mathbb{Z}$ we will denote the k-th scalar multiple of P as $[k]P$.

The Diffie-Hellman key exchange protocol [32] can now be carried out in the group $\langle P \rangle \subseteq E(\mathbb{F})$ as follows: User A chooses a random $a \in \{2, \ldots, |\langle P \rangle| - 1\}$, computes $[a]P$ and sends this to user B. User B chooses a random $b \in \{2, \ldots, |\langle P \rangle| - 1\}$, computes $[b]P$ and sends this to user A. Now both users can compute $Q = [a]([b]P) = [b]([a]P) = [(a \cdot b)]P$.

The joint key for secret key cryptography is then extracted from Q; a common way to do this is to compute a hash value of the x-coordinate of Q.

2.6.2 Montgomery arithmetic

For elliptic curves defined by an equation of the form $By^2 = x^3 + Ax^2 + x$, Montgomery introduced in [72] a fast method to compute the x-coordinate of a point $R = P + Q$, given the x-coordinates of two points P and Q and the x-coordinate of their difference $P - Q$.

These formulas lead to an efficient algorithm to compute the x-coordinate of $Q = [k]P$ for any point P. This algorithm is often referred to as the Montgomery ladder. In this algorithm the x-coordinate x_P of a point P is represented as (X_P, Z_P), where $x_P = X_P/Z_P$; for the representation of the point at infinity see the discussion in Appendix B of [9]. See Algorithms 2.1 and 2.2 for a pseudocode description of the Montgomery ladder.

Algorithm 2.1 The Montgomery ladder for x-coordinate-based scalar multiplication on the elliptic curve $E : By^2 = x^3 + Ax^2 + x$

Input: A scalar $0 \leq k \in \mathbb{Z}$ and the x-coordinate x_P of some point P
Output: $(X_{[k]P}, Z_{[k]P})$ fulfilling $x_{[k]P} = X_{[k]P}/Z_{[k]P}$

```
t = ⌈log_2 k + 1⌉
X_1 = x_P; X_2 = 1; Z_2 = 0; X_3 = x_P; Z_3 = 1
for i ← t − 1 downto 0 do
    if bit i of k is 1 then
        (X3, Z3, X2, Z2) ← LADDERSTEP(X1, X3, Z3, X2, Z2)
    else
        (X2, Z2, X3, Z3) ← LADDERSTEP(X1, X2, Z2, X3, Z3)
    end if
end for
return (X_2, Z_2)
```

Each 'ladder step' as described in Algorithm 2.2 requires 5 multiplications, 4 squarings, 8 additions and one multiplication with the constant $a24 = (A + 2)/4$ in the underlying finite field.

Algorithm 2.2 One ladder step of the Montgomery ladder

const $a24 = (A + 2)/4$ (A from the curve equation)
function LADDERSTEP($X_{Q-P}, X_P, Z_P, X_Q, Z_Q$)
$t_1 \leftarrow X_P + Z_P$
$t_6 \leftarrow t_1^2$
$t_2 \leftarrow X_P - Z_P$
$t_7 \leftarrow t_2^2$
$t_5 \leftarrow t_6 - t_7$
$t_3 \leftarrow X_Q + Z_Q$
$t_4 \leftarrow X_Q - Z_Q$
$t_8 \leftarrow t_4 \cdot t_1$
$t_9 \leftarrow t_3 \cdot t_2$
$X_{P+Q} \leftarrow (t_8 + t_9)^2$
$Z_{P+Q} \leftarrow X_{Q-P} \cdot (t_8 - t_9)^2$
$X_{[2]P} \leftarrow t_6 \cdot t_7$
$Z_{[2]P} \leftarrow t_5 \cdot (t_7 + a24 \cdot t_5)$
return $(X_{[2]P}, Z_{[2]P}, X_{P+Q}, Z_{P+Q})$
end function

3

Implementing Cryptographic Pairings on Smartcards

Background

Smart cards have been used for many different purposes over the last two decades, from simple prepaid credit counter cards used in parking meters, to high security identity cards intended for national ID programs. Their wide spread use in banking cards and GSM/3G mobile phones has possibly made them the most common form of computing device on the planet. One of the overriding design criteria for a smart cards is to keep them low cost. The various cards look similar as the physical interfaces are defined by a simple ISO standard. Under the hood only the most expensive models have processors and storage which allow for the computation required for public key cryptography. Cards designed for security applications tend to have relatively expensive, tamper resistant, dedicated cryptographic co-processors whose design tends to be limited to a small set of algorithms. That said, the form factor and secure execution environments make smart cards the ideal identity token. For Identity based encryption to gain wide spread acceptance it was essential that pairings be seen to work inside these constrained devices.

3.1 Introduction

Pairings on elliptic curves are fast coming of age as cryptographic primitives for deployment in new security applications, particularly in the context of implementations of Identity-Based Encryption (IBE). In this chapter we describe the implementation of various pairings on a contemporary 32-bit smart-card, the Philips HiPerSmart$^{\text{TM}}$, an instantiation of the MIPS-32 based SmartMIPS$^{\text{TM}}$ architecture. Three types of pairing are considered, first the standard Tate pairing on a nonsupersingular curve $E(\mathbb{F}_p)$, second the Ate pairing, also on a nonsupersingular curve $E(\mathbb{F}_p)$, and finally the η_T pairing on a supersingular curve $E(\mathbb{F}_{2^m})$. We demonstrate that pairings can be calculated as efficiently as classic cryptographic primitives on this architecture, with a calculation time of as little as 0.15 seconds.
The appreciation that the Weil and Tate pairings can be used for constructive cryptographic application has caused a minor revolution in cryptography. After a flurry of research results involving new protocols based on new but plausible security assumptions, it is time for the first commercial applications to start appearing. The final, and perhaps most demanding, niche for the implementation of many cryptographic protocols is in the smart-card, a constrained computing environment in which private keys can be adequately protected. It is the purpose of this chapter to demonstrate that such implementations are perfectly feasible.

There have been two previous reported implementations of pairings on smartcards, the first in the form of an announcement by Gemplus (Now Gemalto) [41], and the second in a paper by Bertoni et al. [11]. There have also been proposals for implementations, such as that by Granger et al. [43], which would require special supporting hardware. Bertoni et al. report a timing of 752 milliseconds on a 33MHz ST22 32-bit smartcard [11], for the same level of security as considered here.

As our chosen smart-card has special support for multiprecision arithmetic over $\mathbb{F}_p$, and over $\mathbb{F}_{2^m}$, we will restrict our attention here to these two cases, although the field $\mathbb{F}_{3^m}$ has undoubted advantages (with its nice embedding degree $k = 6$) and has received considerable attention in the context of pairing based cryptography [43].

3.2 The SmartMIPS^{TM} architecture

The SmartMIPS^{TM} specification is of an instruction-set enhanced version of the popular RISC MIPS32 architecture The enhancements are designed to improve the performance of popular cryptographic algorithms, and are largely those envisaged and described by Großschädl and Savas [44]. It is interesting to note that this new generation of 32-bit smartcards do not employ a classic cryptographic co-processor, with its restricted and specialised set of operations, but rather use carefully selected instruction set enhancements, which when combined with the improved overall performance of the 32-bit chip, permit standard cryptographic algorithms to be executed with sufficient speed. It is also fortunately flexible enough to efficiently support new algorithms that were not envisaged when the processor was being designed.

The main idea is that an extended `ACX|HI|LO` triple of registers can be used to accumulate the partial products that arise when employing the popular Comba/Montgomery technique for multi-precision multiplication [44]. This is supported by a modified `MADDU` instruction which carries out an unsigned integer multiplication and addition to the triple register. Another important addition to the instruction set is the inclusion of a `MADDP` instruction which supports binary polynomial multiplication, and which therefore supports field multiplication over $\mathbb{F}_{2^m}$. For many years algorithms over this field have been disadvantaged with respect to the field $\mathbb{F}_p$ by the absence of such an instruction in standard processors. The addition of this instruction finally "levels the playing field", and allows the full potential of fast arithmetic over the field $\mathbb{F}_{2^m}$ to be realised.

One disadvantage of the MIPS architecture for multi-precision integer arithmetic is the lack of a carry flag, and specifically an add-with-carry `ADC` instruction. In fact it takes 5 instructions just to process one digit in a multi-precision integer addition in order to handle the carry-in and carry-out correctly, not including memory loads and stores. Note however that this is not an issue in $\mathbb{F}_{2^m}$ as in this context addition is carry-free.

When considering the performance of any processor the CPU performance equation [46] is relevant

$$\texttt{CPU Time} = \frac{\texttt{Number of Instructions} \times \texttt{Average Clocks Per Instruction}}{\texttt{Clock Speed in cycles per second}}$$

As instantiated by the Philips HiPerSmart$^{\text{TM}}$ our targeted processor is characterised by

- A five stage pipeline
- Maximum clock speed of 36MHz
- 2k Instruction cache
- 256k Flash memory
- 16k RAM memory

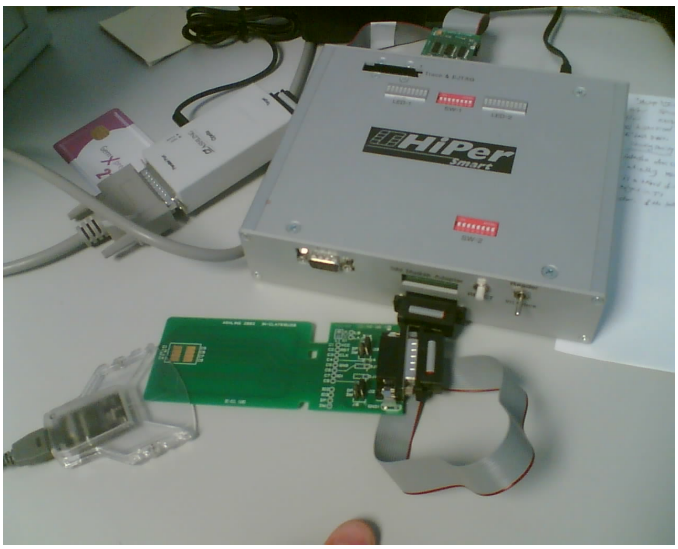

Figure 3.1: Philips HiPerSmart Development Rig

One of the most significant attributes from a programming point of view is the small size of the 2-way associative instruction cache. The MIPS processor as described in [46] is very much designed as a classic RISC processor, which can benefit enormously from loop-unrolling as is indeed the default behaviour of `GCC -O3` compiler optimization. However this is entirely inappropriate with such a small instruction cache. Cache misses are very expensive, and are the main reason for increased CPI (Clocks-Per-Instruction), leading to

poorer performance. Ruthless loop unrolling can dramatically decrease overall instruction count, but only at the cost of much poorer CPI.

While the majority of instructions can complete one pipeline stage per clock tick, certain combinations of instructions will cause a stall in the pipeline. Most of these stalls can be identified and avoided by instruction scheduling (re-ordering). A typical cause for such a stall might be the latency of a multiply instruction like `MADDU`. However as pointed out in [44] these potential performance hits can be avoided if we use the right algorithm. While such pipeline stalls increase CPI, they do so in a fashion which is independent of the clock speed. Cache capacity misses must happen given the small size of the cache, and furthermore conflict misses are inevitable given that the cache is only 2-way associative. These cache misses exact a cost in wasted cycles which can increase dramatically with clock speed, as the access time of main memory becomes much slower than the 1-cycle access time of a cache hit.

3.3 Calculating the Pairing

We consider the scenario in which a smart-card is required to carry out IBE decryption, using either the IBE method of Boneh and Franklin [14] or the method of Sakai and Kasahara as described in [19]. In both cases the critical calculation to recover the plaintext is of the pairing $e(A, B)$, where A is the recipient's private and constant key, and B is a public and variable value associated with the ciphertext. For provable chosen ciphertext security an additional point multiplication is required in both cases, but this is multiplication of a constant point and so fast methods can be used. We omit a formal description of either scheme and instead refer the interested reader to the referenced material.

Much effort has been made to optimize the Tate pairing. In this work we will describe an implementation of the pairing over a prime order finite field $\mathbb{F}_p$ using the BKLS algorithm [2], as described by Scott [85], an implementation of the Ate pairing [47], and an implementation over the small characteristic field $\mathbb{F}_{2^m}$ using the η_T pairing approach described in [5]. In all cases we will exploit the setting in which the pairing is to be calculated to maximize

performance.

3.3.1 The BKLS pairing algorithm

All algorithms for calculating a pairing are elaborations and improvements of the basic Miller algorithm [69]. This particular variation [2] has general applicability to pairing-friendly elliptic curves $E(\mathbb{F}_p)$, either supersingular or non-supersingular. In this case we choose to use an embedding degree of 2 with a non-supersingular curve, very much following the description given in [85]. We use the same non-supersingular curve as described there, where p is a 512-bit prime number and r is the low Hamming weight Solinas prime $2^{159} + 2^{17} + 1$. The point Q is handled as a point on the twisted curve $E'(\mathbb{F}_p)$. Since $p = 3 \bmod 4$, elements of the extension field $\mathbb{F}_{p^2}$ such as m can be described as $m_R + im_I$, where i is the "imaginary" square root of the quadratic non-residue -1.

The helper function $g(.)$ calculates the line functions required by Miller's algorithm, and returns a value in $\mathbb{F}_{p^2}$. This function in turn requires a function *A.add(B)* which adds the elliptic curve points $A = A + B$ using standard methods, and returns the slope of the line joining A and B.

Algorithm 3.1 Function $g(.)$

INPUT: A, B, Q

1: **let** $A = (x_i, y_i)$, $Q = (x_Q, y_Q)$
2: $\lambda_i = A.add(B)$
3: **return** $y_i - \lambda_i(x_Q + x_i) - i.y_Q$

Algorithm 3.2 Computation of the Tate pairing $e(P, Q)$ on $E(\mathbb{F}_p) : y^2 = x^3 + Ax + B$ where P is a point of prime order r on $E(\mathbb{F}_p)$ and Q is a point on the twisted curve $E'(\mathbb{F}_p)$

INPUT: P, Q

1: $m = 1$
2: $A = P$
3: $n = r - 1$
4: **for** $i \leftarrow \lfloor \lg(r) \rfloor - 1$ **downto** 0 **do**
5: $\quad m = m^2 \cdot g(A, A, Q)$
6: $\quad$ **if** $n_i = 1$ **then** $m = m \cdot g(A, P, Q)$
7: **end for**
8: $m = \bar{m}/m$
9: **return** $V_{(p+1)/r}(m_R)$

After the Miller loop, the value of m needs to be subject to a final exponentiation to the power of $(p-1)(p+1)/r$. This is done in two parts – first we calculate m^{p-1} using a conjugation and a division, and then we use a Lucas sequence to raise this value to the power of $(p+1)/r$. The returned value is thus compressed to a single element in $\mathbb{F}_p$ [86].

Observe that the parameter P is in effect being multiplied by its group order r using a standard double-and-add method. The points generated as a result of this process (the x_i and y_i in the $g(.)$ function), and the associated line slopes λ_i, can be precalculated and stored if P is a constant, which it will be in the context under consideration here – in fact its the IBE private key of the card-holder.

Therefore we will precompute and store the points (x_i, y_i, λ_i) that arise in the multiplication of P by r. This results in a much simplified algorithm, where the expensive *A.add(B)* function is no longer required and curve points can be represented using simple affine coordinates.

3.3.2 The Ate pairing algorithm

The Ate pairing [47] is calculated faster than the Tate pairing over non-supersingular curves $E(\mathbb{F}_p)$ if $\lg(t)/\lg(r)$ is less than one, as it uses a truncated Miller loop of length $\lg(t)$ instead of $\lg(r)$ as required above. It was once considered "natural" when implementing the Tate pairing on non-supersingular curves with embedding degree $k \geq 4$, that the first parameter P should be on the the curve defined over the base field $E(\mathbb{F}_p)$ and that the second parameter Q should be a point on a twist of the curve $E'(\mathbb{F}_{p^{k/d}})$, where d can always be 2 [4], but can be as high as 6 for certain curves, such as the BN curves [6]. The authors of [47] however observed that, rather counter-intuitively, the Ate pairing idea works best with P on $E'(\mathbb{F}_{p^{k/d}})$ and Q on $E(\mathbb{F}_p)$. In our application this swapping of roles is not an important issue, as P will be fixed and its multiples can be precalculated and stored as above. More important is the fact that we can get away with a possibly much shorter Miller loop, and still calculate a viable bilinear pairing.

To exploit the Ate pairing we first need a family of elliptic curves which have the required properties. Not only must they be pairing-friendly, but to get the full advantage we want

$\lg(t) < \lg(r)$. The best that can be hoped for is that $\lg(t)/\lg(r) = 1/\deg(\Phi_k(x))$, where $\Phi_k(x)$ is the k-th cyclotomic polynomial [47]. So for a $k = 12$ curve such as that described in [3], the loop may be shortened to as little as one-quarter size. However for our targeted level of security, $k = 12$ is too big. Consider instead the family of elliptic curves defined by

$$\begin{aligned} x &= (Dz^2 - 3)/4,\ t = x + 1,\ r = x^2 + 1 \\ p &= (x^3 + 13x^2 + 26x + 13)/25, \quad \#E = ((x + 13)r)/25 \end{aligned}$$

It can easily be verified that these parameters define a family of pairing-friendly elliptic curve with embedding degree $k = 4$, and with complex multiplication by $-D$. Note that $r = \Phi_4(x)$, and that $\lg(t)/\lg(r) = 0.5$ which is optimal, and so we can leverage the maximum advantage from the Ate pairing idea with a half-length loop. The actual parameters of a curve in the form $y^2 = x^3 + Ax + B$ can then be found using the method of complex multiplication [54]. By choosing random z such that p is prime and 256 bits in length, then we can easily find a value for r which has a 160-bit prime divisor. In this way the conditions that $k.\lg(p) = 1024$ and $\lg(r) = 160$ can be satisfied. For our particular curve, $t - 1$ has a relatively low Hamming weight of 31, and the discriminant $D = 259$. The full algorithm can now be given

Algorithm 3.3 Function $g(.)$

INPUT: A, B, Q

1: **let** $A = (x_i, y_i)$, $Q = (x_Q, y_Q)$
2: $\lambda_i = A.add(B)$
3: **return** $i^2 y_Q - i(i^2 y_i/2 + \lambda_i(i^2 x_i/2 + x_Q))$

In this case the function $g(.)$ returns a value in $\mathbb{F}_{p^4}$ and the Ate pairing returns a compressed value in $\mathbb{F}_{p^2}$. Since we choose $p = 5 \bmod 8$, -2 is a quadratic non-residue in $\mathbb{F}_p$ and $\sqrt{-2}$ is a quadratic non-residue in $\mathbb{F}_{p^2}$, elements in $\mathbb{F}_{p^4}$ can be represented as a pair of elements in $\mathbb{F}_{p^2}$, $m = m_R + im_I$ with $i = (-2)^{1/4}$ [73]. In the function $g(.)$, points on the twisted curve $E'(\mathbb{F}_{p^2})$ must first be converted to coordinates on $E(\mathbb{F}_{p^4})$, which explains the apparent complexity of this function. However given that these can all be precalculated,

Algorithm 3.4 Computation of the Ate pairing $a(P,Q)$ on $E(\mathbb{F}_p) : y^2 = x^3 + Ax + B$ where P is a point of prime order r on the twisted curve $E'(\mathbb{F}_{p^2})$ and Q is a point on the curve $E(\mathbb{F}_p)$

INPUT: P, Q
1: $m = 1$
2: $A = P$
3: $n = t - 1$
4: **for** $i \leftarrow \lfloor \lg(n) \rfloor - 1$ **downto** 0 **do**
5: $\quad m = m^2 \cdot g(A, A, Q)$
6: $\quad$ **if** $n_i = 1$ **then** $m = m \cdot g(A, P, Q)$
7: **end for**
8: $m = \bar{m}/m$
9: **return** $V_{(p^2+1)/r}(m_R)$

this is not an issue in practice.

3.3.3 The BGOhES pairing algorithm

On the supersingular curve

$$E(\mathbb{F}_{2^m}) : y^2 + y = x^3 + x + 1$$

where m is prime and $m = 3 \bmod 8$, the number of points is $2^m + 2^{(m+1)/2} + 1$ [5]. For our choice of $m = 379$, this value is a prime. A suitable irreducible polynomial for the field $\mathbb{F}_{2^{379}}$ is $x^{379} + x^{315} + x^{301} + x^{287} + 1$. This supersingular curve has an embedding degree of $k = 4$. To represent the quartic extension field $\mathbb{F}_{2^{4m}}$, we use the irreducible polynomial $X^4 + X + 1$.

Recall that in a characteristic 2 field with a polynomial basis, field squarings are of linear complexity. Furthermore on this supersingular curve, point doublings require only cheap field squarings (using affine coordinates). Therefore we can anticipate that calculations on this curve will be very efficient.

A distortion map for this particular supersingular curve is $\psi(x, y) = (x + s^2.y + sx + t)$, where $t = X$ and $s = X + X^2$ [66]. A major insight from [5] is that the Tate pairing can be calculated from the more primitive η_T pairing, which requires a half-length loop compared to the Duursma-Lee method [34], with considerable computational savings. The algorithm

as described benefits from unrolling the loops times 2, in which case each iteration costs just seven base field multiplications. The final exponentiation looks a little complex, but in fact can be accomplished with only 4 extension field multiplications, $(m+1)/2$ cheap extension field squarings and some nearly-free Frobenius operations.

Algorithm 3.5 Computation of $\hat{e}(P,Q)$ on $E(\mathbb{F}_{2^m}) : y^2 + y = x^3 + x + b : m \equiv 3 \pmod 8$ case

INPUT: P, Q
OUTPUT: $\hat{e}(P,Q)$
1: **let** $P = (x_P, y_P)$, $Q = (x_Q, y_Q)$
2: $u \leftarrow x_P + 1$
3: $f \leftarrow u \cdot (x_P + x_Q + 1) + y_P + y_Q + b + 1 + (u + x_Q)s + t$
4: **for** $i \leftarrow 1$ **to** $(m+1)/2$ **do**
5: $\quad u \leftarrow x_P$, $x_P \leftarrow \sqrt{x_P}$, $\quad y_P \leftarrow \sqrt{y_P}$
6: $\quad g \leftarrow u \cdot (x_P + x_Q) + y_P + y_Q + x_P + (u + x_Q)s + t$
7: $\quad f \leftarrow f \cdot g$
8: $\quad x_Q \leftarrow x_Q^2$, $\quad y_Q \leftarrow y_Q^2$
9: **end for**
10: **return** $f^{(2^{2m}-1)(2^m - 2^{(m+1)/2}+1)(2^{(m+1)/2}+1)}$

Since P will be fixed, all the square roots in this algorithm can be precalculated and stored with some savings. With this modification, our implementation is largely the same as that described in [5].

3.4 Implementation Issues

Our implementation makes use of the MIRACL multiprecision library [84]. This library is friendly towards those attempting implementations in a constrained environment, like a smartcard. Typically a big number library forces allocation of memory for big variables from the heap. In a constrained environment however a heap is a luxury that often cannot be afforded. Therefore allocation from the stack is appropriate. Header file definitions were used to cut down the amount of code required. This was supplemented with some manual pruning of unwanted functionality.

For optimal performance MIRACL includes a mechanism for generating unrolled Comba code for modular multiplication, squaring, and reduction with respect to a fixed modulus, including specific support for the SmartMIPS$^{\text{TM}}$ processor. However as pointed out above,

fully unrolled code is inappropriate in an environment where the instruction cache is very small. Therefore we found it necessary to take the automatically generated (and correct) code, and to roll it up again into tight loops, much as described in [44]. Extra manually written inline assembly code was provided to support fast squaring in $\mathbb{F}_{2^m}$ using the `MADDP` instruction, and short unrolled assembly language code was provided for fast field addition in $\mathbb{F}_{2^m}$. With these exceptions, the rest of the code was written in standard C.

Precomputation was used to advantage in all cases. The amount of ROM required to store precomputed values was 31232, 25036 and 18432 bytes respectively, for the Tate, Ate and η_T pairing. The RAM requirement in all cases was comfortably with 16K available, typically requiring only half of that. As stack memory is inherently re-usable, a simple restructuring of the programs could reduce this requirement still further.

3.5 Results

We present our results in a series of tables. As well as the timings for the pairings, we include timings for (non-fixed) point multiplications and pairing exponentiations, these as often relevant to pairing based protocols. For each of the three implementations we assume projective coordinates are used for point multiplication, as field inversions which are required for affine point addition are very slow on the smartcard. The point multiplication is taken over the base field $E(\mathbb{F}_q)$ using a random 160-bit multiplier. Field exponentiation is of the pairing value to a random 160-bit exponent. For the $E(\mathbb{F}_p)$ cases we use Lucas exponentiation (also known as a "Montgomery powering ladder") of the compressed pairing, while for the $E(\mathbb{F}_{2^{379}})$ case we use standard windowed exponentiation, as we believe these to be the fastest methods in each case.

Our hardware emulator is only cycle accurate up to 20.57MHz, and so we estimate the timings for the maximum supported speed of 36MHz, using linear interpolation for CPI. For comparision purposes we include figures for 1024-bit RSA decryption (using the Chinese Remainder Theorem), and timings on a standard PC (note that these are faster than previously reported timings, due to their implementation in C rather than C++).

Table 3.1: Instructions required (% icache misses) - Philips HiPerSmart^{TM}

	$E(\mathbb{F}_{2^{379}})$ η_T pairing	$E(\mathbb{F}_p)$ Tate pairing	$E(\mathbb{F}_p)$ Ate pairing
Pairing	3705344 (10.9%)	7753341 (7.3%)	8156645 (15.8%)
Point Mult.	2589569 (9.6%)	7418768 (6.1%)	2663217 (17.5%)
Field exp.	1551117 (11.4%)	1364124 (7.2%)	1614016 (15.7%)
RSA decryption	4372772 (3.4%)		

Table 3.2: Clock cycles required/CPI/time in seconds @ 9 MHz

	$E(\mathbb{F}_{2^{379}})$ η_T pairing	$E(\mathbb{F}_p)$ Tate pairing	$E(\mathbb{F}_p)$ Ate pairing
Pairing	4311454/1.16/0.48	9104450/1.17/1.01	10860479/1.33/1.21
Point Mult.	3118344/1.20/0.35	8529176/1.15/0.95	3739596/1.40/0.42
Field exp.	1924596/1.24/0.21	1593313/1.17/0.18	2122221/1.31/0.24
RSA decryption	4740271/1.08/0.53		

Table 3.3: Clock cycles required/CPI/time in seconds @ 20.57 MHz

	$E(\mathbb{F}_{2^{379}})$ η_T pairing	$E(\mathbb{F}_p)$ Tate pairing	$E(\mathbb{F}_p)$ Ate pairing
Pairing	4590712/1.24/0.22	9755457/1.26/0.47	12207440/1.50/0.59
Point Mult.	3391127/1.31/0.16	9049457/1.22/0.44	4278858/1.61/0.21
Field exp.	2118707/1.37/0.10	1705365/1.25/0.08	2374885/1.47/0.12
RSA decryption	4880323/1.12/0.24		

Table 3.4: Clock cycles required/CPI/time in seconds @ 36MHz (estimated)

	$E(\mathbb{F}_{2^{379}})$ η_T pairing	$E(\mathbb{F}_p)$ Tate pairing	$E(\mathbb{F}_p)$ Ate pairing
Pairing	4891054/1.32/0.14	10467010/1.35/0.29	13621597/1.67/0.38
Point Mult.	3677188/1.42/0.10	9570210/1.29/0.27	4847055/1.82/0.13
Field exp.	2326675/1.50/0.06	1814285/1.33/0.05	2630846/1.63/0.07
RSA decryption	5072415/1.16/0.14		

Table 3.5: Timings in milliseconds on 3GHz Pentium IV

	$E(\mathbb{F}_{2^{379}})$ η_T pairing	$E(\mathbb{F}_p)$ Tate pairing	$E(\mathbb{F}_p)$ Ate pairing
Pairing	3.88	2.97	3.16
Point Mult.	1.82	3.08	1.17
Field exp.	1.14	0.54	0.62
RSA decryption	1.92		

The most surprising and significant observation to be made is that the η_T pairing can be calculated as quickly as a standard RSA decryption, for approximately the same level of security. As expected CPI goes up as clock speed increases, as we are punished more heavily for cache misses. This has less impact on algorithms that spend more time in tight loops, and hence disadvantages the η_T and Ate pairings with their more elaborate structures and higher extension fields. Note that RSA, due to its simplicity, suffers least from increasing CPI.

3.6 Does pairing delegation make sense?

The idea of securely delegating the calculation of a pairing to the terminal was considered in [20]. This was motivated by the assumption that the pairing calculation might be too resource consuming to be carried out on a smartcard. Here we present a slightly modified version of the method described in Section 6.2 of [20]. In the context of IBE decryption the calculation of $e(A, B)$ involves a constant and private A (in fact the IBE private key), and a public B (in fact part of the ciphertext). It is assumed that the smartcard also has stored a random secret point Q and the value of $e(A, Q)$.

- The card generates random x,y,and z, and queries the following pairings to the terminal.

$$\alpha_1 = e(x^{-1}A, B), \quad \alpha_2 = e(yA, z(B + Q))$$

- The card computes

$$e_{AB} = \alpha_1^x$$

- The card checks that

$$\alpha_1^r = 1, \quad \alpha_1^{xyz \bmod r} = \alpha_2 / e(A, Q)^{yz \bmod r}$$

If successful the protocol outputs $e(A, B) = e_{AB}$. Observe that two of the point multiplications are of the fixed point A. These may be calculated offline, or at the very least can benefit from fast methods for fixed-point multiplication. Also $e(A, Q)^{yz}$ can be precalculated, or calculated using fixed-base exponentiation [67]. So the major online cost will be of 3 exponentiations and one point multiplication. From the tables above it is clear that the η_T pairing is so fast that delegation is unlikely to be beneficial. The standard Tate pairing ($k = 2$) implementation suffers badly as point multiplication is over a large 512-bit field. However in the case of our Ate pairing implementation, with its smaller 256-bit field size, it appears that delegation might be beneficial.

3.7 Conclusions

We have demonstrated for the first time that cryptographic pairings can be implemented just as quickly as classic public key cryptographic operations on a standard smartcard, hence clearing the way for their more widespread adoption. The issue of pairing delegation has been investigated, and it appears that despite the efficiency of our implementations, it may be advantageous in certain circumstances.

4

Pairing Friendly Curves Search

4.1 Introduction

The work carried out for Chapter 3 left many major questions about how cryptographic pairings would be used in practice. Many researchers developed protocols which assumed an implementation "black box" that would be fast, robust, secure, and implement standards akin to the Public-Key Cryptography Standards (PKCS) [80] or NIST standard curves (see Section 7.1.4) for interoperability etc. However, it is early days with respect to the industrial acceptance of pairing base cryptography.

In order to further the adoption of the field of pairing based cryptography, we attempted to reduce the overhead for researchers by solving an open problem of locating *pairing friendly elliptic curves* (Section 2.4.4) by systematically documenting all available curves in a wide search space using super-computing resources. An open problem to its successful adoption is the arbitrary selection of elliptic curves by different research teams, often without understanding critical security and/or performance issues associated with their choice. We tried to produce the definitive list of known "friendly" curves for use by research and industry worldwide. At the present time, finding even one such candidate "friendly" curve can take a number of days on a standard PC. We used the computation power available at the "Irish Centre for High-End Computing" (ICHEC) [53] to pre-compute suitable curves to embed in low-end applications such as the smart card applets. Overall, from start to finish, the

project duration was 9 months.

4.2 Calculating Pairing Friendly curves

Freeman Scott and Teske [38] published a taxonomy that encompasses all of the constructions of pairing-friendly elliptic curves currently in existance. Their paper outlines three general methods to find solutions that have been proposed.

- 1. The MNT/Freeman idea (MNT/F) [70], [37]
- 2. The Barreto-Lynn-Scott/Brezing and Weng idea (BLS/BW) [3], [16]
- 3. The Scott-Barreto idea (SB) [87]

In order to find a pairing friendly elliptic curve, it is imperative to find a curve such that $q|p^k - 1$, where q divides the number of points on the curve n. However, if we are to find an actual curve we also require the parameters of the curve (n, p and t) to satisfy the CM condition (Complex Multiplication See [27]). This is that $4p - t^2$, or equivalently $4n - (t-2)^2$, should factor as DV^2, where D is relatively small, less than about 1012. Bear in mind that $n = p + 1 - t$.

Now the the pairing friendly condition $q|p^k - 1$ can be transformed to the condition that $q|\Phi_k(t-1)$, where Φ_k is the k-th cyclotomic polynomial [3]. The k-th cyclotomic polynomial is the new factor that appears in the factorisation of $\Phi_k(x)$. For example:

$$\begin{aligned}
\Phi_1(x) &= (x-1) \\
\Phi_2(x) &= (x+1) \\
\Phi_3(x) &= x^2 + x + 1 \\
\Phi_4(x) &= x^2 + 1 \\
&.... \\
\Phi_{12}(x) &= x^4 - x^2 + 1
\end{aligned}$$

One approach to finding pairing friendly curves is to try and satisfy the polynomial equation

$$DV(x)^2 = 4c(x) \times \Phi_k(t(x) - 1) - (t(x) - 2)^2$$

for some cofactor $c(x)$ (which would ideally be 1), some $t(x)$, and a small value for D. However it is also sufficient that q should divide some factor of $\Phi_k(t(x) - 1)$, and so if this factors into two or more irreducible polynomials, these polynomial factors can also be used as candidates.

One approach [70], [37] is to try and force the RHS of the CM equation to be a quadratic in x. Should this be the case, solutions can readily be found, as the CM equation can be forced into the form of the well-known Pell equation.

A second approach is to use a brute force search through $c(x)$ and $t(x)$ to find a satisfactory solution. For example if we choose $c(x) = ((x-1)^2)/3$, and $t(x) = x + 1$, then the RHS of the CM equation simplifies to $3((x-1)(2x-1)/3)^2$, and so we have a satisfactory solution with $D = 3$ [3].

Our search programs typically attempted to search through the space of $c(x), t(x), k$ and D to find pairing-friendly curves.

4.3 Implementation

4.3.1 Hardware

DCU

We initially constructed a small test cluster based on a 24 450Mhz Intel Celerons to fine tune the application and discover potential resource requirements. This cluster was configured to run OSCAR [74], a Beowulf-type, high-performance computing cluster compatible with more commercial super-computing resources. One of OSCAR's strengths is that it is possible to install multiple *message passing interface* (MPI) implementations on one cluster and easily switch between them. While this facilitated our development ramp-up speed, our

Figure 4.1: DCU cluster

hardware set-up was too undersized to address the specific problem.

ICHEC

Figure 4.2: ICHEC

We then moved our code to ICHEC whose mission is to provide high-performance computing (HPC) resources for researchers in Irish third-level institutions. We successfully applied for *class C* level resources which allowed us to use 25,000 Core Hours on Opteron 2.4GHz CPUs.

4.3.2 Software

We designed a reusable framework using MPI, NTL and GnuMP which could plug-in any polynomial factoring program. The configuration established an embedding degree, available number of terms for the polynomial and a range for the co-efficient of each term.

Table 4.1: K=34 16 CPUs 5 terms

cpu time	mem (kb)	vmem (kb)	wall time
62 : 05 : 26	1,539,924	1,918,304	04 : 11 : 39

The solution set is found by means of a set of custom written polynomial factoring programs which have been developed in standard C++ on Linux with output in text format to standard file type. Subsequently, through the course of a *class C* project on ICHEC we managed to further optimise the processing to be more aware of the resources available (master/slave model for MPI). The application is designed to grow with the substantial CPU resources required but has modest memory and data storage overheads. The *class C* module environment platform comprised mpich/gcc. However, we do believe the application to be portable to any Unix, MPI or compiler environment.

There is a dependence on the math libraries NTL and GnuMP

- NTL : A Library for doing Number Theory
- Gnu MP : GNU Multiple Precision Arithmetic Library.

Our code is C++ and Assembly. It is not dependent on a specific number of CPUs but adapts to the number provided by the scheduler. There is little dependency between nodes during execution except for internal job creation and some I/O blocking when a rare result hit occurs. This limited communication between nodes restricts the network and memory loads. It is difficult to fully determine the performance characteristics of our system as execution requirements depend on false positives passing tests, which require some computation, but later are rejected by failing subsequent tests.

With our experiments, we typically segmented our jobs to run on 16 nodes. Each unit of work [1] used CPU time outlined in Table 4.3.2.

With the class C proposal our *width* (range of coefficients) was usually ±12 and we worked with 5 terms in a polynomial. We would normally use between 16 and 32 of these runs to satisfy a search space which is about 1984 CPU hours per experiment. We had

[1] based on the MNTF method

Table 4.2: K=10 32 CPUs 8 terms

cpu time	mem (kb)	vmem (kb)	wall time
432 : 16 : 35	62,496,600	94,722,904	23 : 34 : 35

3 different experiment tests to run all with similar computation overhead. This gave us roughly 6000 hours total CPU time. We also needed some development time and therefore ran some of the experiments on our own limited cluster.

4.4 Results and Future Work

Our class C project did validate known curves, but also found a number of new (or new variations) of known curves. However, from these results we've established that the wider space used (from our estimation of what would fit inside a class C project's CPU time) didn't produce any significantly different results.

Future work

For the future, we would see a ICHEC class B project where we aim to *flatten* the search space by reducing the range of coefficients but extend the *depth* by increasing the number of terms. That is set the range at ± 3 but with 8 terms rather than the existing 4. This will greatly increase the computations per experiment. Table 4.4 lists an equivalent execution to Table 4.3.2 with the 4 extra terms added.

This increases to about 7 times the computation required per unit of work. Rolling this through the various experiments, we would see a total of about $42,000$ hours for perfect running experiments. We could also attempt a small number of larger runs with the width set at ± 12 and with up to 12 terms on likely polynomials (where most of the results have been so far).

Future work would include conducting more refinement on our program code to introduce checkpoints and restart capabilities to allow for better use of resources if, for whatever reason a job fails to complete, it would be possible to restart from last know good position.

Runtime

Each attempt at mapping a search space takes about 24 wall hours. As previously stated, we require about 16 runs for each of the 3 experiments to gather enough data to produce a judgement and potentially enough for publication. We would then attempt to completely document the full space by completing missing runs.

Chapter Acknowledgements

We wish to thank the SFI HEA Irish Centre for High-End Computing (ICHEC) for the provision of computational facilities and support.

5

The Cell Broadband Engine

Background

While working with the super-computing resources in our search for pairing friendly curves (Chapter 4), we became aware of ambitious plans from IBM/Toshiba/Sony to launch a new processor which would challenge the performance levels achieved by current super-computers. The published specifications indicated that these processors were very interesting from a cryptographic point of view and that this radical design could break all existing speed records. While our research focus switched from slow constrained devices to a processor design with seemingly unlimited scalability, the techniques for optimal efficient utilisation were very similar.

5.1 Introduction

Recently, the major performance chip manufacturers have turned to multi-core technology as the more cost-effective alternative to ever increasing clock speeds. Well known examples of multi-core architectures include the Intel Core Quad and AMD Phenom X4 X2 range of chips. IBM have introduced the Cell Broadband Engine (more commonly referred to as 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^{TM}. The Cell contains a number of specialist synergistic processor units (SPUs) optimised for multimedia processing and offers a rich, vector processing based API to developers. 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.

We started this project in 2005 before any Cell hardware existed. IBM released a full, cycle accurate simulator that could run linux inside a virtual machine hosted on a linux platform enveloped inside a Tk/Tcl environment that allowed for close monitoring of the instructions as they ran. The SDK is extremely powerful and a great example of a modern linux development environment. It allows one to configure processor descriptions such as instruction latency times for what-if scenarios for theoretical CPUs.

In November 2006 Sony made the Cell available inside the Playstation 3. It was released initially in Japan and North America, with a European release in March 2007. While not fully supported by the IBM SDK, we were anxious to see our code run on actual physical hardware. We imported a Japanese model, quickly reformatting it to run Linux, and we installed the SDK to see if our simulated 'theoretical' matched the actual results. We were not disappointed (and possibly had the first playstation 3 in Ireland!). Later, IBM provided a number of academic institutions with Cell blade resources accessible on the web.

5.2 The Cell Broadband Engine

The Cell has 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 (204 GB/s) [48] offering a multi-core environment with two-instruction sets and enormous processing power. Central to the 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 last generation PowerMac range. This PPU works as the primary processor and as

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

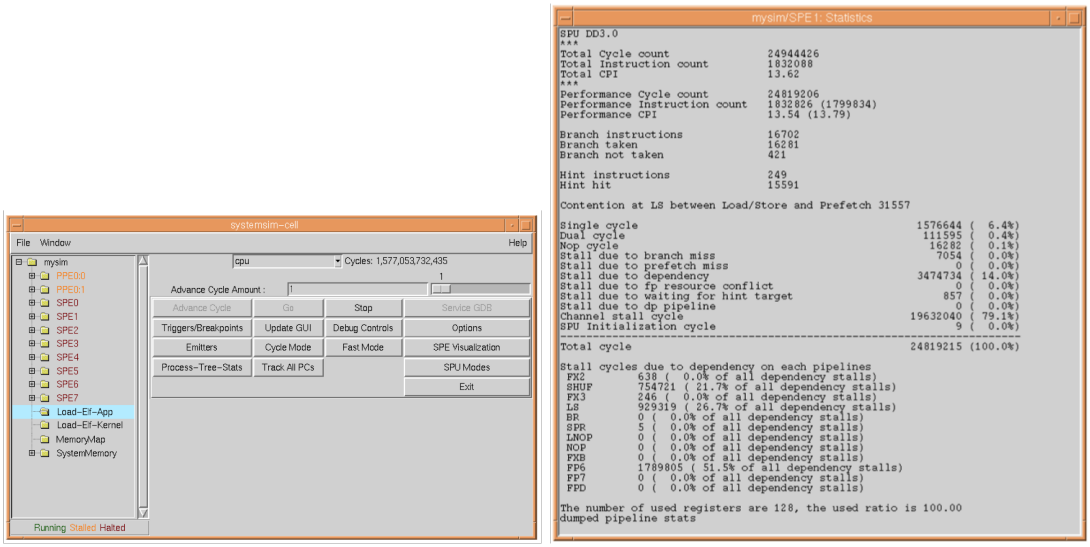

Figure 5.1: Cell Simulator

supervisor for the other cores.

The Cell can be found in the Sony Playstation 3 and the IBM QS20 and QS21 blade server series. Note that the CBE in the Playstation 3 makes just 6 out of 8 SPUs available for general purpose computations. Toshiba equips several laptops of the Qosmio series with the SpursEngine consisting of 4 SPUs intended for media processing. This SpursEngine can also be found in a PCI Express card called WinFast pxVC1100 manufactured by Leadtek which is currently available only in Japan.

5.2.1 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 [2] instruction set and a large (128) array of 128-bit registers. Each SPU has its own local memory store (LS). Currently, this LS is limited to just 256K. The SPU can access the LS in the same clock cycle as its register operations. While the architecture allows for any number of SPUs, a standard Cell, and those currently in production, has 8 SPUs.

A processor with just 256K, no hardware cache and with no access to I/O doesn't appear

[2]Single Instruction Multiple Data

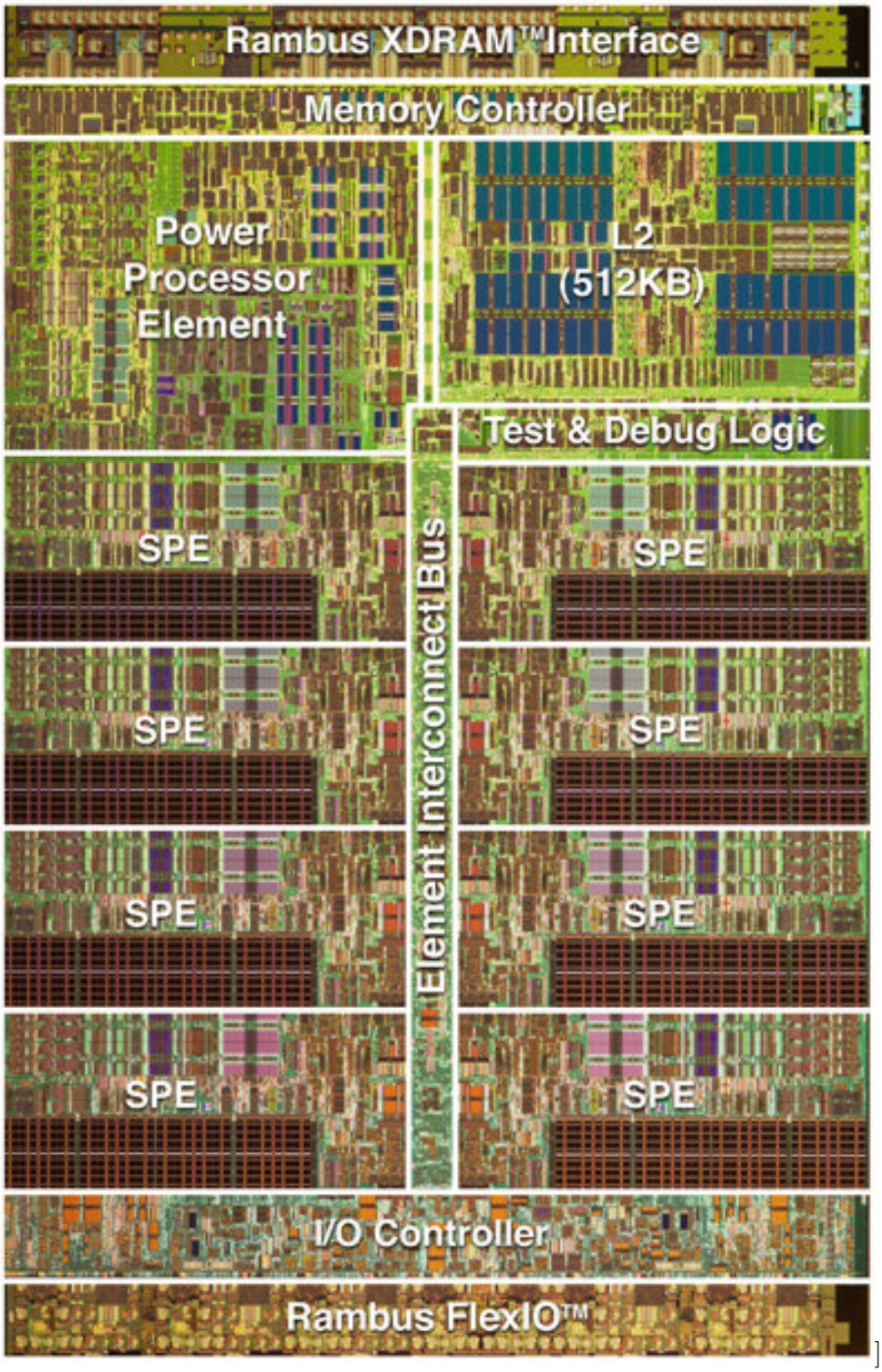

Figure 5.2: Cell BE Die Layout (source IBM marketing material)

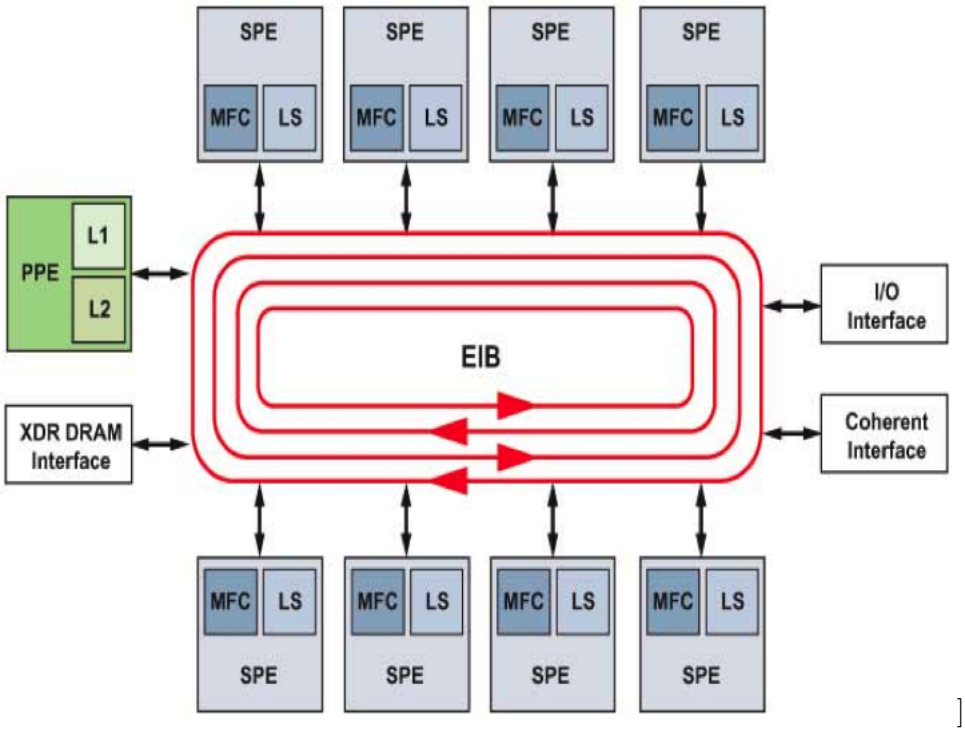

Figure 5.3: Cell BE logical diagram

to be anything exciting when compared to the PPU or other modern CPUs. It is the fact that the Cell offers 8 SPUs on one die all designed to operate in parallel combined 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 complex. 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. 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 direct memory access (DMA) library.

The SPU has two pipelines (pipeline 0 and pipeline 1); each cycle it can dispatch one instruction per pipeline. Whether or not the SPU really dispatches two instructions in a given cycle is highly dependent on instruction scheduling and alignment. This is subject to the following conditions:

- Execution of instructions is purely in-order.
- The two pipelines execute disjoint sets of instructions (i.e. each instruction is either a pipeline-0 or a pipeline-1 instruction).
- The SPU has a fetch queue that can contain at most two instructions.
- Instructions are fetched into the fetch queue only if the fetch queue is empty.
- Instructions are fetched in pairs; the first instruction in such a pair is from an even word address, the second from an odd word address.
- The SPU executes two instructions in one cycle only if two instructions are in the fetch queue, the first being a pipeline-0 instruction and the second being a pipeline-1 instruction and all inputs to these instructions being available and not pending due to latencies of previously executed instructions.

Hence, instruction *scheduling* has to ensure that pipeline-0 and pipeline-1 instructions are interleaved and that latencies are hidden; instruction *alignment* has to ensure that pipeline-0 instructions are at even word addresses and pipeline-1 instructions are at odd word addresses.

The implementations outlined later in this thesis (Chapter 7 and Chapter 8) and the IBM SDK's MPM library build the finite field arithmetic on the integer arithmetic instructions of the SPU. This is due to the fact that single-precision floating-point arithmetic offers too small a mantissa and that double-precision, floating-point arithmetic causes excessive pipeline stalls on the SPU and is therefore very inefficient.

All integer arithmetic instructions (except shift and rotate instructions) are SIMD instructions operating either on 4 32-bit word elements or on 8 16-bit halfword elements or on 16 8-bit byte elements of a 128-bit register.

This simple example is a 4-wide add. Each of the 4 elements in register vector VA is added to the corresponding element in register VB the 4 results are placed in the appropriate slots in reg VC. Obviously, this becomes more complex when one considers overflow after an add.

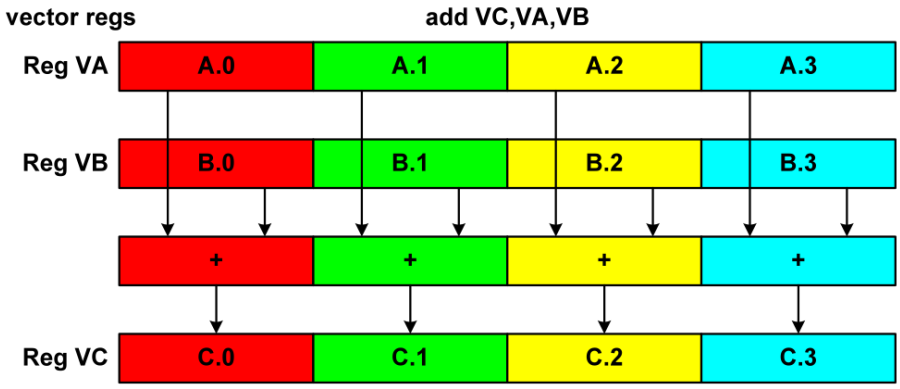

Figure 5.4: A SIMD Instruction example (source IBM training material)

Integer multiplication is an exception to this rule: The integer multiplication instructions multiply 4 16-bit halfwords in parallel and store the 32-bit results in the 4-word elements of the result register.

The following instructions are the most relevant for our implementations; for a detailed description of the SPU instruction set see [50], for a list of instruction latencies and associated pipelines see [49, Appendix B].

`a:` Adds each 32-bit word element of a register a to the corresponding word element of a register b and stores the results in a register r.

`mpy:` Multiplies the 16 least significant bits of each 32-bit word element of a register a with the corresponding 16 bits of each word element of a register b and stores the resulting four 32-bit results in the four word elements of a register r.

`mpya:` Multiplies 16-bit halfwords as the `mpy` instruction but adds the resulting four 32-bit word elements to the corresponding word elements of a register c and stores the resulting sum in a register r.

`shl:` Shifts each word element of a register a to the left by the number of bits given by the corresponding word element of a register b and stores the result in a register r.

`rotmi:` Shifts of each word element of a register a to the right by the number of bits given in an immediate value and stores the result in a register r.

`shufb:` Allows to set each byte of the result register r to either the value of an arbitrary byte of one of two input registers a and b or to a constant value of `0`, `0x80` or `0xff`.

5.2.2 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 `spe_create_thread()` 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 5.5 [21] outlines the build process.

5.2.3 The Cell as a Hardware Security Module

The Cell has been designed with an interesting security architecture [90]. The feature set suggests it is primarily to aid in the management of digital rights, however, the interface is also open to third party developers to implement additional security functionality into code

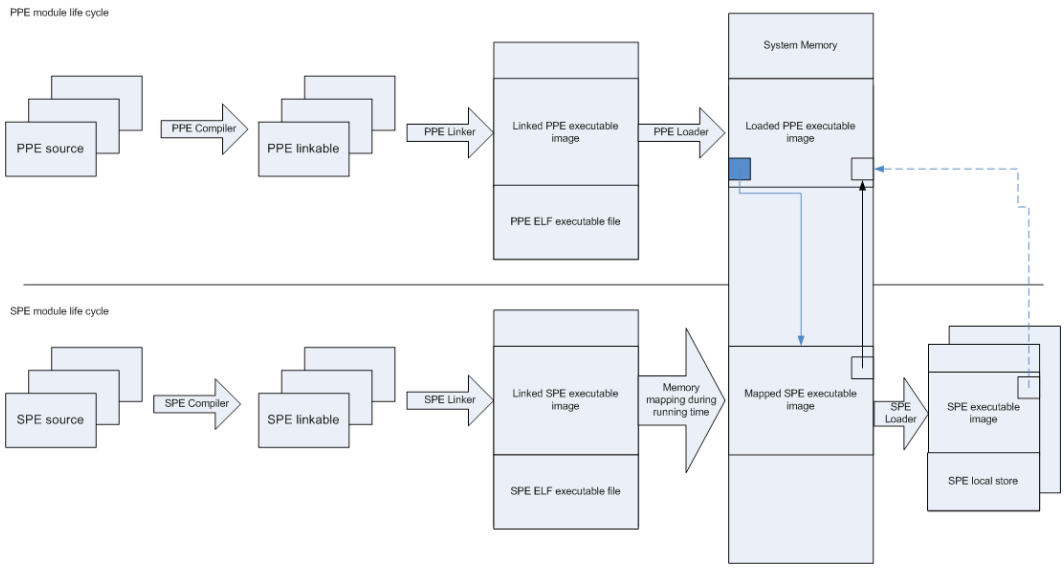

Figure 5.5: Cell BE build process[21]

running on an SPU. This architecture can be used to make security critical code run in a protected environment as a *Hardware Security Module* (HSM). Commercially, most of the SSL accelerator vendors offer HSM's in high-end configurations.

To operate in a more protected environment, the critical SSL code can be run in an SPU in *isolated mode*. For an SSL accelerator, this means that it is possible to have any key generation method make use of a cryptographically secure random number generator, that key data can be protected from other processes running on the Cell, key data can be encrypted in shared memory locations and program code can check its integrity.

The Cell achieves this level of security by implementing a hardware based process in which

1. The code and data in an SPU can be executed in physically isolated memory space.
2. There is hardware based code signing (referred to as secure boot) where the integrity of code about to be executed can be verified.
3. An isolated process in an SPU can use hardware-key based data encryption/decryption that can only be used by code that has been verified.
4. The random number generator can be configured to use a *physical* sample source such

as the Cell Hardware RNG found on some models of the Cell. (note: this functionality is limited on our PS3 based Cell).

This extra security comes with a performance penalty. Initialisation would be affected by secure boot, if data were to be stored off SPU. Then there can be runtime overheads in decryption. The various random number generators degrade in performance as one moves towards a more random, physical sample-based library. Unfortunately implementation details are only available under Non-Disclosure Agreement with IBM and so we are not able to test our SPU acceleration code in this interesting environment. While this should concern people using our accelerator in production environments, the raw performance figures we use throughout this paper reflect the speed of the underlying mathematical operations and would be the same post initialisation.

For further information on the Cell's HSM see IBM's Cell resource centre [28]

5.3 Development

For applications like those constructed in Chapter 6 we need to build PPU libraries (32 or 64-bit) that plug into a PPU build of higher layer libraries through defined interfaces. Inside these libraries we embed an SPU ELF executable which can act upon the 128-bit registers and utilises IBM's MPM library. In other cases, where we were more concerned with raw performance numbers, we can use small spulets. These are SPU ELF binaries, in a minimal PPU wrapper. These can be looked upon as standalone SPU executables, however, we have to use separate performance analysis tools when profiling them and unfortunately these are slightly inaccurate. All these SPU ELF executables need 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 to offload as much as possible 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.

5.3.1 Direct Memory Access

As per 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 informing the PPU, via the tag, that it has done so.

5.3.2 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. We 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.

The following intrinsics 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, we implement an elegant speed up technique which can be used when adding a 128-bit value to a 64-bit value where overflow is not a concern. 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
```

Longer, more detailed code segments are attached in Appendix B. This code is the output of an automated code generator which was written as part of my Ph.D. work. It was used extensively in the remainder of this thesis and in particular to produce the code highlighted in Chapter 7.

Code generation was considered as manually unrolling code by cut n paste leads to hard to find errors. It also offers the most flexibility when considering fixed sizes.

5.3.3 Usage Models

The multi-core architecture, with small SPU core, allows developers to use concurrency and produces the interesting code/data models outlined in 5.6. Simply put run in sequence, run in parallel, or have dedicated parallel processes.

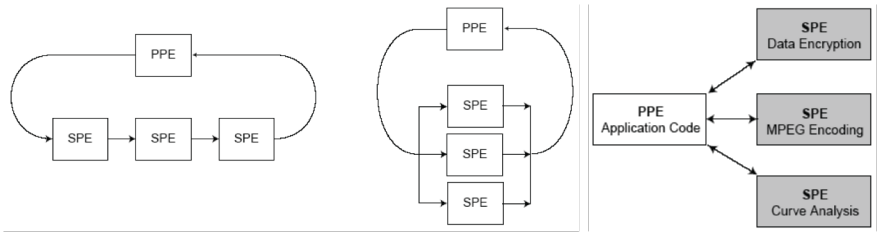

Figure 5.6: Usage models

An enlightening analogy from a coding SPU presentation from Insomniac (the game developers): Old model :- Large semi truck. Stuff everything in. Then stuff some more. Then put some stuff up front. Then drive away. New model :- Fleet of Ford GTs taking off every five minutes. Each one only fits so much. They also say The ultimate goal: Get everything on the SPUs. Leave the PPU for shuffling stuff around. This appears to the design paradigm for all Cell developers.

In Chapter 6 we are more concerned with throughput and follow a model of same code for multi SPUs with the PPU as controller. Whereas in Chapter 7 and Chapter 8, we focus on models where the SPU is viewed a standalone special purpose CPU.

5.3.4 Compilation

There are two C compilers for the Cell, derived ppu-gcc/spu-gcc combination or IBM's commercial XLC. Our experience is that both generate good code, neither is consistently better than the other and that for critical sections of code, both benefit from manual optimisation steps. All numbers are quoted with the highest -O3 optimisation flag set.

Chapter Acknowledgements

For development tools and background information found ourselves repeated turning to IBM's DeveloperWorks resource centre and the Cell SDK. We would like to thank the Cell development community particularly Séan Starke and the IBM team. The authors acknowledge Georgia Institute of Technology, its Sony-Toshiba-IBM Center of Competence, and the National Science Foundation for the use of Cell Broadband Engine resources that have contributed to this research.

6

Accelerating SSL with the Cell Broadband Engine

Background

The Cell development environment came pre-loaded with a commonly used open source network and cryptographic toolkit (see Chapter 6 Section 6.1.1) which is commonly used for processor benchmarking. It allowed us to compare performance levels of this much anticipated processor with traditional desktop and servers. Initial results were not so impressive as the toolkit had not been optimised to take advantage of this exciting new design.

6.1 Why SSL?

Despite huge gains in computing performance and bandwidth, the widespread use of secure communications over the Internet is still essentially limited to SSL connections for password logins or for credit card payments. Despite this, SSL implementations are widely distributed and well analysed, making it the de-facto standard for secure communications. The main reason encryption is limited to logins and payments, and is not 'always on', 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 chapter 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.

6.1.1 OpenSSL

OpenSSL [1] is an open source toolkit released under under a BSD style license and is the de facto open source SSL toolkit. It is included in virtually all UNIX distributions including Linux, MacOSX™, and Solaris™.

The name OpenSSL is misleading as the toolkit provides a vast array of building blocks and interfaces from big number routines, cryptographic primitives through 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 symmetric 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.* [12] 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 [1], and since the main computational 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 Chapter we focus on RSA

[1] One server is expected to deal with many clients which makes accurate sampling difficult

but the improvement is relevant to all.

6.2 Architecture

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) [94]. 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 should perform full RSA decryption using the Chinese Remainder Theorem. Figure 6.1 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:

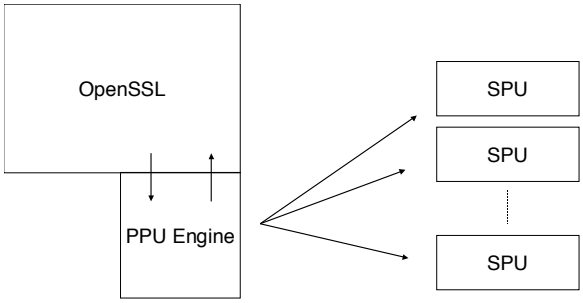

Figure 6.1: OpenSSL with Engine and SPUs

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 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 it breaks the guideline of offloading as much computation as possible to

an SPU. With (3 and 4) the latency per SSL connection will be reduced but, as it adds extra DMA data to the bus, the overall maximum throughput will be affected. With (2, 3 and 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. We implemented (1) and (2) and found the initial speed up to be marginally better but the maximum throughput to be slightly lower. This is explained by an increased amount of SPU invocations. 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 exponentmod$q - 1$. $iqmp$ is the inverse of $q \bmod p$. I_0 is the ciphertext. A decryption exponent d, for a prime p, is a number d, such that $m^{ed} \bmod p = m$ or $ed = 1 \bmod (p - 1)$, where e is the encryption exponent, commonly chosen to be 3 or 65537.

At the RSA initialisation stage OpenSSL passes the $(p, q, dmp_1, dmq_1, iqmp)$ parameters 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 and thread-local memory store. OpenSSL uses this reference again when making calls 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 successful completion 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.

Big number representation Unfortunately there is no standard method for big number representation with most multi-precision systems/libraries choosing variants of

```
struct {
        unsigned int  size;
        <largest basic type>    *words;
}
```

Subtle differences exist. Be it big endian or little endian ordering of the words, or if the number structure keeps track of its own data size. The OpenSSL representation is of the form

```
struct bignum_st{
      BN_ULONG *d;
      int top;
      int dmax;
      int neg;
      int flags;
};
```

BN_ULONG represents the largest underlying type and `dmax`, `neg`, and `flags` hold useful internal management data. On the PPU we can build either a 32-bit or 64-bit binary. We chose 64 and so BN_ULONG is a 64 bit type (`unsigned long long`). The IBM MPM on the SPU can utilise the vector quadword register to contain 8 elements of a smaller 16-bit `unsigned short` type which it can handle very efficiently in its 16x16 multiplier. For example a 12 instruction load and store (LS) has a latency of just 6 clock cycles [30]. The type is implicit and of the simple form

```
vector * unsigned int;
```

Use of the IBM MPM library is, therefore, a little more complicated as one needs to keep track of the size of each variable and one has to remember that the most significant word is a zero padded quadword. An example can illustrate.

Take the big number

```
0x11112222333344445555666677778888999
```

OpenSSL places this in an array

```
[0] 66667777 88889999
[1] 22223333 44445555
[2] 00000000 00001111
```

The IBM MPM library represents this in a 2 quadword array as follows:

```
[0] 00000000 00000000 00000000 00001111
[1] 22223333 44445555 66667777 88889999
```

6.3 RSA/CRT

We implement traditional RSA Decryption using Chinese Remainder Theorem but with a small modification. Because the SPU is restrictive in some respects we need to maintain a sequence of calls that ensure the results of any modular exponentiation stay positive. We must also be cognisant of the following:

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

Algorithm 6.1 RSA Decryption using Chinese Remainder Theorem modified for the IBM MPM unsigned restrictions.

INPUT: $p, q, I_0, dmq_1, dmp_1, iqmp$
OUTPUT: r_0
1: $r_1 \leftarrow I_0 \bmod q$
2: $m_1 \leftarrow {r_1}^{dmq_1} \bmod q$
3: $r_1 \leftarrow I_0 \bmod p$
4: $r_0 \leftarrow r_1^{dmp_1} \bmod p$
5: $r_0 \leftarrow r_0 - m_1$
6: if $r_0 < 0$ $r_0 \leftarrow r_0 + p$
7: $r_1 \leftarrow r_0 \cdot iqmp$
8: $r_0 \leftarrow r_1 \bmod p$
9: $r_1 \leftarrow r_0 \cdot q$
10: $r_0 \leftarrow r_1 + m_1$

To overcome these restrictions we assume p is always less than q, a condition OpenSSL guarantees. The modified algorithm is outlined in Algorithm 6.1. *Note* We follow OpenSSL notation found in all engine implementations.

The IBM-MPM library offers an alternative modular exponentiation function which uses the Montgomery reduction technique. This is more efficient than the classic product then reduce the result modulo n approach as it keeps the numbers from growing unnecessarily. See ([67]. Chapter 14)

6.4 Results

Table 6.1 lists timings in cycles counts and milliseconds for the main processor intensive 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 6.1) we can expect the SPU to be able to process 14.8 4096-bit decryptions in a second. Interestingly (from Table 6.2) we achieve close to this at 14.1. Obviously there is additional overhead from DMA and the process

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

function	calls	cycle count	cycles	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		
sub-total		187039082			0.05845	
Total *includes other calls*		215159632			0.06724	14.87

Table 6.1: RSA/CRT decryption implemented in IBM MPM function calls with cycle count and time in milliseconds for a 4096-bit key

queue on the main PPU. Cycle counts are from 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 Fedora with kernel version 2.6.27[3]. 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. In reality we observe slightly lower results (Table 6.3). 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. OpenSSL is configured for 64-bit PPC/G5 ASM [4].

From Table 6.3 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 sign/sec vs. 9.1).

To see the full impact of the multi-core we need to use the -multi [n] option to the speed

[3]Linux ps3 2.6.27.9-159.fc10.ppc64

[4]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 key length	PPU		1 SPU	
	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 6.2: OpenSSL speed on PPU vs. 1 SPU using IBM-MPM on 3.2GHz Cell

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 are single threaded. Again we compare the PPU with an SPU enabled engine.

We see from Table 6.3 similar overheads impacting the 1024-bit keys. However there is a huge improvement in 2048-bit (329.7 vs 71.7) and 4096-bit (83.6 vs 9.1).

RSA key length	PPU		6 SPUs	
	sign	sign/sec	sign	sign/sec
1024-bits	0.003435s	291.2	0.001906s	524.7
2048-bits	0.017541s	57.0	0.003033s	329.7
4096-bits	0.109793s	9.1	0.011925s	83.9

Table 6.3: OpenSSL speed on PPU vs. 6 SPUs using IBM-MPM on 3.2GHz Cell, 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 6.4) are consistent with the trend from the Playstation 3 results.

RSA key length	2 PPUs		16 SPUs	
	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 6.4: OpenSSL speed on 2 PPUs vs. 16 SPUs using IBM-MPM on 3.2GHz Cell, 16 parallel processes.

6.5 Conclusions and Future Work

We believe that we have pushed the Cell SDK's IBM-MPM library to its limits. The library is an excellent demonstration of the power of SPU intrinsics 'vector' programming. However, we believe the introduction of an optimised number library more suited to cryptography 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 parameters, however most commonly used 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 so 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 [24] method would be more suitable for the large, fixed sized numbers used by cryptography. Furthermore, as the number length moves beyond 1024-bit the Comba method can be combined with the Karatsuba technique [57] 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 ([84] and Chapter 7, Section 7.1.2) with fixed key sizes on fixed 128-bit alignment, 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 MIRACL library.

The PPU to SPU data transfers are based on the commonly used DMA transfer usually associated with streaming and double buffering, it would be an interesting exercise to preload the SPU with the RSA code at initialisation and then use Mailboxes to manage the

data transfers. This technique may reduce the data on the high bandwidth bus.

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, we would like to test the results on full 16 SPU dual Cell with a commercial grade SSL/HTTP load testing suite.

Commercial, built for purpose, SSL accelerators tend to offer secure key management capabilities. It would be interesting to examine the full design, and performance impact of the full HSM outlined in (Section 5.2.3).

Chapter Acknowledgements

We would also like to acknowledge the valuable feedback given by the anonymous reviewers from the SPEED 2007 (Amsterdam) workshop at which the work described in this chapter was initially presented.

7

Utilising the Cell's SPU for ECC

Background

The publication of the results achieved at the security protocol level (see Chapter 6) on the Cell led us to collaborate with a team looking at Pseudo-random number generators on the Cell for use by super computing packages (such as weather forecast simulators). Our work focused on developing efficient integer arithmetic routines used by a linear congruential generator (LCG). The subsequent publication by IBM of an extensive SDK which allowed third party developers to access the physical random sources on the Cell led to this work being redundant. We then reused the fast 64-bit integer arithmetic routines by merging them into a wider toolkit usable inside a cryptographic library with interesting results.

7.1 Introduction

We focus on a range of optimising techniques for the Cell SPU (SPU intrinsics, pipe-line analysis, branch reduction). We show the significant performance improvements that can be achieved when first optimising the lowest level multiply, then applying these steps to the IBM multi-precision (MPM) library at a fixed, relatively small number size as used by ECC

7.1.1 Modular Components

Alvaro, Kurzak and Dongarra, [59] introduce the idea of a *computation micro-kernel* for the SPU where the restricted code & data size of the SPU become important design criteria but issues such as inter-chip communication and synchronisation are not considered. The kernel focuses on utilisation of the wide registry and the instruction level parallelism. Furthermore, for security-aware applications such as those using ECC, there is an interesting security architecture where an SPU can run in *isolation mode*, where inter-chip communications, loading and unloading program code incur significant overhead. We aim to design a small, fit-for-purpose, micro-kernel suitable for use inside a larger security application, such as a Hardware Security Module (HSM), perhaps as a key negotiation module for an AES stream encryptor kernel inside a Cell SPU.

7.1.2 Multi-precision Tookits

High-level programming languages have limits on the size of the native basic data types. C++, for example, usually has a limit of 64-bits for its largest integer (an unsigned long long). The SPUs 128-bit register is too large for this type, however, it can hold the *result* of the multiplication of the product of a pair of unsigned long longs. To fully utilise the register size and perform the larger multi-precison math, we use 3rd party libraries. These libraries fragment the operations on big numbers into smaller "chunks" which fit the processor's word length. This deconstruction comes with a performance penalty. The larger the chunk, the more efficient the library. For example, the SPU hardware multiplier is just 16-bit by 16-bit. The most efficient method for $a.b$ (64×64-bit) multiply is to break them down into 16-bit *sub-words* to use a standard school book (classical or "grammar-school" [27] Chapter 9 Section 9.1) multiply via the spu_mulo() intrinsic to carry out 4 partial products in parallel.

MIRACL

MIRACL [1] [84] is a portable, light weight, multi-precision library widely used for building cryptographic toolkits. Its key strength is its ability to get near optimal performance from

[1] Mike Scott, my supervisor, is the principle behind the MIRACL library.

any processor and to do so in small, compact code size. It achieves this by allowing a developer to provide substitute routines in assembly for each particular architecture. We use this technique to provide routines utilising SPU intrinsics for 64-bit multiply. We then use this code to produce optimised, unrolled code for 256-bit (SPU or PPU). MIRACL is the easiest multi-precision library to use for this task as it is implemented in standard C, is very extensive, well-documented and allows the developer to choose the level of abstraction from the processor. The level of abstraction (C++, C, assembler) tends to impact on performance.

Listing 7.1: MIRACL source example

```
big mir_r0,mir_r1, mir_p,mir_r1,mir_dmq1,mir_q,mir_m1;
powmod(mip,mir_r1,mir_dmq1,mir_q,mir_m1);
mir_mod(mir_r0,mir_r1, mir_p);
```

IBM MPM

As previously mentioned (Chapter 5 Section 5.1), IBM offer a software development kit [29] containing development tools and code samples including a vector optimised Multi-Precision Math library (IBM MPM) [51]. The library is limited compared to MIRACL, has had some "question marks" over the integrity of the reduction algorithm (see Section 7.5), but is very efficient on the SPU.

Listing 7.2: MPM source example

```
vector unsigned int v_r1[N];
const int mod_exp_window_sz=6;
mpm_mod(v_r1,v_I0,sz_I0,v_q,sz_q);
sz_r1=sz_q;
mpm_mont_mod_exp(v_m1,v_r1,v_dmq1,sz_dmq1, v_q,sz_q,mod_exp_window_sz);
```

QHASM

Dan J. Bernstein developed `qhasm` [8] as a "portable" assembly language to help develop cryptographic functions as close to native CPU assembler as possible. Together with Peter Schwabe, we ported `qhasm` to the Cell's SPU. The results are described in more detail in (Chapter 8). To quote Bernstein "We need languages that are *not* portable in the second sense. Some speed-critical chunks of code are written separately for different CPUs; we need programming tools that don't tie the programmer's hands. It's no problem if the resulting code can't run on more than one CPU." The result is a powerful ability to describe cryptographic primitives, required data types and work close to instruction scheduling. The task is essentially mapping assembly instructions to `qhasm` code and using a `qhasm` language to develop algorithms.

Listing 7.3: `qhasm` source example

```
vec128 a3
vec128 shlw0001
a3 = *(vec128 *) ((binp + 0) & ~15)
int32323232 a3s1 = a3 << shlw0001
```

7.1.3 ECC Hierarchy

In ([45] Chapter 5 Section 5.2.1) Hankerson, Menezes & Vanstone outline a hierarchy of operations in ECC as protocols, point multiplication, elliptic curve addition and doubling and finite field arithmetic. Fan, Sakiyama & Verbauwhede [36] expand this to describe a 5-layer pyramid of

1. Integrity, confidentially, Authentication
2. EC Scalar Multiplication kP
3. Point addition and doubling
4. Modular operations on $\mathbb{F}_p$
5. Instructions of a w-bit core

We implement the stack outlined in the ECC hierarchy using Diffie-Hellman key exchange as an application at the protocol layer and MIRACL for point multiplication, addition & doubling. We then provide SPU optimised routines for modular lower layer operations. We approach this in a various ways and compare the results.

The main issue with symmetric key encryption algorithms, such as AES is the *key distribution problem.* In 1976 a solution was proposed by Diffie-Hellman [31] in the same ground breaking paper they introduced public key cryptography, where the computationally expensive asymmetric encryption is used to swap keys to be subsequently used by the more efficient symmetric algorithms.

7.1.4 Suitable Curve

To ease adoption, and allay fears of weakness for certain curves, the US standards body, NIST, has recommended 15 curves of varying security levels for use by US federal agencies. Most commercial implementations make use of these curves.

The NIST-256 prime $2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$ is unsuitable as 224 is not divisible by 64, so we choose to generate our own elliptic curve over $\mathbb{F}_p$.

As we hope to use the IBM MPM library as a building block for the large number math, we chose to use a curve defined over $\mathbb{F}_p$ where p is 256-bits, a multiple of the SPUs register size. It is more common for embedded or resource constrained devices to use a 192-bit prime curve.

For efficient implementation, we choose a modulus that supports fast reduction and A was fixed at -3. This is a pseudo-mersenne prime of the form $2^n - c$ where c is small. Unfortunately, pseudo-mersenne prime's were patented by Crandall while he worked for Next. Thereafter, they passed to Apple, who have never enforced them. When it came to standardisation the standards body chose not to use a patented method and use Generalised Mersenne prime of the form $2^n \pm -2^m + ..1$, where n, m, etc are all divisible by the word length of the computer. This is not as satisfactory for efficient reduction, but it is unpatented.

We also need the number of points on the curve (the group order) q, to be prime. We are looking for p to be 256-bits. We first locate c so that $2^{256} - c$ where c is small. We find

that $2^{256} - 189$ [2] is prime. To locate a suitable curve with this c we use tools included with MIRACL library [84] which implement some improvements on the Schoof [83] algorithm for counting points on a $\mathbb{F}_p$) elliptic curve. The *Schoof-Elkies-Atkin* algorithm finds a number of suitable curves in approximately one hour on a standard PC. However, we add the additional constraint the $q > p$ so as to fit inside our 256-bit optimised routines. This extends the search to approximately one day. These curve-finding utilities require being run just once and can be reused.

The first suitable curve, in short form Weierstraß, that fits our extended criteria is:

$$y^2 = x^3 - 3x + 1403 \mod 2^{256} - 189 \tag{7.1}$$

7.2 Multiply bottleneck

At the lowest level, the instructions of a w-bit core, the bottleneck for performance is the frequency with which it must perform multiply/add (also known as *MADD()*) operations. That is $d = ((a \times b + c) \mod (2^{128}))$.

As mentioned before, the large register size is ideal for the number-crunching operations required for *MADD()*. However, most operations work with, at most, 32-bit sub-sections of the quadword register, the integer multiply operations being especially limiting, reducing to a 16-bit hardware multiplier. This sub-quadword size is suitable for SIMD operations on media-like data streams like those expected in a PS3 multimedia application. For larger size integer operations it takes a slight re-factoring to get the most efficient throughput. This is how standard multi-precision number manipulation is done in all cryptography libraries, however, in this case, we can take advantage of the parallel nature of the SIMD.

7.2.1 Behind a 64-bit Multiply

In C/C++ for example, using a `MADD()` with the largest type causes overflow, with the most significant half of the bits being discarded causing a subsequent loss in integrity. There are numerous techniques to overcome this issue when using the basic C/C++ types but most

[2] $2^{256} - 189 = 115792089237316195423570985008687907853269984665640564039457584007913129639747$

are relatively inefficient.

We present an efficient 64×64-bit multiplier with full integrity, running at nearly twice the speed of the most commonly used alternative. This new multiplier still uses the limiting 16-bit hardware multiplier in the SPU but takes full advantage of the Cell SPU's large register file.

As the basic multiplier is 16-bit by 16-bit, the most efficient method for $a \times b$ (64×64-bit) multiply is to break it down into 16-bit *(*sub-words) arranged as such a_3, a_2, a_1, a_0. For $a \times b$ is broken down to 4×16-bit values using the standard school book multiply described above and outlined in the documentation of MIRACL [84].

		a_3	a_2	a_1	a_0
	x	b_3	b_2	b_1	b_0

			$a_3.b_0$	$a_2.b_0$	$a_1.b_0$	$a_0.b_0$
		$a_3.b_1$	$a_2.b_1$	$a_1.b_1$	$a_0.b_1$	
	$a_3.b_2$	$a_2.b_2$	$a_1.b_2$	$a_0.b_2$		
$a_3.b_3$	$a_2.b_3$	$a_1.b_3$	$a_0.b_3$			

The first line ($a_3.b_0$ $a_2.b_0$ $a_1.b_0$ $a_0.b_0$) can be calculated by re-arranging (shuffling) a in the 128-bit register as $(0|a_3|0|a_2|0|a_1|0|a_0)$ and multiplying this by $(0|b_0|0|b_0|0|b_0|0|b_0)$

	0	a_3	0	a_2	0	a_1	0	a_0
x	0	b_0	0	b_0	0	b_0	0	b_0

$a_3.b_0$	$a_2.b_0$	$a_1.b_0$	$a_0.b_0$

This results in $(a_3.b_0|a_2.b_0|a_1.b_0|a_0.b_0)$, 4 32-bit partial products, which is the first line, as required. Next, we calculate and store in registers the next 3 lines in exactly the same

way. We then add the lines and propagate the carries.

Each line can be considered as ($3H|3L|2H|2L|1H|1L|0H|0L$) (H=high, L=low) For the first line, shuffle this into two registers as ($0|0|0|0|3L|2L|1L|0L$) and ($0|0|0|3H|2H|1H|0H|0$)

The sum of these two registers and the 3 other pairs of registers produce the final result (shifting over by one place each time)

	00	00	00	00	$3L$	$2L$	$1L$	$0L$
+	00	00	00	$3H$	$2H$	$1H$	$0H$	00
+	00	00	00	$3L$	$2L$	$1L$	$0L$	00
+	00	00	$3H$	$2H$	$1H$	$0H$	00	00
+	00	00	$3L$	$2L$	$1L$	$0L$	00	00
+	00	$3H$	$2H$	$1H$	$0H$	00	00	00
+	00	$3L$	$2L$	$1L$	$0L$	00	00	00
+	$3H$	$2H$	$1H$	$0H$	00	00	00	00

The optimised SPU intrinsics code for is this can be viewed in listing 3 in Appendix B.

7.3 Implementation

7.3.1 ECC Performance Bottleneck

The performance bottleneck we focus on is the elliptic curve scalar multiplication. This is implemented using a sequence of point addition and point doubling operations. These in turn are implemented via a sequence of modular multiplication, squaring, addition and subtractions. It is these 'low layer' routines that we implement as multi-precision routines using the SPU intrinsics to get maximum performance from the SPU core.

7.3.2 Approach

Since our goal is to speed up a standard PPU implementation, we first implement a fast, C based, 64-bit multiply routine to act as our benchmark. Subsequently, we optimise scalar multiplication at the w-core level by implementing the 64-bit multiply using SPU intrinsics.

Then, we move up a layer and use modular arithmetic routines provided by the MPM library. Finally, we remove some inefficiency inside MPM by unrolling loops & removing branches with MPM derived code for fixed sized 256-bit modular routines. We take a number of standard approaches, each improving on the last. Separately, we built a test script using bash and the GNU bc calculator which could test the integrity at each stage.

64-bit Multiply in C

IBM have published an efficient 64-bit multiply implementation utilising [57], suitable for the PowerPC family of processors. See [17] for details and listing.

compiler	cycles	instructions	CPI	M / sec
spu-gcc	223	170 (153)	1.31 (1.46)	14.314

Table 7.1: 64-bit multiply in C. Million Multiplies per second

64-bit Multiply using SPU intrinsics in an inline function

In order to help the compiler achieve optimal code, we apply the SDK cycle accurate simulator and additional code analysis tools. For example, as the SPU pipelines are not equal, with operations using a pipe based on the *instruction class* we examine the SPU intrinsics code using the SPU_timing tool and a pipe depth/dependency table. All integer operations use an even pipe while the load/stores use the odd pipeline. Carefully interleaving of the instructions improves the Cycles Per Instruction (CPI) ratio. Note there are two pipelines, but as as the pipelines are not equal, it is difficult to achieve an optimal 0.5 CPI. The SDK documentation cites a more reasonable goal of 0.7.

The SPU is interesting in that while it can do one clock cycle parallel operation, it is relatively inefficient for load / store operations. Branches, in particular, incur long stall delays. Optimal code limits the amount of movement in and out of registers, and attempts to flow without *ifs, loops, and function calls*. To overcome these overheads we use the C pre-processor to our advantage by:

1. Declaring the vector *splat* patterns as constant globals.

2. Loading the 64-bit multipliers into two half's of a 128-bit vector saving a load().

3. Manually in-lining the full multiply code in a C header file via macros to save a function call branch, effectively in-lining the C code.

4. Removing loops (unrolling) by manually expanding the vector access rather than processing via traditional arrays.

See Appendix B listing for code sample. If we implement the school book algorithm in the most basic fashion where we naively carry out each step of the algorithm but take advantage of the SIMD we get nearly double throughput.

compiler	cycles	instructions	CPI	M / sec
spu-gcc	107	108 (102)	0.99 (1.05)	29.831

Table 7.2: SPU intrinsics. Million Multiplies per second

64-bit Multiply in SPU intrinsics. C Macro

Taking the code one step further, aware that the Cell SPU stalls badly if it incurs branches, we re-code this as a C macro, moving some constants out and eliminating the function call branch and we achieve a further gain of about 20%. See Appendix B listing for code sample.

complier	cycles	instructions	CPI	M / sec
spu-gcc	89	96 (92)	0.93 (097)	35.865

Table 7.3: C Macro. Million Multiplies per second

7.3.3 Automatic Code Generation

For future reuse, rather than fixing our library at 256-bit, and following on from the experience of hand unrolling the MPM multiply routines, we developed an automated method based on code-to-code compiling/code generation techniques for MIRACL.

We used the multiple-precision multiplication algorithm from [67] (Chapter 14 of Section 14.2.3) but we automated the code unrolling to limit the chance of errors which could

otherwise be introduced by manual methods such as cut and paste, i.e: hard to debug errors introduced by the excessive code unrolling and index de-referencing. We implemented a Ruby script to parse C code and do inline substitutions replacing C defines (e.g such as MADD(N,a,b,c)) with fully unrolled code to calculate N-bit $a \times b + c$. This $a \times b + c$ is based around an optimal 64-bit multiply function created by examining the instruction dependencies using the SPU_timing tool from the SDK where we minimised the number of loadstore operations and removed all loops and arrays (and array de-referencing).

See B for the Ruby code, a generated 256-bit example, and the fully optimised 64-bit multiplier.

7.3.4 Using MPM

256-bit mod multiply using MPM

The MPM library provides a number of options such as an alternative modular exponentiation function which uses the Montgomery reduction technique. This is more efficient than the classic multiply then reduce modulo N approach as it keeps the numbers from growing unnecessarily. See [67] (Chapter 14). Note that by adding the MPM library to the test harness we now have the issue of switching big number formats between those used natively by MIRACL and MPM.

256-bit mod multiply using optimised MPM

MPM doesn't provide a square() function which is used by the ECC point doubling routines. Significant optimisations can be made over a multiply if the operands are equal as one can reuse SIMD partial products saving expensive multiply calls.

The MPM library is provided in source code and is generic for arbitrary size numbers. But as we know we need 256-bit (and some 512-bit) numbers, we can help the compiler by providing some domain knowledge and modify the library by editing the source to manually unroll loops and removing branches.

The MPM multiply function, at a pseudo-high level, follows a pattern of (Algorithm 7.1)

Algorithm 7.1 MPM Multiply function

INPUT: $A[N], B[N]$
OUTPUT: $Result[2 * N]$

Variables Declaration as array $[0..N]$
Initialise Variables [0 to N]
for Outer Loop 0 to N **do**
  for Inner Loop to 0 to N **do**
  end for
end for
Gather results [0 to N]

For 256-bit ECC we can unroll this to a pattern (Algorithm 7.2)

Algorithm 7.2 MPM Multiply function 256-bit unrolled

INPUT: $A[N], B[N]$
OUTPUT: $Result[2 * N]$

$Variables$ 1 2 3 4
$Initialise$ 1 2 3 4
$Outer1$
 $Inner1$
 $Inner2$
$Outer2$
 $Inner1$
 $Inner2$
$Gather$ 1 2 3 4

Using explicit variable naming as oppose to implicit array calls and fully unrolling the loop helps both the GCC and IBM XLC compilers.

7.3.5 Results

Table 7.4 giving Performance of the ecurve_mult()[3]

7.4 Branch Prediction

The SPU suffers a high penalty (18 cycles) for misdirected branching. To reduce penalties, the SPU addresses branch prediction through a set of hint for branch (HBR) instructions

[3]Tests are run on a 3.2 GHz Playstation 3 with just **6** SPUs running Fedora 7 [77] with kernel version 2.6.21-1.3194.fc7.

	ticks	pclocks	micro secs
64-bit C	865318	34612720	10843.58
SPU 64-bit function()	823517	32940680	10319.76
SPU 64-bit define	788304	31532160	9878.50
SPU 256-bit unrolled	788304	31532160	9878.50
Standard MPM	618037	24721480	7744.82
Optimised 256-bit MPM	602956	24118240	7555.84

Table 7.4: Results

that facilitate efficient branch processing. At a higher level, the C/C++ Language Extensions for the Cell provide a __builtin_expect directive to allow programmers to predict conditional program statements. For example (Listing 7.4) predicts the a is not larger than b.

Listing 7.4: Simple branch prediction

```
if(__builtin_expect((a>b),0))
    c += a;
else
    d += 1;
```

We implemented the prediction inside the code for our special-form moduli $(2^{256} - 189)$ function where we can branch-predict the small, very unlikely, last check. See the Montgomery-Multiplication-with-Reduction algorithm (See [27] Algorithm 9.2.13 and Chapter 2 Section C.0.4) (Fast mod operation for special/form moduli) where there is a simple final check for overflow (if $x >= N$ then $x = x - N$).

There is a problem with resolution of the run time timers on the PS3 (about 20 cycles) versus the slow performance of the large number routines inside the cycle accurate simulator, where the host environment is very slow to run the high computations required. This makes it complex to get an accurate picture of the success of small improvements.

We observed that branch is taken roughly 1% of the time based on a few million random inputs to the branch hint code and the prediction code just gives a relatively small, (>1%) improvement.

However the conditional subtract at the end of the algorithm also represents a side channel weakness (See [23] Section 10.4.3). This small improvement is not worth the security issue.

Listing 7.5: MIRACL code for (x reduced modulo $2^{256} - 189$) with branch prediction

```
void nc_mod256(_MIPD_ big x,big y)
{
    int i;
    big A,B;
    miracl *mr_mip=get_mip();
    char mem_big[MR_BIG_RESERVE(2)];      /* we need 2 bigs... */
    memset(mem_big,0,MR_BIG_RESERVE(2)); /* clear the memory  */

    A=mirvar_mem(mr_mip, mem_big, 0);        /* Initialise big numbers */
    copy(x,mr_mip->w0);
    B=mirvar_mem(mr_mip, mem_big, 1);

    if(__builtin_expect((mr_mip->w0->len>4),1))   //
if(mr_mip->w0->len>4)
    {
        zero(A);
        zero(B);
        A->len=B->len=4;
        for (i=0;i<4;i++)
        {
            A->w[i]=mr_mip->w0->w[i];
            B->w[i]=mr_mip->w0->w[4+i];
        }
        mr_lzero(A);
        mr_lzero(B);
          /* A is bottom half */ /* B is top half     */
        premult(_MIPP_ B,0xBD,B);
        add(_MIPP_ A,B,mr_mip->w0);
```

```
    }

    if(__builtin_expect((mr_mip->w0->len>4),1))    //
if(mr_mip->w0->len>4)
    {
        zero(A);
        A->len=1;
        A->w[0]=mr_mip->w0->w[4];
        mr_lzero(A);
        zero(B);
        B->len=4;
        for (i=0;i<4;i++)
        {
            B->w[i]=mr_mip->w0->w[i];
        }
        premult(_MIPP_ A,0xBD,A);
        add(_MIPP_ A,B,mr_mip->w0);
    }

   if(__builtin_expect((compare(mr_mip->w0,mr_mip->modulus)==1),0)) //
if (compare(mr_mip->w0,mr_mip->modulus)==1)
   {
            subtract(_MIPP_ mr_mip->w0,mr_mip->modulus,mr_mip->w0);
   }

    copy(mr_mip->w0,y);
    memset(mem_big,0,MR_BIG_RESERVE(2));
}
```

7.5 Future Work

Recently, discussions on the IBM Cell developer forum have cast some doubt over the integrity of the MPM mod() functions. Researchers working at École Polytechnique Fédérale de Lausanne (EPFL) have found some cases where the reduction functions provided incorrect results. The replacements offered by IBM appear to work; we have verified them empirically. However, they do not seem, by examining source code, to be as efficient as the original code. We intend to further examine the MPM source and swap in our own MPM compatible routines.

We are still *thunking* formats between the upper layer (ECC) MIRACL code's and the MPM's big number representation. Our observations, at 256-bit ECC, is that this is a 25% overhead. Further work would be to attempt tying the large number representation inside the ECC point multiply and addition algorithms in one format.

8

Fast Elliptic-Curve Cryptography on the Cell Broadband Engine

Background

Our work on ECC on the Cell (Chapter 7) introduced us to Dan Bernstein and the team from Technische Universiteit, Eindhoven (TU/e). Bernstein has been championing an elliptic curve (curve25519) which has slightly less security level than standard 256-bit curves but is very suitable for reduced representation implementations. These reduced representations would, for example, be able to do addition without any processing of a carry. Our work with Peter Scwabe exploited these efficiencies on the Cells SPU and, for a time at least, achieved speed records for ECC.

8.1 Introduction

In this chapter we present a high-speed implementation of elliptic-curve Diffie-Hellman (ECDH) key exchange for the Cell, which needs 697080 cycles on one SPU for a scalar multiplication on a 255-bit elliptic curve, including the costs for key verification and key compression. This cycle count is independent of inputs therefore protecting against timing attacks.

This speed relies on a new representation of elements of the underlying finite field suited for the unconventional instruction set of this architecture.

We also demonstrate that an implementation based on the multi-precision integer arithmetic functions provided by IBM's multi-precision math (MPM) library (Chapter 7 Section 7.1.2) and [51]) would take at least 2227040 cycles.

Comparison with implementations of the same function for other architectures shows that the Cell is competitive in terms of cost-performance ratio to other recent processors such as the Intel Core 2 for public-key cryptography.

Specifically, the state-of-the-art Galbraith-Lin-Scott ECDH software performs 27370 scalar multiplications per second when using all four cores of a 2.5GHz Intel Core 2 Quad Q9300 inside a $296 computer [1], while the new software reported in this Chapter performs 27474 scalar multiplications per second on a Playstation 3 that costs just $221. Both of these speed reports are for high-security 256-bit elliptic-curve cryptography.

This chapter describes a high-speed implementation of state-of-the-art public-key cryptography for the Cell Broadband Engine (CBE). More specifically we describe an implementation of the `curve25519` function, an elliptic-curve Diffie-Hellman key exchange (ECDH) function introduced in [9].

Implementations of this function have been achieving speed records for high-security ECDH software on different platforms for example [9] and [40]. Benchmarks of our implementation show that the CBE is competitive (in terms of cost-performance ratio) with respect to other recent processors such as the Intel Core 2 for public-key cryptography.

Our implementation needs 697080 cycles on one SPU. This includes not only scalar multiplication on the underlying 255-bit elliptic curve, but also costs for key compression, key validation and protection against timing attacks. We put our implementation into the public domain to maximize the impact of our research. It is available as part of the SUPERCOP benchmarking suite [10] and at `http://cryptojedi.org/crypto/index.shtml#celldh`.

[1] prices quoted use an internet search for cheapest source using google product search April 09

8.1.1 How these speeds were achieved

As described in (Chapter 7 Section 7.1.3) elliptic-curve cryptography (ECC) is usually implemented as a sequence of arithmetic operations in a finite field. Chapter 7 described the obvious approach for the implementation of ECC on the CBE is using the IBM-MPM or MIRACL libraries for the underlying finite field arithmetic.

However, we will show that the targeted performance cannot be achieved following this approach, not even with optimizing some functions of the MPM library for arithmetic in fields of the desired size.

Instead, the speed of our implementation is achieved by

- Parting with the traditional way of implementing elliptic-curve cryptography which uses arithmetic operations in the underlying field as smallest building blocks,
- Representing finite field elements in a way that takes into account the special structure of the finite field and the unconventional SPU instruction set, and
- Careful optimization of the code at assembly level.

Related work Implementations of public-key cryptography for the Cell Broadband Engine have not yet been extensively studied. In particular we don't know of any previous implementation of ECC for the Cell Broadband Engine.

An implementation of the Digital Signature Algorithm (DSA) supporting key lengths up to 1024 bits is included in the SPE Cryptographic Library [52].

In [89] Shimizu et al. report 4074000 cycles for 1024-bit-RSA encryption or decryption and 1331000 cycles for 1024-bit-DSA key generation. Furthermore they report 2250000 cycles for 1024-bit-DSA signature generation and 4375000 cycles for 1024-bit-DSA signature verification.

The Cell Broadband Engine has recently demonstrated its power for cryptanalysis of symmetric cryptographic primitives [93], [92].

Organization of the Chapter Section 8.2 describes the `curve25519` function including some necessary background on elliptic-curve arithmetic. Chapter 7 describes IBM's MPM

library including optimizations we applied to accelerate arithmetic in finite fields of the desired size. We show that an implementation based on this library cannot achieve the targeted performance. In Section 8.4 we detail our implementation of `curve25519`. We conclude the Chapter with a discussion of benchmarking results and a comparison to ECDH implementations for other architectures in Section 8.5.

8.2 The `curve25519` function

8.2.1 The `curve25519` function

Bernstein proposed in [9] the `curve25519` function for elliptic-curve Diffie-Hellman key exchange. This function uses arithmetic on the elliptic curve defined by the equation $E : y^2 = x^3 + Ax^2 + x$ over the field $\mathbb{F}_p$, where $p = 2^{255} - 19$ and $A = 486662$; observe that this elliptic curve allows for the x-coordinate-based scalar multiplication described above.

The elliptic curve and underlying finite field are carefully chosen to meet high security requirements and to allow for fast implementation, For a detailed discussion of the security properties of `curve25519` see [9].

The `curve25519` function takes as input two 32-byte strings, one representing the x-coordinate of a point P and the other representing a 256-bit scalar k. It gives as output a 32-byte string representing the x-coordinate x_Q of $Q = [k]P$. For each of these values `curve25519` is assuming little-endian representation.

For our implementation we decided to follow [9] and compute x_Q by first using Algorithm 2.1 to compute (X_Q, Z_Q) and then computing $x_Q = Z_Q^{-1} \cdot X_Q$.

8.3 The MPM library and ECC

8.3.1 $\mathbb{F}_p$ arithmetic using the MPM library

In Section 8.2 we described how the upper 3 layers of this hierarchy are handled. Hence, the obvious next step is to look at efficient modular operations in $\mathbb{F}_p$ and how these operations can be mapped to the SIMD instructions on 128-bit registers of the SPU.

This task of mapping operations on large integers to the SPU instruction set, is handled by the vector-optimized multi-precision math (MPM) library [51].

This MPM library is provided in source code and its algorithms are generic for arbitrary sized numbers. They operate on 16-bit halfwords as smallest units, elements of our 255-bit field are therefore actually handled as 256-bit values.

As our computation is mostly bottlenecked by costs for multiplications and squarings in the finite field we decided to optimize these functions for 256-bit input values.

The original MPM multiplication functions were optimsized as outlined in (Chapter 7 Section 7.3.4)

While these manual unroll and branch hint techniques help both the GCC and IBM XLC compilers it should be noted that, for this unrolled MPM code, the GCC-derived compiler achieves a 10% improvement over the XLC compiler[2].

The MPM library supplies a specialized function for squaring where significant optimizations should be made over a general multiply by reusing partial products. However our timings indicate that such savings are not achieved until the size of the multi-precision inputs exceeds 512-bits. We therefore take the timings of a multiplication for a squaring.

8.3.2 What speed can we achieve using MPM?

The Montgomery ladder in the `curve25519` computation consists of 255 ladder steps, hence, computation takes 1276 multiplications, 1020 squarings, 255 multiplications with a constant, 2040 additions and one inversion in the finite field $\mathbb{F}_{2^{255}-19}$. Table 8.1 gives the number of CPU cycles required for each of these operations (except inversion).

For finite field multiplication and squaring we benchmarked two possibilities: a call to `mpm_mul` followed by a call to `mpm_mod` and the Montgomery multiplication function `mpm_mont_mod_mul`. Addition is implemented as a call to `mpm_add` and a conditional call (`mpm_cmpge`) to `mpm_sub`. For multiplication we include timings of the original MPM functions and of our optimized versions. The original MPM library offers a number of options for each operation. We select the inlined option with equal input sizes for fair comparison.

[2]IBM XL C/C++ for Multicore Acceleration for Linux, V10.1. CBE SDK 3.1

Operation	Number of cycles
Addition/Subtraction	86
Multiplication (original MPM)	4334
Multiplication (optimized)	4124
Montgomery Multiplication (original MPM)	1197
Montgomery Multiplication (optimized)	892

Table 8.1: MPM performance for arithmetic operations in a 256-bit finite field

From these numbers we can compute a lower bound of 2227040 cycles (1276M + 1020S + 2040A, where M, S and A stand for the costs of multiplication, squaring and addition respectively) required for the `curve25519` computation when using MPM. Observe that this lower bound still ignores costs for the inversion and for multiplication with the constant.

The high cost for modular reduction in these algorithms results from the fact that the MPM library cannot make use of the special form of the modulus $2^{255} - 19$; an improved specialized reduction routine would probably yield a smaller lower bound. We therefore investigate what lower bound we get when entirely ignoring costs for modular reduction. Table 8.2 gives numbers of cycles for multiplication and addition of 256-bit integers without modular reduction. This yields a lower bound of 934080 cycles. Any real implementation would, of course, take significantly more time as it would need to account for operations not considered in this estimation.

Operation	Number of cycles
Addition/Subtraction	52
Multiplication (original MPM)	594
Multiplication (optimized)	360

Table 8.2: MPM performance for arithmetic operations on 256-bit integers

8.4 Implementation of `curve25519`

As described in Section 8.2 the computation of the `curve25519` function consists of two parts, the Montgomery ladder computing (X_Q, Z_Q) and the inversion of Z_Q.

We decided to implement the inversion as an exponentiation with $p-2 = 2^{255}-21$ using

the same sequence of 254 squarings and 11 multiplications as [9]. This might not be the most efficient algorithm for inversion, but it is the easiest way to implement an inversion algorithm which takes constant time.

The addition chain is specialized for the particular exponent and cannot be implemented as a simple square-and-multiply loop; completely inlining all multiplications and squarings would result in an excessive increase of the overall code size. We therefore implement multiplication and squaring functions and use calls to these functions.

However for the first part, the Montgomery ladder, we do not use calls to these functions but take one ladder step as smallest building block and implement the complete Montgomery ladder in one function. This allows for a higher degree of data-level parallelism, especially in the modular reductions, and thus yields a significantly increased performance.

For the speed-critical parts of our implementation we use the `qhasm` programming language ([8] and Chapter 7 Section 7.1.2), which offers us all flexibility for code optimization at assembly level, while still supporting a more convenient development environment than plain assembly. We extended this language to also support the SPU of the Cell Broadband Engine as target architecture.

In the description of our implementation we will use the term ‘register variable‘. Note that for `qhasm` (unlike C) the term register variable refers to variables that are forced to be kept in registers.

8.4.1 Fast arithmetic

In the following section we will first describe how we represent elements of the finite field $\mathbb{F}_{2^{255}-19}$ and then detail the three algorithms that influence execution speed of curve25519 most, namely finite field multiplications, finite field squaring and a Montgomery ladder step.

8.4.2 Representing elements of $\mathbb{F}_{2^{255}-19}$

We represent an element a of $\mathbb{F}_{2^{255}-19}$ as a tuple $(a_0, \ldots, a_{19})$ where

$$a = \sum_{i=0}^{19} a_i 2^{\lceil 12.75i \rceil}. \tag{8.1}$$

We call a coefficient a_i *reduced* if $a_i \in [0, 2^{13} - 1]$. Analogously we call the representation of an element $a \in \mathbb{F}_{2^{255}-19}$ reduced if all its coefficients $a_0, \ldots, a_{19}$ are reduced.

As described in section (Chapter 5 Section 5.2) the Cell Broadband Engine can only perform 16-bit integer multiplication, where one instruction performs 4 such multiplications in parallel. In order to achieve high performance of finite field arithmetic it is crucial to properly arrange the values $a_0, \ldots a_{19}$ in registers and to adapt algorithms for field arithmetic to make use of this SIMD capability.

Multiplication and Squaring in $\mathbb{F}_{2^{255}-19}$

As input to field multiplication we get two finite field elements $(a_0, \ldots, a_{19})$ and $(b_0, \ldots, b_{19})$. We assume that these field elements are in reduced representation. This input is arranged in 10 register variables `a03`, `a47`, `a811`, `a1215`, `a1619`, `b03`, `b47`, `b811`, `b1215` and `b1619` as follows: Register variable `a03` contains in its word elements the coefficients a_0, a_1, a_2, a_3, register variable `a47` contains in its word elements the coefficients a_4, a_5, a_6, a_7, and so on.

The idea of multiplication is to compute coefficients $r_0, \ldots, r_{38}$ of $r = ab$ where:

$$
\begin{aligned}
r_0 &= a_0b_0 \\
r_1 &= a_1b_0 + a_0b_1 \\
r_2 &= a_2b_0 + a_1b_1 + a_0b_2 \\
r_3 &= a_3b_0 + a_2b_1 + a_1b_2 + a_0b_3 \\
r_4 &= a_4b_0 + 2a_3b_1 + 2a_2b_2 + 2a_1b_3 + a_0b_4 \\
r_5 &= a_5b_0 + a_4b_1 + 2a_3b_2 + 2a_2b_3 + a_1b_4 + a_0b_5 \\
r_6 &= a_6b_0 + a_5b_1 + a_4b_2 + 2a_3b_3 + a_2b_4 + a_1b_5 + a_0b_6 \\
r_7 &= a_7b_0 + a_6b_1 + a_5b_2 + a_4b_3 + a_3b_4 + a_2b_5 + a_1b_6 + a_0b_7 \\
r_8 &= a_8b_0 + 2a_7b_1 + 2a_6b_2 + 2a_5b_3 + a_4b_4 + 2a_3b_5 + 2a_2b_6 + 2a_1b_7 + a_0b_8 \\
&\vdots
\end{aligned}
$$

This computation requires 400 multiplications and 361 additions. Making use of the

SIMD instructions, at best 4 of these multiplications can be done in parallel, adding the result of a multiplication is at best for free using the `mpya` instruction, so we need at least 100 instructions to compute the coefficients $r_0, \ldots, r_{38}$. Furthermore we need to multiply some intermediate products by 2, an effect resulting from the non-integer radix 12.75 used for the representation of finite field elements. As we assume the inputs to have reduced coefficients, all result coefficients r_i fit into 32-bit word elements.

We will now describe how the coefficients $r0, \ldots, r38$ can be computed using 145 pipeline-0 instructions (arithmetic instructions). This computation requires some rearrangement of coefficients in registers using the `shufb` instruction but with careful instruction scheduling and alignment these pipeline-1 instructions do not increase the number of cycles needed for multiplication. From the description of the arithmetic instructions it should be clear which rearrangement of inputs is necessary.

First use 15 `shl` instructions to have register variables

`b03s1` containing $b_0, b_1, b_2, 2b_3$,

`b03s2` containing $b_0, b_1, 2b_2, 2b_3$,

`b03s3` containing $b_0, 2b_1, 2b_2, 2b_3$,

`b47s1` containing $b_4, b_5, b_6, 2b_7$ and so on.

Now we can proceed producing intermediate result variables

`r03` containing $a_0b_0, a_0b_1, a_0b_2, a_0b_3$ (one `mpy` instruction),

`r14` containing $a_1b_0, a_1b_1, a_1b_2, 2a_1b_3$ (one `mpy` instruction),

`r25` containing $a_2b_0, a_2b_1, 2a_2b_2, 2a_2b_3$ (one `mpy` instruction),

`r36` containing $a_3b_0, 2a_3b_1, 2a_3b_2, 2a_3b_3$ (one `mpy` instruction),

`r47` containing $a_4b_0 + a_0b_4, a_4b_1 + a_0b_5, a_4b_2 + a_0b_6, a_4b_3 + a_0b_7$ (one `mpy` and one `mpya` instruction),

`r58` containing $a_5b_0 + a_1b_4, a_5b_1 + a_1b_5, a_5b_2 + a_1b_6, 2a_5b_3 + 2a_1b_7$ (one `mpy` and one `mpya` instruction) and so on. In total these computations need 36 `mpy` and 64 `mpya` instructions.

As a final step these intermediate results have to be joined to produce the coefficients $r_0, \ldots r_{38}$ in the register variables `r03`, `r47`,... `r3639`. We can do this using 30 additions if we first combine intermediate results using the `shufb` instruction. For example we join

in one register variable the highest word of `r14` and the three lowest words of `r58` before adding this register variable to `r47`.

The basic idea for squaring is the same as for multiplication. We can make squaring slightly more efficient by exploiting the fact that some intermediate results are equal.

For a squaring of a value a given in reduced representation $(a_0, \ldots, a_{19})$, formulas for the result coefficients $r_0, \ldots, r_{38}$ are the following:

$$
\begin{aligned}
r_0 &= a_0 a_0 \\
r_1 &= 2a_1 a_0 \\
r_2 &= 2a_2 a_0 + a_1 a_1 \\
r_3 &= 2a_3 a_0 + 2a_2 a_1 \\
r_4 &= 2a_4 a_0 + 4a_3 a_1 + 2a_2 a_2 \\
r_5 &= 2a_5 a_0 + 2a_4 a_1 + 4a_3 a_2 \\
r_6 &= 2a_6 a_0 + 2a_5 a_1 + 2a_4 a_2 + 2a_3 a_3 \\
r_7 &= 2a_7 a_0 + 2a_6 a_1 + 2a_5 a_2 + 2a_4 a_3 \\
r_8 &= 2a_8 a_0 + 4a_7 a_1 + 4a_6 a_2 + 4a_5 a_3 + a_4 a_4 \\
&\vdots
\end{aligned}
$$

The main part of the computation only requires 60 multiplications (24 `mpya` and 36 `mpy` instructions). However, some partial results have to be multiplied by 4; this requires more preprocessing of the inputs, we end up using 35 instead of 15 `shl` instructions before entering the main block of multiplications. Squaring is therefore only 20 cycles faster than multiplication.

During both multiplication and squaring, we can overcome latencies by interleaving independent instructions.

8.4.3 Reduction

The task of the reduction step is to compute from the coefficients $r_0, \ldots r_{38}$ a reduced representation $(r_0, \ldots, r_{19})$. Implementing this computation efficiently is challenging in two ways: In a typical reduction chain every instruction is dependent on the result of the preceding instruction. This makes it very hard to vectorize operations in SIMD instructions and to minimise latencies.

We will now describe a way to handle reduction hiding most instruction latencies but without data level parallelism through SIMD instructions.

The basic idea of reduction is to first reduce the coefficients r_{20} to r_{38} (producing a coefficient r_{39}), then add $19r_{20}$ to r_0, $19r_{21}$ to r_1 and so on until adding $19r_{39}$ to r_{19} and then reduce the coefficients r_0 to r_{19}.

Multiplications by 19 result from the fact that the coefficient a_{20} stands for $a_{20} \cdot 2^{255}$ (see equation (8.1)). By the definition of the finite field $\mathbb{F}_{2^{255}-19}$, $2^{255}a_{20}$ is the same as $19a_{20}$. Equivalent statements hold for the coefficients $a_{21}, \ldots, a_{39}$.

The most time consuming parts of this reduction are the two carry chains from r_{20} to r_{39} and from r_0 to r_{19}. In order to overcome latencies in these chains we break each of them into four parallel carry chains, Algorithm 8.1 describes this structure of our modular reduction algorithm.

Each of the carry operations in Algorithm 8.1 can be done using one `shufb`, one `rotmi` and one `a` instruction. Furthermore we need 8 masking instructions (bitwise `and`) for each of the two carry chains.

In total, a call to the multiplication function (including reduction) takes 444 cycles, a call to the squaring function takes 424 cycles. This includes 144 cycles for multiplication (124 cycles for squaring), 244 cycles for reduction and some more cycles to load input and store output. Furthermore the cost of a function call is included in these numbers.

Montgomery ladder step

For the implementation of a Montgomery ladder step we exploit the fact that we can optimize a fixed sequence of arithmetic instructions in $\mathbb{F}_{2^{255}-19}$ instead of single instructions.

Algorithm 8.1 Structure of the modular reduction

Carry from r_{20} to r_{21}, from r_{24} to r_{25}, from r_{28} to r_{29} and from r_{32} to r_{33}
Carry from r_{21} to r_{22}, from r_{25} to r_{26}, from r_{29} to r_{30} and from r_{33} to r_{34}
Carry from r_{22} to r_{23}, from r_{26} to r_{27}, from r_{30} to r_{31} and from r_{34} to r_{35}
Carry from r_{23} to r_{24}, from r_{27} to r_{28}, from r_{31} to r_{32} and from r_{35} to r_{36}

Carry from r_{24} to r_{25}, from r_{28} to r_{29}, from r_{32} to r_{33} and from r_{36} to r_{37}
Carry from r_{25} to r_{26}, from r_{29} to r_{30}, from r_{33} to r_{34} and from r_{37} to r_{38}
Carry from r_{26} to r_{27}, from r_{30} to r_{31}, from r_{34} to r_{35} and from r_{38} to r_{39}
Carry from r_{27} to r_{28}, from r_{31} to r_{32} and from r_{35} to r_{36}

Add $19r_{20}$ to r_0, add $19r_{21}$ to r_1, add $19r_{22}$ to r_2 and add $19r_{23}$ to r_3
Add $19r_{24}$ to r_4, add $19r_{25}$ to r_5, add $19r_{26}$ to r_6 and add $19r_{27}$ to r_7
Add $19r_{28}$ to r_8, add $19r_{29}$ to r_9, add $19r_{30}$ to r_{10} and add $19r_{31}$ to r_{11}
Add $19r_{32}$ to r_{12}, add $19r_{33}$ to r_{13}, add $19r_{34}$ to r_{14} and add $19r_{35}$ to r_{15}
Add $19r_{36}$ to r_{16}, add $19r_{37}$ to r_{17}, add $19r_{38}$ to r_{18} and add $19r_{39}$ to r_{19}

Carry from r_{16} to r_{17}, from r_{17} to r_{18}, from r_{18} to r_{19} and from r_{19} to r_{20}
Add $19r_{20}$ to r_0

Carry from r_0 to r_1, from r_4 to r_5, from r_8 to r_9 and from r_{12} to r_{13}
Carry from r_1 to r_2, from r_5 to r_6, from r_9 to r_{10} and from r_{13} to r_{14}
Carry from r_2 to r_3, from r_6 to r_7, from r_{10} to r_{11} and from r_{14} to r_{15}
Carry from r_3 to r_4, from r_7 to r_8, from r_{11} to r_{12} and from r_{15} to r_{16}

Carry from r_4 to r_5, from r_8 to r_9, from r_{12} to r_{13} and from r_{16} to r_{17}
Carry from r_5 to r_6, from r_9 to r_{10}, from r_{13} to r_{14} and from r_{17} to r_{18}
Carry from r_6 to r_7, from r_{10} to r_{11}, from r_{14} to r_{15} and from r_{18} to r_{19}
Carry from r_7 to r_8, from r_{11} to r_{12} and from r_{15} to r_{16}

This makes it much easier to make efficient use of the SIMD instruction set, in particular, for modular reduction.

The idea is to arrange the operations in $\mathbb{F}_{2^{255}-19}$ into blocks of 4 equal or similar instructions, similar meaning that multiplications and squarings can be grouped together and additions and subtractions can be grouped together as well. Then these operations can be carried out using the 4-way parallel SIMD instructions in the obvious way; for example for 4 multiplications $r = a \cdot b$, $s = c \cdot d$, $t = e \cdot f$ and $u = g \cdot h$ we first produce register variables `aceg0` containing in its word elements a_0, c_0, e_0, g_0 and `bdgh0` containing b_0, d_0, e_0, g_0 and so on. Then the first coefficient of r, s, t and u can be computed by applying the `mpy` instruction on `aceg0` and `bdfh0`. All other result coefficients of r, s, t and u can be computed in a

similar way using `mpy` and `mpya` instructions.

This way of using the SIMD capabilities of CPUs was introduced in [42] as 'digit-slicing'. In our case it not only makes multiplication slightly faster (420 arithmetic instructions instead of 576 for 4 multiplications), it also allows for much faster reduction: The reduction algorithm described above can now be applied to 4 results in parallel, reducing the cost of a reduction by a factor of 4.

In Algorithm 8.2 we describe how we divide a Montgomery ladder step into blocks of 4 similar operations. In this algorithm the computation of Z_{P+Q} in the last step requires one multiplication and reduction which we carry out as described in the previous section. The computation of a ladder step again requires rearrangement of data in registers using the `shufb` instruction. Again we can hide these pipeline-1 instructions almost entirely by interleaving with arithmetic pipeline-0 instructions.

One remark regarding subtractions occurring in this computation: As reduction expects all coefficients to be larger than zero, we cannot just compute the difference of each coefficient. Instead, for the subtraction $a - b$ we first add $2p$ to a and then subtract b. For blocks containing additions and subtractions in Algorithm 8.2 we compute the additions together with additions of $2p$ and perform the subtraction in a separate step.

In total one call to the ladder-step function takes 2433 cycles.

8.5 Results and Comparison

8.5.1 Benchmarking Methodology

In order to make our benchmarking results comparable and verifiable we use the SUPERCOP toolkit, a benchmarking framework developed within eBACS, the benchmarking project of ECRYPT II [10]. The software presented in this Chapter passes the extensive tests of this toolkit showing compatibility to other curve25519 implementations, in particular the reference implementation included in the toolkit.

For scalar multiplication software, SUPERCOP measures two different cycle counts: The `crypto_scalarmult` benchmark measures cycles for a scalar multiplication of an arbi-

Algorithm 8.2 Structure of a Montgomery ladder step (see Algorithm 2.2) optimized for 4-way parallel computation

$t_1 \leftarrow X_P + Z_P$
$t_2 \leftarrow X_P - Z_P$
$t_3 \leftarrow X_Q + Z_Q$
$t_4 \leftarrow X_Q - Z_Q$
Reduce t_1, t_2, t_2, t_3

$t_6 \leftarrow t_1^2$
$t_7 \leftarrow t_2^2$
$t_8 \leftarrow t_4 \cdot t_1$
$t_9 \leftarrow t_3 \cdot t_2$
Reduce t_6, t_7, t_8, t_9

$t_{10} = a24 \cdot t_6$
$t_{11} = (a24 - 1) \cdot t_7$

$t_5 \leftarrow t_6 - t_7$
$t_4 \leftarrow t_{10} - t_{11}$
$t_1 \leftarrow t_8 - t_9$
$t_0 \leftarrow t_8 + t_9$
Reduce t_5, t_4, t_1, t_0

$Z_{[2]P} \leftarrow t_5 \cdot t_4$
$X_{P+Q} \leftarrow t_0^2$
$X_{[2]P} \leftarrow t_6 \cdot t_7$
$t_2 \leftarrow t_1^2$
Reduce $Z_{[2]P}, X_{P+Q}, X_{[2]P}, t_2$

$Z_{P+Q} \leftarrow X_{Q-P} \cdot t_2$
Reduce Z_{P+Q}

trary point; the `crypto_scalarmult_base` benchmark measures cycles needed for a scalar multiplication of a fixed base point.

We currently implement `crypto_scalarmult_base` as `crypto_scalarmult`; faster implementations would be useful in applications that frequently call `crypto_scalarmult_base`.

Two further benchmarks regard our `curve25519` software in the context of Diffie-Hellman key exchange: The `crypto_dh_keypair` benchmark measures the number of cycles to generate a key pair consisting of a secret and a public key. The `crypto_dh` benchmark measures cycles to compute a joint key, given a secret and a public key.

8.5.2 Results

We benchmarked our software on `hex01`, a QS21 blade containing two 3200 MHz Cell Broadband Engine processors (revision 5.1) at the Chair for Operating Systems at RWTH Aachen University. We also benchmarked the software on `node001`, a QS22 blade at the Research Center Jülich containing two 3200 MHz PowerXCell 8i processors (Cell Broadband Engine (revision 48.0)). Furthermore we benchmarked the software on `cosmovoid`, a Sony Playstation 3 containing a 3192 MHz Cell Broadband Engine processor (revision 5.1) located at the Chair for Operating Systems at RWTH Aachen University. All measurements used one SPU of one CBE.

SUPERCOP benchmark	`hex01`	`node001`	`cosmovoid`
`crypto_scalarmult`	697080	697080	697040
`crypto_scalarmult_base`	697080	697080	697080
`crypto_dh_keypair`	720120	720120	720200
`crypto_dh`	697080	697080	697040

Table 8.3: Cycle counts of our software on different machines

8.5.3 Comparison

To give an impression of the power of the Cell Broadband Engine for asymmetric cryptography we compare our results on a cost-performance basis with ECDH software for Intel processors.

For this comparison we consider the cheapest hardware configuration containing a Cell Broadband Engine, namely the Sony Playstation 3, and compare the results to an Intel-Core-2-based configuration running the ECDH software presented in [39]. This is currently the fastest implementation of ECDH for the Core 2 processor providing a similar security as `curve25519`. Note that this software is not protected against timing attacks.

SUPERCOP reports 365363 cycles for the `crypto_dh` benchmark (this software is not benchmarked as scalar-multiplication software). Key-pair generation specializes the scalar multiplication algorithm for the known basepoint; the `crypto_dh_keypair` benchmark reports 151215 cycles.

To estimate a price for a complete workstation including an Intel Core 2 Quad Q9300 processor we determined the lowest prices for processor, case, motherboard, memory, hard disk and power supply from different online retailers using Google Product Search yielding $296 (Mar 30, 2009).

To determine the best price for the Sony Playstation 3 we also used Google Product Search. The currently (Mar 30, 2009) cheapest offer is $221 for the Playstation 3 with a 40 GB hard disk.

The Sony Playstation 3 makes 6 SPUs available for general purpose computations. Using our implementation running at 697080 cycles (`crypto_dh` on `cosmovoid`) on 6 SPUs operating at 3192MHz yields 27474 `curve25519` computations per second. Taking the $221 market price for the Playstation as a basis, the cheapest CBE-based hardware can thus perform 124 computations of `curve25519` per second per dollar.

The Q9300-based workstation has 4 cores operating at 2.5GHz, using the above-mentioned implementation which takes 365363 cycles, we can thus perform 27368 joint-key computations per second. Taking $296 market price for a Q9300-based workstation as a basis, the cheapest Core-2-based hardware can thus perform 92 joint-key computations per second per dollar.

Note, that this comparison is not fair in several ways: The cheapest Q9300-based workstation has for example more memory than the Playstation 3 (1GB instead of 256MB).

On the other hand we only use the 6 SPUs of the CBE for the `curve25519` computation, the PPU is still available for other tasks, whereas the performance estimation for the Core-2-based system assumes 100% workload on all CPU cores.

Furthermore hardware prices are subject to frequent changes and different price-performance ratios are achieved for other Intel or AMD processors.

In any case the above figures demonstrate that the Cell Broadband Engine, when used properly, is one of the best available CPUs for public-key cryptography.

Bibliography

[1] OpenSSL library. Open source library, 1988. `http://www.openssl.org`.

[2] Barreto, P., Kim, H., Lynn, B., and Scott, M. Efficient algorithms for pairing-based cryptosystems. In *Advances in Cryptology – Crypto'2002* (2002), vol. 2442 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 377–87.

[3] Barreto, P., Lynn, B., and Scott, M. Constructing elliptic curves with prescribed embedding degrees. In *Security in Communication Networks – SCN'2002* (2002), vol. 2576 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 263–273.

[4] Barreto, P., Lynn, B., and Scott, M. On the selection of pairing-friendly groups. In *Selected Areas in Cryptography – SAC 2003* (2003). to appear.

[5] Barreto, P. S., Galbraith, S. D., Héigeartaigh, C. O., and Scott, M. Efficient pairing computation on supersingular abelian varieties. *Des. Codes Cryptography 42*, 3 (2007), 239–271.

[6] Barreto, P. S., and Naehrig, M. Pairing-friendly elliptic curves of prime order. In *Selected Areas in Cryptography: 12th International Workshop, SAC 2005* (Kingston, Canada, LNCS Vol. 3897, Feb. 2006), pp. 319–331.

[7] Barreto, P. S. L. M. The pairing-based crypto lounge. `http://paginas.terra.com.br/informatica/paulobarreto/pblounge.html`.

[8] Bernstein, D. J. qhasm: tools to help write high-speed software. `http://cr.yp.to/qhasm.html` (accessed Jan 1, 2009).

[9] Bernstein, D. J. Curve25519: new Diffie-Hellman speed records. In *Public Key Cryptography – PKC 2006* (2005), vol. 3958 of *Lecture Notes in Computer Science*, Springer, pp. 207–228.

[10] Bernstein, D. J., and (editors), T. L. eBACS: ECRYPT benchmarking of cryptographic systems, Nov 2008. `http://bench.cr.yp.to/` (accessed Jan 1, 2009).

[11] Bertoni, G. M., Chen, L., Fragneto, P., Harrison, K. A., and Pelosi, G. Computing tate pairing on smartcards, 2005. http://www.st.com/stonline/products/families/smartcard/ches2005_v4.pdf.

[12] Bhuyan, Z. I. S. M. L. Anatomy and performance of SSL processing. In *Proc. IEEE Int. Symp. Performance Analysis of Systems and Software* (2005), pp. 197–206.

[13] Blake, I., Seroussi, G., and Smart, N. *Elliptic Curves in Cryptography*. Cambridge University Press, London, 1999.

[14] Boneh, D., and Franklin, M. Identity-based encryption from the Weil pairing. extended abstract. In *Crypto '2001* (2001), vol. 2139 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 213–229.

[15] Boneh, D., Lynn, B., and Shacham, H. Short signatures from the Weil pairing. In *Advances in Cryptology – Asiacrypt'2001* (2002), vol. 2248 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 514–532.

[16] Brezing, F., and Weng, A. Elliptic curves suitable for pairing based cryptography. *Designs, Codes and Cryptography 37* (2003), 133–141.

[17] Cavanna, C. The Power Architecture Time Base register in 64-bit Linux, April 2007. http://www.ibm.com/developerworks/linux/library/pa-timebase/.

[18] CESG Communications and Electronic Security Group. http://www.cesg.gov.uk.

[19] Chen, L., and Cheng, Z. Security proof of sakai-kasahara's identity-based encryption scheme. In *In Proceedings of Cryptography and Coding 2005, LNCS 3706* (2005), Springer-Verlag, pp. 442–459.

[20] Chevallier-Mames, B., Coron, J.-S., McCullagh, N., Naccache, D., and Scott, M. Secure delegation of elliptic-curve pairing, 2005. http://eprint.iacr.org/2005/150.

[21] Chow, A. C. Programming the Cell Broadband Engine. *Embedded Systems Design* (2006). http://www.embedded.com/columns/showArticle.jhtml?articleID=188101999.

[22] Cocks, C. An identity based encryption scheme based on quadratic residues. In *VIII IMA International Conference on Cryptography and Coding* (2001), vol. 2260 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 360–263.

[23] Cohen, H., and Frey, G., Eds. *Handbook of Elliptic and Hyperelliptic Curve Cryptography*. Discrete Mathematics and its Applications. Chapman & Hall/CRC, 2006, ch. 10, Integer Arithmetic, Christophe Doche. K.H. Rosen, series editor.

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

[25] Coppersmith, D. Fast evaluation of logarithms in fields of characteristics two. In *IEEE Transactions on Information Theory* (1984), vol. 30, pp. 587–594.

[26] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 2001.

[27] Crandall, R., and Pomerance, C. *Prime Numbers: a Computational Perspective*. Springer-Verlag, Berlin, 2001.

[28] DeveloperWorks, I. Cell Broadband Engine resource center. http://www.ibm.com/developerworks/power/cell/.

[29] DeveloperWorks, I. Cell Broadband Engine SDK. http://www.ibm.com/developerworks/power/cell/.

[30] DeveloperWorks, I. Cell Broadband Engine SDK programming handbook v1.0, 2006. http://www.ibm.com/developerworks/power/cell/.

[31] Diffie, W., and Hellman, M. New directions in cryptography. *IEEE Transactions on Information Theory 22* (1976), 644–654.

[32] Diffie, W., and Hellman, M. E. New directions in cryptography. *IEEE Transactions on Information Theory IT-22*, 6 (Nov 1976), 644–654. `http://citeseer.ist.psu.edu/diffie76new.html`.

[33] Donald E. EastlaKe, K. N. Digital cryptography: A subtle art. Sample Chapter, 2002. `http://www.awprofessional.com/articles/article.asp?p=29054&seqNum=4`.

[34] Duursma, I., and Lee, H.-S. Tate pairing implementation for hyperelliptic curves $y^2 = x^p - x + d$. In *Advances in Cryptology – Asiacrypt'2003* (2003), vol. 2894 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 111–123.

[35] Ellis, J. History of ID-PKI. Web page with links to slides, 2001. `http://www.cesg.gov.uk/`.

[36] Fan, J., Sakiyama, K., and Verbauwhede, I. Elliptic curve cryptography on embedded multicore systems. In *Workshop on Embedded Systems Security - WESS 2007* (Salzburg, Austria, 2007), pp. 17–22.

[37] Freeman, D. Constructing pairing-friendly elliptic curves with embedding degree 10. In *10th Workshop on Elliptic Curves in Cryptography (ECC 2006)* (2006), Springer-Verlag, pp. 452–465.

[38] Freeman, D., Scott, M., and Teske, E. A taxonomy of pairing-friendly elliptic curves. *Journal of Cryptology*.

[39] Galbraith, S. D., Lin, X., and Scott, M. Endomorphisms for faster elliptic curve cryptography on a large class of curves. In *EUROCRYPT '09: Proceedings of the 28th Annual International Conference on Advances in Cryptology* (Berlin, Heidelberg, 2009), Springer-Verlag, pp. 518–535.

[40] Gaudry, P., and Thomé, E. The $\texttt{mp}\mathbb{F}_q$ library and implementing curve-based key exchanges. In *Proceedings of SPEED workshop* (2007). `http://www.loria.fr/~gaudry/publis/mpfq.pdf`.

[41] Gemplus. ID based Cryptography and Smartcards, 2005. http://www.gemplus.com/smart/rd/publications/pdf/Joy05iden.pdf.

[42] Grabher, P., schädl, J. G., and Page, D. On software parallel implementation of cryptographic pairings. In *Selected Areas in Cryptography – SAC 2008* (2009), vol. 5381 of *Lecture Notes in Computer Science*, p. 34âĂŞ49. to appear.

[43] Granger, R., and Stam, M. Hardware and software normal basis arithmetic for pairing based cryptography in characteristic three. *IEEE Transactions on Computers 54* (2005), 852–860.

[44] Grossschädl, J., and Savas, E. Instruction set extensions for fast arithmetic in finite fields GF(p) and GF(2^{m}). In *CHES* (2004), pp. 133–147.

[45] Hankerson, D., Menezes, A. J., and Vanstone, S. *Guide to Elliptic Curve Cryptography*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[46] Hennessy, J., and Patterson, D. *Computer Architecture - a Qualitative Approach (third edition)*. Morgan Kaufmann, 2003.

[47] Hess, F., Smart, N., Vercauteren, F., and Berlin, T. U. The eta pairing revisited. *IEEE Transactions on Information Theory 52* (2006), 4595–4602.

[48] IBM DeveloperWorks. Cell broadband engine architecture and its first implementation, Nov 2005. http://www.ibm.com/developerworks/power/library/pa-cellperf/.

[49] IBM DeveloperWorks. Cell Broadband Engine Programming Handbook (version 1.1), April 2007. http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/7A77CCDF14FE70D5852575CA0074E8ED.

[50] IBM DeveloperWorks. SPU assembly language specification (version 1.6), Sep 2007. http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/EFA2B196893B550787257060006FC9FB.

[51] IBM DeveloperWorks. Example library API reference (version 3.1), Sep 2008. http://www.ibm.com/developerworks/power/cell/documents.html.

[52] IBM DeveloperWorks. SPE cryptographic library user documentation 1.0, Sep 2008. http://www.ibm.com/developerworks/power/cell/documents.html.

[53] ICHEC. The Irish Centre for High-End Computing :-ICHEC. http://www.ichec.ie/.

[54] IEEE Std 1363-2000. Standard specifications for public-key cryptography. IEEE P1363 Working Group, 2000.

[55] Joux, A. A one round protocol for tripartite Diffie-Hellman. In *Algorithm Number Theory Symposium – IVth Sympisium* (2000), I. W. B. Ed., Ed., vol. 1838 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 385–394.

[56] Karatsuba, A., and Ofman, Y. Multiplication of multiple numbers by means of automata,. *Doklady Akad. Nauk USSR, vol. 145, no. 2, pp. 293-294, 1962 (in Russian)*.

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

[58] Koblitz, N. Elliptic curve cryptosystems. *Mathematics of Computation 48*, 177 (1987), 203–209.

[59] Kurzak, J., Buttari, A., and Dongarra, J. Solving Systems of Linear Equations on the CELL Processor Using Cholesky Factorization. *IEEE Transactions on Parallel and Distributed Systems* (2007). To appear. LAPACK Working Note 184.

[60] Lenstra, A. K. Unbelievable security. Matching AES security using public key systems. In *Advances in Cryptology – Asiacrypt 2001* (2001), vol. 2248, Springer-Verlag, pp. 67–86.

[61] Lercier, R. Discrete logarithms in GF(p). Posting to NMBRTHRY List, 2001.

[62] Levy, S. The open secret. *Wired 07*, 04 (April 1999).

[63] Mao, W. *Modern Cryptography: Theory and Practice*. Prentice-Hall, PTR. HP press, Upper Saddle River, New Jersey, USA, 2004.

[64] McCullagh, N. `http://www.computing.dcu.ie/~nmcculla`.

[65] McCullagh, N., and Barreto, P. S. L. M. Efficient and forward-secure identity-based signcryption. Cryptology ePrint Archive, Report 2004/117, 2004. `http://eprint.iacr.org/2004/117`.

[66] Menezes, A. *Elliptic Curve Public Key Cryptosystems*. Kluwer Academic Publishers, 1993.

[67] Menezes, A. J., van Oorschot, P. C., and Vanstone, S. A. *Handbook of Applied Cryptography*. CRC, http://www.cacr.math.uwaterloo.ca/hac/, October 1996.

[68] Miller, V. Uses of Elliptic Curves in cryptography. In *Advances in Crytology - CRYPTO '85*, H. Williams, Ed., pp. 417–426.

[69] Miller, V. Short programs for functions on curves. unpublished manuscript, 1986.

[70] Miyaji, A., Nakabayashi, M., and Takano, S. New explicit conditions of elliptic curve traces for FR-reduction. *IEICE Transactions on Fundamentals E84-A*, 5 (2001), 1234–1243.

[71] Montgomery, P. L. Modular multiplication without trial division,. *Mathematics of Computation 44*, 170 (1985), 519–521.

[72] Montgomery, P. L. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation 48*, 177 (1987), 243–264.

[73] Nogami, Y., and Morikawa, Y. A fast implementation of elliptic curve cryptosystem with prime order defined over f_{p^8}, 1998. `http://www.trans.cne.okayama-u.ac.jp/nogami-group/papers/kiyou(2).pdf`.

[74] OSCAR. Open source cluster application resources :- OSCAR. `http://svn.oscar.openclustergroup.org/trac/oscar`.

[75] Page, D., Smart, N. P., and Vercauteren, F. A comparison of MNT curves and supersingular curves. Cryptology ePrint Archive, 2004. http://eprint.iacr.org/2004/165.

[76] R. Rivest, A. S., and Adleman., L. A method for obtaining digital signatures and public-key cryptosystems. In *Communications of the ACM* (1978), vol. 21(2) of *Communications of the ACM*, ACM, pp. 120–126.

[77] Redhat. Fedora Core 7. http://fedoraproject.org/.

[78] Rivest, R. L. Founder RSA security and professor electrical engineering MIT. http://theory.lcs.mit.edu/~rivest/.

[79] Rodríguez-Henríquez, F., Saqib, N. A., Díaz-Pèrez, A., and Koc, C. K. *Cryptographic Algorithms on Reconfigurable Hardware (Signals and Communication Technology)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[80] RSA Data Security, Inc. *PKCS #1: RSA Encryption Standard*, June 1991. Version 1.4.

[81] Sakai, R., and Kasahara, M. ID based cryptosystems with pairing on elliptic curve. Cryptography ePrint Archive, Report 2003/054, 2003. http://eprint.iacr.org/2003/054.

[82] Sakai, R., Ohgishi, K., and Kasahara, M. Cryptosystems based on pairing. The 2000 Symposium on Cryptography and Information Security, Okinawa, Japan, 2000.

[83] Schoof, R. Elliptic curves over finite fields and the computation of square roots mod p. *Mathematics of Computation 44*, 170 (apr 1985), 483–494.

[84] Scott, M. MIRACL Multiprecision Integer and Rational Arithmetic C/C++ Library. http://www.shamus.ie.

[85] Scott, M. Computing the Tate pairing. In *CT-RSA* (2005), vol. 3376 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 293–304.

[86] Scott, M., and Barreto, P. Compressed pairings. In *Advances in Cryptology – Crypto' 2004* (2004), vol. 3152 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 140–156. Also available from `http://eprint.iacr.org/2004/032/`.

[87] Scott, M., and Barreto, P. S. Generating More MNT Elliptic Curves. *Des. Codes Cryptography 38*, 2 (2006), 209–217.

[88] Shamir, A. Identity-based cryptosystems and signature schemes. In *Springer-Verlag, Lecture Notes in Computer Science* (1984), vol. 30 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 47–53.

[89] Shimizu, K., Brokenshire, D., and Peyravian, M. Cell Broadband Engine support for privacy, security, and digital rights management applications. White paper, IBM, Oct 2005. `http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/3F88DA69A1C0AC40872570AB00570985`.

[90] Shimizu, K., Hofstee, H. P., and Liberty, J. S. Cell Broadband Engine processor vault security architecture. *IBM Journal of Research and Development 51*, 5 (Sept. 2007), 521–528.

[91] Smart, N. *Cryptography: An Introduction*. Mcgraw-Hill College, December 2004.

[92] Sotirov, A., Stevens, M., Appelbaum, J., Lenstra, A., Molnar, D., Osvik, D. A., and de Weger, B. MD5 considered harmful today, Dec 2008. `http://www.win.tue.nl/hashclash/rogue-ca/` (accessed Jan 4, 2009).

[93] Stevens, M., Lenstra, A., and de Weger, B. Nostradamus – predicting the winner of the 2008 US presidential elections using a Sony PlayStation 3, Nov 2007. `http://www.win.tue.nl/hashclash/Nostradamus/` (accessed Jan 4, 2009).

[94] SWOX / Free Software Foundation. GNU Multiple Precision Arithmetic Library. `http://gmplib.org/`.

[95] Thomé, E. Computation of discrete logarithms in $\mathbb{F}_{2^{607}}$. In *Advances in Cryptology – Asiacrypt'2001* (2001), vol. 2248 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 107–124.

[96] Weisstein, E. W. Group generators. From MathWorld–A Wolfram Web Resource. `http://mathworld.wolfram.com/GroupGenerators.html`.

[97] Wells, D. *Prime numbers; the most mysterious figures in math.* John Wiley & Sons, 2005.

A

Glossary

ADC Add with Carry

AES Advanced Encryption Standard

Altivec Floating point and integer SIMD instruction set designed and owned by Apple

BDH Bilinear Diffie-Hellman

CDH computational Diffie-Hellman

CPI Cycles per instruction

CRT Chinese Remainder Theorem

DDH Decision Diffie-Hellman

DES Data Encryption Standard

DH Diffie?Hellman

DLP Discrete logarithm problem

DMA Direct memory access

ECC Elliptic curve cryptography

ECDH Elliptic-Curve Diffie-Hellman key exchange

ELF Executable and Linkable Format

GCHQ UK Government Communications Headquarters

GMP Gnu Multi Precision

GPU Graphics Processing Unit

HSM Hardware security module

IBE Identity Base Encryption

ICHEC Irish Centre for High-End Computing

IDEA International Data Encryption Algorithm

IpSec Internet Protocol Security

KDC Key Distribution Centre

LS Local Store

MADDU Multiply and Add Unsigned

MIPS Microprocessor without Interlocked Pipeline Stages

MIRACL Multi-precision Integer and Rational Arithmetic C/C++ Library

MPM IBM multi-precision library

NIST National Institute of Standards and Technology

NSA National Security Agency

NTL Number Theory Library

OpenSSL Open Source toolkit implementing the Secure Sockets Layer

PKCS Public-Key Cryptography Standards

POSIX Portable Operating System Interface

PPC PowerPC Performance Optimisation With Enhanced RISC Performance Computing

PPU Power Processor Unit

RISC Reduced Instruction Set Computer

RSA Public key algorithm invented by Rivest, Shamir and Adleman

S/Mime Secure/Multipurpose Internet Mail Extensions

SIMD Single Instruction, Multiple Data.

SPU Synergistic Processing Unit

SSH Secure Shell

SSL Secure Sockets layer

VPN Virtual Private Network

B

Code Generation

The Ruby script in Listing B.1 is used to parse C/C++ files.

Listing B.1: Ruby code for multi-precision multiply

```
require "ftools"
unless ARGV[0]
       print "mmadd.gen usage: maddgen file.c file.c\n"
       exit
end
def expand_madd(nn,a, b, c)
  n = (nn/64)-1
  for i in (0..n)
     print "\n\ncarry=0;\n"
     for j in (0..n)
        # multiply each digit of y by x[i] #
        # z[i+j] , carry =  (x[i]*y[i] + carry + z[i+j]) #
        print "muldvd2_v(_x[#{i}],_y[#{j}],&carry,&_z[#{i+j}]);\n"
     end
     print "_z[#{n+i+1}]=carry;\n"
   end
end

fh = File.open( ARGV[0] ) text = fh.read() fh.close
text.gsub!( /(MADD\()(.*?) (\b\d+\b) (\b\d+\b) (\b\d+\b)(\);\n)/m ) {
```

```
                expand_madd( $2.to_i ,$3 ,  $4 ,  $5 )
}
```

The C/C++ code listed in B.2 holds tokens such as MADD.

Listing B.2: C code to be parsed

```
{
    MADD(256 5 6 7);
}
```

Expands to produce the C code in Listing B.3

Listing B.3: Generated 256-bit multiply as C code

```
{
carry=0;
muldvd2_v(_x[0],_y[0],&carry,&_z[0]);
muldvd2_v(_x[0],_y[1],&carry,&_z[1]);
muldvd2_v(_x[0],_y[2],&carry,&_z[2]);
muldvd2_v(_x[0],_y[3],&carry,&_z[3]);
_z[4]=carry;

carry=0;
muldvd2_v(_x[1],_y[0],&carry,&_z[1]);
muldvd2_v(_x[1],_y[1],&carry,&_z[2]);
muldvd2_v(_x[1],_y[2],&carry,&_z[3]);
muldvd2_v(_x[1],_y[3],&carry,&_z[4]);
_z[5]=carry;

carry=0;
muldvd2_v(_x[2],_y[0],&carry,&_z[2]);
muldvd2_v(_x[2],_y[1],&carry,&_z[3]);
muldvd2_v(_x[2],_y[2],&carry,&_z[4]);
muldvd2_v(_x[2],_y[3],&carry,&_z[5]);
_z[6]=carry;

carry=0;
muldvd2_v(_x[3],_y[0],&carry,&_z[3]);
```

```
muldvd2_v(_x[3],_y[1],&carry,&_z[4]);
muldvd2_v(_x[3],_y[2],&carry,&_z[5]);
muldvd2_v(_x[3],_y[3],&carry,&_z[6]);
_z[7]=carry;
return 0;
```

Where `muldvd_2_v()` is a wrapper for a 64×64-bit multiply core in Listing B.4

Listing B.4: 64-bit C code wrapper

```
#define muldvd2_v( _a, _b, _c, _rp) \
{    \
  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);\
  C=(vector unsigned int)spu_insert(*_c,(vector unsigned long long)C,0x01);\
  SPU_MULDVD2_V(AB,C,&result,*_rp);\
  *_rp=spu_extract((vec_ullong2)result,0x1);\
  *_c=spu_extract((vec_ullong2)result,0x0);\
}
```

and the code in Listing B.5 `SPU_MULDVD2_V();` is unrolled and ordered for optimal instruction ordering

Listing B.5: 64-bit multiply as spu_intrinsic code unrolled and ordered

```
\lstset{label=code::64bitUnRolled}
#define SPU_MULDVD2_V(AB,C,_rp,_usb) \
{ \
  rp_v=(vector unsigned int)si_from_ullong(_usb);\
  rp_v=spu_rlmaskqwbyte(rp_v,-8);\
  b0=spu_shuffle(AB,AB,splat_short0); \
  b1=spu_shuffle(AB,AB,splat_short1); \
  A=spu_shuffle(AB,AB,splat_shortA1); \
  p0=spu_mulo(A,b0);  \
  b2=spu_shuffle(AB,AB,splat_short2); \
  p1=spu_mulo(A,b1);\
  b3=spu_shuffle(AB,AB,splat_short3);\
  h0=spu_shuffle(p0,p0,splat_shortH0); \
```

```
 l0=spu_shuffle(p0,p0,splat_shortL0); \
 p2=spu_mulo(A,b2); \
 p3=spu_mulo(A,b3); \
 MPM_ADD_FULL_2_CARRY(psum0, l0, h0);\
 h1=spu_shuffle(p1,p1,splat_shortH1); \
 l1=spu_shuffle(p1,p1,splat_shortL1);\
 MPM_ADD_FULL_2_CARRY(psum1, l1, h1);\
 h2=spu_shuffle(p2,p2,splat_shortH2); \
l2=spu_shuffle(p2,p2,splat_shortL2); \
MPM_ADD_FULL_2_CARRY(psum2, l2, h2);\
h3=spu_shuffle(p3,p3,splat_shortH3); \
l3=spu_shuffle(p3,p3,splat_shortL3);\
MPM_ADD_FULL_2_CARRY(psum3, l3, h3);\
MPM_ADD_FULL_1_CARRY(psum0,psum0,psum3);\
MPM_ADD_FULL_2_CARRY(psum1,psum1,psum2);\
MPM_ADD_FULL_NO_CARRY(psum0,psum0,psum1);\
MPM_ADD_FULL_2_CARRY(*_rp,psum0,C);\
MPM_ADD_FULL_2_CARRY(*_rp,*_rp,rp_v);\
}
```

with `MPM_ADD_FULL_2_CARRY()` in Listing B.6

Listing B.6: 128-bit vector added to 64 bit integer with optimal carry

```
#define MPM_ADD_FULL_2_CARRY(_out_s, _in_a128, _in_a64) {   \
  _c0    = spu_genc(_in_a128, _in_a64);                     \
  _sum   = spu_add(_in_a128, _in_a64);                      \
  _t0     = spu_slqwbyte(_c0, 4);                           \
  _c0    = spu_genc(_sum, _t0);                             \
  _sum   = spu_add(_sum, _t0);                              \
  _t0    = spu_slqwbyte(_c0, 4);                            \
  _out_s  = spu_add(_sum, _t0);                             \
}
```

C

Multiple-precision Arithmetic

Computers have optimized operations (Compare, Add, Subtract, Bitwise Shift etc.) for single-precision integers. We build on these operations to provide the required basic operations of Add, Subtract, Multiply and Squaring on non negative numbers. The underlying algorithms outlined here are generally referred to as schoolbook methods and assume positive numbers. We assume the base b is the same. In practice, it is usually a power of 2 ($2^8, 2^{16}, 2^{32}$ etc.).

Multiple-precision Add

Algorithm C.1 Multiple-precision Addition

INPUT: x and y, where x has $n+1$ digits.
OUTPUT: $w = x + y$, of size $n+1$ words.

1: $carry \leftarrow 0$.
2: **for** i from 0 to n **do**
3: $\quad w_i \leftarrow (x_i + y_i + carry) \bmod b$
4: $\quad$ **if** $(x_i + y_i + carry) < b$ **then** $carry \leftarrow 0$ **otherwise** $carry \leftarrow 1$.
5: **end for**
6: $w_{n+1} \leftarrow carry$.
7: Return $((w_{n+1}w_n....w_1w_0))$.

Algorithm C.2 Multiple-precision Subtraction

INPUT: x and y, where x has $n+1$ digits and $x \geq y$.
OUTPUT: $w = x - y$, of size n words.

1: $carry \leftarrow 0$.
2: **for** i from 0 to n **do**
3: $\quad w_i \leftarrow (x_i - y_i + carry) \bmod b$
4: $\quad$ **if** $(x_i - y_i + carry) \geq 0$ **then** $carry \leftarrow 0$ **otherwise** $carry \leftarrow -1$.
5: **end for**
6: Return $((w_n w_{n-1} .... w_1 w_0))$.

Multiple-precision Subtract

When working with integers of different lengths, we must pad the smaller number with 0s to make them the same length.

Multiple-precision Multiplication

Algorithm C.3 Multiple-precision Multiplication

INPUT: x and y, where x has $n+1$ digits, and y has $t+1$ digits.
OUTPUT: $w = x.y$, of size $n+t$ words.

1: **for** i from 0 to $(n+t+1)$ **do**
2: $\quad w_i \leftarrow 0$.
3: **end for**
4: **for** i from 0 to t **do**
5: $\quad carry \leftarrow 0$.
6: $\quad$ **for** j from 0 to n **do**
7: $\quad\quad (uv)_b \leftarrow w_{i+j} + x_j \times y_i + carry$.
8: $\quad\quad w_{i+j} \leftarrow v$.
9: $\quad\quad carry \leftarrow u$.
10: $\quad$ **end for**
11: $\quad w_{i+n+1} \leftarrow u$.
12: **end for**
13: Return $((w_{n+t+1} .... w_1 w_0))$.

It is Step 7, the inner product operation, that is the computationally expensive operation. On some processors a special instruction exists to help accelerate this operation.

In some cases, the schoolbook method outlined in Algorithm C is not optimal. There are a number of alternatives. One method, by Karatsuba [56], can be faster depending on the size of the arguments and the underlying processor architecture.

Squaring is a special case of multiplication whereby inner product operation can be

reused saving clock cycles.

Multiple-precision Squaring

Algorithm C.4 Multiple-precision Squaring

INPUT: x where x has t digits.
OUTPUT: $w = x^2$, of size $2t$ words.

1: **for** i from 0 to $(2t-1)$ **do**
2: $w_i \leftarrow 0$.
3: **end for**
4: **for** i from 0 to $(t-1)$ **do**
5: $(uv)_b \leftarrow w_{2i} + x_i \times x_i$
6: $w_{2i} \leftarrow v$,
7: $carry \leftarrow u$.
8: **for** j from $(i+1)$ to $(t-1)$ **do**
9: $(uv)_b \leftarrow w_{i+j} + 2x_j \times x_i + carry$,
10: $w_{i+j} \leftarrow v$,
11: $carry \leftarrow u$.
12: **end for**
13: $w_{i+n+1} \leftarrow u$.
14: **end for**
15: Return $((w_{2t-1}w_{2t-2}....w_1w_0))$.

Squaring takes only $(n^2+n)/2$ single-precision multiplications versus n^2 for general multiplications. In practice, Algorithm C is about 20% faster than the general multiplication $u \times v$ (See [23] Section 10.3.3).

C.0.4 Modular reduction

Generally, we are more concerned with operations on the set of integers $\mathbb{Z}$ modulo m where m is positive.

Modular addition and subtraction

Modular addition and subtraction are the simplest to perform. Subtraction is the same as Algorithm C.2 if $x \geq y$. If x and y non-negitive numbers with $x, y > m$ then

1. $x + y > 2m$;

2. **if** $x \geq y$ **then** $0 \leq x - y < m$; and

3. **if** $x < y$ **then** $0 \leq x + m - y < m$;

Modular Multiplication is more complex. The obvious algorithm is simply to calculate the remainder on division by m (Algorithm C.0.4)

Algorithm C.5 Classic Modular Multiplication

INPUT: x and y and modulus m.
OUTPUT: $x \times y \bmod m$
1: Compute $x \times y$ using Algorithm C
2: Compute remainder r when $x \times y$ is divided by m
3: Return ((r)).

Montgomery Reduction

Algorithm C.0.4 is not the most efficient method for multiply - reduction. Better algorithms exist in which the steps of multiplication and reduction are interleaved. Furthermore Montgomery [71], in 1985, introduced an algorithm that can derive the result without performing a division by the modulus m. Montgomery used an ingenious representation of the residue class modulo m. This algorithm replaces division by n operations with division by a power of 2 which is extremely efficient on computers due to the numbers being represented in binary form.

Modular reduction for moduli of special form

When the modulus has a special form that can make it easier to factor, that can be especially chosen for efficient computation, we can employ yet another reduction algorithm.

Algorithm C.6 Reduction modulo $m = b^t - c$

INPUT: A base b, positive integer x, and a modulus $m = b^t - c$, where c is an l-digit base b integer for some $l < t$.

OUTPUT: $r = x \bmod m$.

1: $q_0 \leftarrow (x/b^t)$,
2: $r_0 \leftarrow x - q_0 b^t$,
3: $r \leftarrow r_0$,
4: $i \leftarrow 0$,
5: **while** $q_i > 0$ **do**
6: $\quad q_{i+1} \leftarrow (q_i c/b^t)$,
7: $\quad r_{i+1} \leftarrow q_i c - q_{i+1} b^t$,
8: $\quad i \leftarrow i + 1$,
9: $\quad r \leftarrow r + r_i$.
10: **end while**
11: **while** $r \geq m$ **do**
12: $\quad r \leftarrow r - m$.
13: **end while**
14: Return ((r)).