UNIVERSITÀ DEGLI STUDI DI SIENA

FACOLTÀ DI INGEGNERIA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Dottorato di Ricerca in Ingegneria dell'Informazione

XVII ciclo

Instruction Set Extensions for Elliptic Curve Cryptography over Binary Finite Fields

Candidate

Irina Branović

Advisors

Prof. Roberto Giorgi

Prof. Enrico Martinelli

ANNO ACCADEMICO 2003/2004

Acknowledgements

I would first like to thank to my advisor, Prof. Roberto Giorgi, who has always provided me with guidance and resources. I will be indebted with him forever for all that he has done during my stay in Italy.
Special thanks go to my B.Sc. advisor, Prof. Veljko Milutinović, who initially started me on my journey through academia that has led me here. Without his encouragement I probably would never have arrived to a Ph.D.
The work this thesis describes is built upon a lot of help and insight that I received from Sandro Bartolini, who has always been a pleasure to interact with.
I would also like to thank to my second advisor, Prof. Enrico Martinelli, for support he gave me in cryptography research.
It has been my privilege to get to know colleagues such as Marco Turchi, Leonardo Rigutini, Lorenzo Sarti, Gabriele Monfardini, Marco Ernandes and Giovanni Angelini who stood close to me in hard times and amused me in our miniature office…
Finally, I thank to my dearest: my husband Srdjan, my parents Ana and Želimir, my sister Larisa, my grandmother Margita, all my friends and family for endless emotional support without which I would not have succeeded.

Abstract

One of the most significant trends in modern computing is the drive towards the global network that allows users to communicate and share information with other users spread around the globe. Security issues play an important role in computer networks, especially with the development of e-commerce and wireless communications. As Internet is becoming more and more accessible to users, security measures will have to be strengthened. At the same time, Internet is being used on portable, battery-powered devices that allow users greater utility than ever before, but have very limited computing power. The most widespread public-key algorithm, RSA, becomes impractical due to increasing bit lengths, which is the primary concern in the applications where computing power and bandwidth are limited. While performing typical public-key operations such as modular multiplication or exponentiation on very long integer data (1024 bits and longer) is not critical on current desktop computers, the performance of public-key cryptography methods is still critical in embedded environments, i.e. in wireless, handheld Internet devices and smart cards with small memory and limited computing power.

Elliptic Curve Cryptosystems (ECC) allow for shorter operand lengths compared to other public-key cryptosystems and are thus attractive for use in many security applications, especially for embedded devices (for example, 163-bit ECC key offers similar security level as 1024-bit RSA key). The reason for this is the fact that there is no known sub-exponential algorithm to solve discrete logarithm problem on a general elliptic curve. The standard protocols in cryptography which make use of the discrete logarithm problem in finite fields, such as Diffie-Hellman key exchange, ElGamal encryption, and Digital Signature Algorithm (DSA) all

have analogues in the elliptic curve case.
Elliptic Curve Cryptosystems use finite field arithmetic. The fields of characteristic two, $GF(2^m)$, are preferred in software implementations because they provide carry-free arithmetic, and at the same time offer a simple way to represent field elements on current processor architectures.

There are two usual solutions for improving ECC performance: optimizing the software, or implementing arithmetic coprocessors in hardware. Software implementation is flexible, but does not yield acceptable performance in embedded devices, while hardware accelerators are expensive and not flexible. In this dissertation, we explored the compromise solution, and studied the possibility of extending the instruction set of embedded processors in order to improve ECC performance with only modest modifications of the processor architecture.

We report some very promising experimental results showing that very simple instruction set architecture extensions improve the efficiency of elliptic curve cryptography over $GF(2^m)$ fields in terms of execution time. Another contribution of this thesis is the performance analysis of such ISA extensions for different software implementations of ECC in embedded systems.
The thesis is organized as follows.
In Chapter 1, we explain the basics of cryptography and number theory which serve as the foundation upon which the subsequent chapters are built. In Chapter 2, we give an overview of finite field arithmetic and elliptic curve operations; the intention was to provide enough material to make the thesis understandable. Chapter 3 is devoted to the discussion different ways of implementing elliptic curve operations in software. In Chapter 4, we describe the benchmarks that we developed and used for estimating the performance of ECC cryptography algorithms, methodology for performing experiments using the SimpleScalar architectural simulator, as well as related work in this area.
The first contribution of this thesis is presented in Chapter 5, where we analyzed and presented instruction set extensions for improving the overall execution time of ECC over binary finite fields. Finally, in Chapter 6 we analyzed the effects of such extensions with different implementations of finite field arithmetic and elliptic curve operations, which can be used as a guideline for implementing ECC in embedded systems.

Contents

Chapter 1

A Primer in Cryptography

1.1 Goals of Cryptography

The increasing demand on applications for data communication and computer networks has made a cryptography a hot topic in current research. Basically, every time one needs to secure and authenticate digital data, cryptographic algorithms are employed.

Cryptography algorithms (called also ciphers) are used for encryption and decryption. They are split into two categories: *symmetric* (or private-key) and *asymmetric* (or public-key). In symmetric algorithms, the communicating parties share a common private key, which is used to transform the original message (called *plaintext*) into ciphered message (called *ciphertext*). The ciphered message is then deciphered using the same private key.
W. Diffie and M. E. Hellman in their famous 1976 paper ”New directions in cryptography” [17] demonstrated for the first time that secret communications were possible without any transfer of a secret key between sender and receiver, thus establishing the epoch of public-key cryptography. In public-key system, the key is made of two parts: a public key and a private key. The public key, known to all, is used for encrypting messages, however the ciphered message can only be de-

crypted using the receiver's private key. There is no need to exchange private keys. Modern cryptography, as applied in the commercial world, is concerned with a number of problems. The most important of these are:

- *Confidentiality*: A message sent from A to B cannot be read by somebody else. *Secrecy* and *privacy* are terms synonymous with confidentiality.
- *Authenticity*: B knows that only A could have sent the message he has just received. A *digital signature* is a cryptographic means through which the origin of a document, the identity of the sender, the time and date a document was sent, the identity of a computer user etc. can be proved.
- *Integrity*: B knows that the message from A has not been tampered with in the transit.
- *Non-repudiation*: It is impossible for A to turn around later and say that she did not send the message.

When considering cryptography, it is important to make the distinction between protocols and algorithms. A *protocol* is a specification of the complete set of steps involved in carrying out a cryptographic activity, including explicit specification on how to proceed in each case. An *algorithm* is much more narrow procedure of transforming some digital data into some other data. More details on this can be found in dedicated cryptography books, such as [47] and [63].

1.2 Private-key Cryptography

To emphasize that the same secret key is used both by encrypter and decrypter, private-key cryptosystems are also called *symmetric* cryptosystems (Figure 1.1).

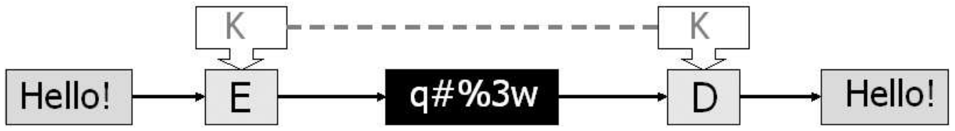

Figure 1.1. The principle of private-key cryptography.

The key K is constructed of letters in some finite alphabet, most often chosen to be the binary alphabet $0, 1$. The sender generates the plaintext P and forms the ciphertext C as the function of K. We write this encrypting transformation E as:

$$C = E_k(P)$$

The receiver must be able to invert this transformation using a decryption function D, that is:

$$P = D_k(C)$$

which expresses the fact that the plaintext P must be some function of the ciphertext C, where the particular function is determined by the secret key alone. Functions E_k and D_k must be reversible, because otherwise the ciphertext could not be decrypted:

$$D_k(E_k(P)) = P$$

Within the cipher function, a series of reversible operations is performed to randomly impress upon each of the output bits some information from each of the input bits. Almost all modern algorithms consist of multiple ”rounds” of a similar sub-algorithm. The most widely used functions in private-key ciphers are rotations, modular additions (XOR operation), modular multiplications, permutations, and substitutions.

Two classes of private-key encryption schemes are commonly distinguished: block ciphers and stream ciphers. A *block cipher* is an encryption scheme that breaks up the plaintext messages into strings (called blocks) of a fixed length, and encrypts one block at a time. *Stream ciphers* work on very small plaintext block, say a single letter or a bit, at a time. In a block cipher, the encryption of the particular plaintext block will result in the same ciphertext when the same key is used. With a stream cipher, the transformation of smaller plaintext blocks will vary, depending on when they are encountered during the encryption process. Most commercial private-key algorithms are block ciphers, with typical block size of 64 bits.

The essential feature of a private-key cryptosystem is the ”secure channel” by which the secret key is delivered to the intended receiver, protected from the prying eyes of the enemy cryptanalyst. A number of methods for securely distributing and exchanging secret keys have been invented over time, but the most popular solution is using an entity trusted by both sides, called trusted third-party. The trusted third-party maintains the database containing the secret keys of all users, authenticates their identities, and distributes session keys to users who wish to authenticate each other.

There are many high-quality private-key algorithms that will be used or are already used in popular applications and protocols; some of them are Blowfish, IDEA, Safer, FEAL, Skipjack, CAST, 3DES, RC4. The US National Institute of Standards and Technologies (NIST) had coordinated a four-year AES (Advanced Encryption Standard) competition which ultimately lead to selection of *Rijndael* algorithm [57] as the new US encryption standard in 2000. Rijndael replaced the Data Encryption Standard (DES) which had been standard since 1977. The evaluation criteria of the AES competition were divided into three major categories: security, cost, characteristics of algorithm and implementation, in the order of importance. Rijndael has a consistently good performance in both hardware and software across a wide range of computing environments, and its very low memory requirements make it very well suited for restricted-space environments, in which it demonstrates excellent performance. However, it will take some time before Rijndael gets wide acceptance in practice. For example, in SSL Handshake Protocol [55], a number of private-key ciphers are used: RC2, RC4, IDEA, DES, and 3DES, PGP e-mail software uses 3DES, IDEA, and CAST ciphers, while Kerberos protocol uses DES cipher for authentication and encryption.

Although private-key cryptography is adequate for many applications, it has some disadvantages [46]:

- *Key distribution problem*: Two users have to select a key in secret before they can initiate the communication over the insecure channel.

- *Key Management Problem*: In a network of n users, every pair of users must share a secret key, for a total of $n(n-1)/2$ keys. If n is large, than the number of keys becomes unmanageable.

- *No signatures possible*: A *digital signature* is an electronic analogue of a hand-written signature. In a private-key cryptosystem, A and B have the same capabilities of encryption and decryption, and thus B cannot convince a third party that a message he received from A is in fact originated from A.

1.3 Public-key Cryptography

In a public-key cryptosystem, each entity has a public key and a corresponding private key. To encrypt a message using a public-key cipher, it is first converted to ciphertext using the receiver's public key. The resulting ciphertext can only be deciphered using the receiver's private key (Figure 1.2). The public key need not to be kept secret, in fact, it should be widely available – only its authenticity is required to guarantee that the person to whom it belongs is indeed the only party who knows the corresponding private key. Anyone can subsequently send encrypted messages that only that person can decrypt. Therefore, it should be noted that public-key systems eliminate the need for secure channel to distribute keys, but do not eliminate the need for authentication.

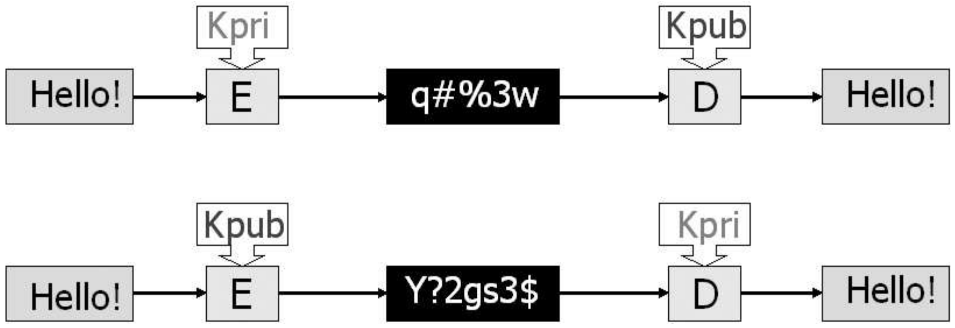

Figure 1.2. The principle of public-key cryptography.

The two important unusual definitions in public-key cryptography are those of a ”one-way function”, and of a ”trapdoor one-way function”. A *one-way function* is defined as a function f such that for every x in the domain of f, $y = f(x)$ is easy to compute, but it is *computationally infeasible* to compute the inverse, that is to find an x such that $y = f(x)$ for *virtually all* y in the range of f. It is believed that there are many functions that are computationally easy to compute, but computationally infeasible to reverse. However, there is still no rigorous mathematical proof that one-way functions are really so hard to reverse as we believe.

It is less clear how a one-way function could be of cryptographic use: to build a cipher that even the legitimate receiver could not decipher seems absurd. By defining a *trapdoor one-way function* as one-way function with the additional property that, given some extra information (called *trapdoor information*), it becomes feasible to find an x such that $f(x) = y$, the usefulness for cryptography becomes obvious.

As a likely candidate for a one-way function, Diffie and Hellman [17] proposed the discrete exponential function:

$$f(x) = \alpha^x \bmod p$$

where p is a prime number and α $(1 \leq \alpha \leq p)$ an integer such that $\alpha \bmod p$, $\alpha^2 \bmod p$, ..., $\alpha^{p-1} \bmod p$ are, in some order, equal to $1, 2, \ldots, p-1$. For example, with $p = 7$, one could take $\alpha = 3$ because $\alpha \bmod 7 = 3$, $\alpha^2 \bmod 7 = 2$, $\alpha^3 \bmod 7 = 6$, $\alpha^4 \bmod 7 = 4$, $\alpha^5 \bmod 7 = 5$, and $\alpha^6 \bmod 7 = 1$. In algebraic terminology, such an α is called a *primitive element* of a finite field $GF(p)$, and such elements are known to exist [38].

If $y = \alpha^x \bmod p$, than $x = log_\alpha(y)$, so that inverting $f(x)$ becomes the problem of calculating discrete logarithms. The *discrete logarithm problem* is, therefore, defined as follows: given an element α in a finite field $GF(p)$ and another element y in the same field, find an integer x such that $y = \alpha^x \bmod p$. Currently the best known method of calculating discrete algorithms is the *number field sieve* method [12].

In a public-key cryptosystem, the private key is always linked mathematically to the public key. Therefore, it is always possible to attack a public-key system by deriving the private key from the public key. Typically, the defense against this is to make problem of deriving the private key from the public key as difficult as possible, i.e. to find a good trapdoor function. Another well-known source of trapdoor one-way functions is the integer factorization problem, which relies on the fact that factoring very large integers is computationally infeasible. Many other one-way functions have been proposed during time, some almost immediately been exposed as insecure, others still appear promising, but no one has yet produced a proof that any function is indeed a one-way function.

1.3.1 Some definitions from Number Theory

The primary mathematical constructs used in public-key cryptography are those of a finite group and field. Each of these constructs consists of a non-empty set of elements, together with one on more functions that operate on elements of the set to generate other elements of the set. The most general of these structures is an abelian group, which is formally defined below.

Definition 1.3.1 *An abelian group, G, is a non-empty set of elements together with a binary operator $\cdot$ which exhibit the following properties:*

1. *$a \cdot b \in G$ for all $a, b \in G$ (i.e., G is closed under the operation $\cdot$)*
2. *$(a \cdot b) \cdot c = a \cdot (b \cdot c)$ for all $a, b, c \in G$ (i.e., $\cdot$ obeys the associative law)*
3. *There exists an identity element $e \in G$ for all $a \in G$ such that $a \cdot e = e \cdot a = a$*
4. *For all $a \in G$, there exist an inverse element $a^{-1} \in G$ such that $a \cdot a^{-1} = e$*
5. *$a \cdot b = b \cdot a$ for all $a, b \in G$ (i.e., $\cdot$ is commutative)*

An example of an abelian group is that of the integers under the addition operation, with the identity element $e = 0$ and inverse $a^{-1} = -a$. In cryptography, groups typically used have a finite number of elements, the number which is referred to as the *order* of the group. A finite group G is said to be cyclic if all elements of the group can be generated by repeated application of the group operation $\cdot$ to an element $a \in G$ which is denoted as a *generator* of the group G. The order of an element a of a finite cyclic group G is the smallest positive integer b such that

$$a \cdot a \cdot a \ldots \cdot a \quad (b \quad times) = e$$

Hence, the order of a generator of G will be the order of the group, and the order of any element $a \in G$ will always divide the group order.

Definition 1.3.2 *A field, F, is a non-empty set of elements together with two binary operators, $+$ and $*$, which exhibit the following properties:*

1. *F is an abelian group under the operation $+$*
2. *The non zero elements of F form an abelian group under the operation $*$*
3. *Distributivity over $(+, *)$ applies*

Given a finite field with q elements (i.e., of order q), we commonly refer to it as a Galois field, denoted as $GF(q)$, which can be shown to be unique and to exist for all $q = p^m$, where p is a prime called the *characteristic* of GF, and m is a positive integer. If $m = 1$, then GF is called a prime field; if $m \geq 2$, then GF is called an extension field. Furthermore, it can also be shown that a set of integers modulo a prime p is a field. Finite fields of order 2^m are called *binary fields* or *characteristic-two* finite fields.

Definition 1.3.3 *Let $gcd(i, n)$ denote the greatest common divisor of the integers i and n (for example, $gcd(12, 18) = 6$). The Euler's totient function $\Phi(n)$, where n is a positive integer, is defined as the number of positive integers $1 < i < n$ such that $gcd(i, n) = 1$. For a prime p, $\Phi(p) = p - 1$, and if p and q are distinct primes, $\Phi(p \cdot q) = (p-1) \cdot (q-1)$. For instance, $\Phi(6) = \Phi(2 \cdot 3) = \Phi(2) \cdot \Phi(3) = 1 \cdot 2 = 2$.*

Definition 1.3.4 *Euler's theorem: for any positive integers x and n such that $x < n$, $x^{\Phi(n)} \bmod n = 1$, provided that $gcd(x, n) = 1$.*

1.3.2 RSA

It was not clear to Diffie and Hellman whether a trapdoor one-way function existed, so it was left to R. Rivest, A. Shamir, and L. Adleman of MIT to make the first proposal of such a function in their famous 1978 paper "A Method for Obtaining Digital Signatures and Public-key Cryptosystems" [58]. The RSA trapdoor one-way function is remarkably simple:

$$f(x) = x^e \bmod n$$

where x is a positive integer such that $x < n$, n is a product of two distinct, very large primes p, q such that $\Phi(n) = (p-1) \cdot (q-1)$ can be factorized as a product of a very large prime factor and some other number, and e is a positive integer such that $e < \Phi(n)$ and $gcd(e, \Phi(n)) = 1$. Trapdoor information is (p, q, e). Euclid's theorem states that for e chosen in this way, there is unique d such that $0 < d < \Phi(n)$ and $d \cdot e \bmod \Phi = 1$, moreover d can be found in the process of using Euclid's Extended algorithm for computing $gcd(n, e)$ [38]. It is believed that it is computationally impossible to invert this function when one knows only n and e. The security of RSA cryptosystem lies in the fact that the only way of inverting f is equivalent to factoring $n = p \cdot q$, but this assumption has never been proved. The two primes p and q should be of roughly equal length, because this makes the modulo n harder to factorize than if one of the primes is much smaller than the other. Public exponents in common use today are 3 and $2^{16}+1$; because these numbers are small, RSA encryption is fast relative to decryption.

The RSA encryption is built into hundreds of millions of copies of the most popular applications, including Web browsers (SSL, TLS), Internet Security Protocols (IPSec), SSH, e-commerce systems, e-mail clients (S/MIME, PGP). In practice, RSA is very slow compared to symmetric ciphers, especially in key generation. RSA is not well suited for limited environments like PDAs, mobile phones, and smart cards because it is hard to implement large modular arithmetic in such environments.

Example: RSA Encryption with artificially small parameters

1. Alice chooses primes $p = 2357$, $q = 2551$, computes $n = p \cdot q = 6012707$, and Euler's totient function $\Phi(n) = (p-1) \cdot (q-1) = 6007800$.
2. Then chooses $e = 3674911$ and, using the extended Euclidean algorithm, finds $d = 422191$ such that $e \cdot d \bmod \Phi = 1$.
3. To encrypt a message $M = 5234641$, Bob uses Alice's public key (e, n) to compute $C = M^e \bmod n = 3650502$ and sends this to Alice.
4. To decrypt a message, Alice computes $M = C^d \bmod n = 5234671$.

1.3.3 Diffie-Hellman Key Exchange Protocol

Diffie and Hellman suggested a simple way in which the discrete logarithm can be used to create secret keys between pair of users using only public messages.
All users are presumed to know the public domain parameters (p, q, g) where p is a prime, q is a prime divisor of p, and $g \in [1, p-1]$ has the order q (i.e. q is the smallest positive integer satisfying $g^q \equiv 1 \bmod p$).

Diffie-Hellman Key Exchange

1. If Alice and Bob want to exchange a key, Alice should choose a secret large integer $a \in [1, q-1]$ and Bob should choose an integer $b \in [1, q-1]$.
2. Alice computes and publishes her public key $K_A = g^a \bmod p$
3. Bob computes and publishes his public key $K_B = g^b \bmod p$
4. When Bob wants to communicate secretly with Alice, he fetches K_A from the directory, and than uses his private key b to form a shared secret key $K_{AB} = (K_A)^b \bmod p = (g^a)^b \bmod p = g^{ab} \bmod p$.
5. Alice similarly calculates the shared key $K_{AB} = (K_B)^a \bmod p =$ $g^{ab} \bmod p$.

If an enemy could solve the discrete logarithm problem, he could take K_A and K_B from the directory, solve $b = log_g K_B$, and then form K_{AB} in the same manner as Bob did. There seems to be no other way for an enemy to find K_{AB}, but again there is no proof of this. Although this is the oldest proposal for eliminating the transfer of secret keys in cryptography, it is still generally considered to be one of the most secure and practical public-key schemes.

1.3.4 ElGamal Cryptosystem

In 1985, T. ElGamal [19] proposed the following cryptosystem:

ElGamal encryption/decryption

1. Alice and Bob choose a large prime p, a prime q which is a divisor of $p-1$, and $g \in [1, p-1]$ of order q.
2. Bob chooses a secret large integer $b \in [1, q-1]$ (his *private key*) and publishes his public key $\beta = g^b \bmod p$
3. If Alice wants to send a message m to Bob, Alice generates a random large integer a and calculates her public key $\alpha = g^a \bmod p$
4. Alice takes Bob's public key β (that corresponds to g^b mod p), calculates $m' = m\beta^a \bmod p$ and sends to Bob a copy (α, m')
5. Bob reconstructs the message $m = m'\alpha^{-b}$

It can easily be seen that the security of the ElGamal cryptosystem and the Diffie-Hellman key exchange are equivalent, and based on the difficulty of the discrete logarithm problem.

1.3.5 Digital Signature Algorithm (DSA)

Digital signature of the document is a cryptographic means of ensuring the identity of the sender and the authenticity of data. It is information based on both the document and signer's private key (Figure 1.3).

In 1991, the US National Institute of Standards and Technology (NIST) proposed a Digital Signature Standard (DSS) [53]. This standard requires the use of Secure Hash Algorithm (SHA-1), specified in the Secure Hash Standard [52]. The SHA-1 algorithm takes a long message and produces its 160-bit digest, in a way that is infeasible to invert in practice; this method is known as *hashing*. The message digest is then encrypted using the private key of the signer; the signature can be verified using the sender's public key.

As with Diffie-Hellman and ElGamal, users know public domain parameters: a large prime p, q which is a prime divisor of $p-1$, and $g \in [1, p-1]$ of order q

Digital Signature Algorithm

Signature Generation

1. Alice generates her private key $a \in [1, q-1]$ and a random integer $k \in [1, q-1]$
2. Alice computes $r = (g^k \bmod p) \bmod q$ and the message hash $H(m)$ (obtained by applying the SHA-1 algorithm).
3. Alice solves the equation $s = k^{-1}(H(m) + ar)$.
4. Alice sends to Bob the signature (the pair (r, s)) and the message m.

Signature verification

1. Bob computes $w = s^{-1} \bmod q$
2. Then computes $u_1 = H(m)w \bmod q$ and $u_2 = rw \bmod q$.
3. Bob computes $v = ((g^{u_1} y^{u_2}) \bmod p) \bmod q$.
4. If $v = r$, signature is verified, otherwise the message has been changed.

1.4 The Future of Cryptography

It seems that the benefits of public-key systems would eliminate the need for private-key systems. However, computational performance of private-key systems is superior to that of a public-key systems due to huge computational power needed to deal with large integers. The primary benefit of using private-key cryptography

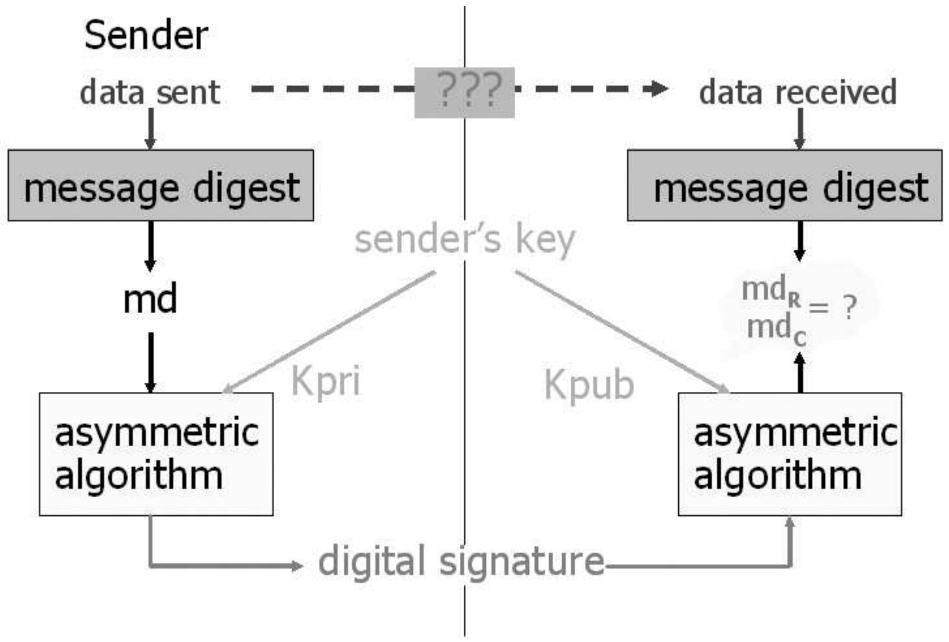

Figure 1.3. Digital signature is information based both on the document and a signer's private key, and can be verified using the signer's public key.

is the speed on which private-key algorithms operate; for example, the encryption/decryption speeds for AES finalists are on the order of tens of megabits per second using conventional processors [9]. In comparison, public-key algorithms typically operate at speeds at the order of tens of kilobytes per second.

Private keys in public-key systems must be larger (1024 bits and more) than private keys in private-key cryptosystems: while the most effective attack on private-key systems is a brute-force attack, all known public-key systems are subject to ”short-cut” attack (for example, factoring large integers). Public-key cryptography is not meant to replace private-key cryptography, but rather to supplement it, to make it more secure; private-key cryptography still remains extremely important and is the subject of much ongoing study and research. A good solution is to combine public and private-key cryptosystems in order to get both the security advantages of public-key systems and the speed advantages of private-key systems.

Chapter 2

Elliptic Curve Cryptography

All public-key algorithms work for a general abelian group, so one could consider various groups to use in such protocols. However, since the protocols are to be realized in hardware or software, the group operation should be simple to implement. One of the most interesting proposals, done independently by Koblitz [37] and Miller [48] is to use the group of points of an elliptic curve defined in a finite field, that allows for the definition of an operation called ”addition” of the points of the elliptic curve.

There are two primary reasons for using elliptic curves as an alternative to standard public-key methods based on discrete logarithm problem. First of all, the field over which an elliptic curve is defined can be chosen so that it would be easier to implement ECC methods in limited environments such as PDAs and mobile phones. Subsequently, even the physical space necessary for constructing cryptochips can be significantly reduced if elliptic curve methods are used.
Second, a problem all cryptographic systems are facing with is the key length necessary for maintaining a certain level of security. This problem is not caused by the cryptographic method itself, but by the fact that the operations are performed in finite field. At the time being, the discrete logarithm problem in elliptic curve groups seems harder than the discrete logarithm problem in the group of prime numbers.

A n bit key in elliptic curve cryptosystem offers the same level of security as a N bit key in standard cryptosystem [46] where

$$n = 4.91N^{1/3}(log(Nlog2))^{2/3}$$

Security level of k bits means that the best algorithm for breaking the system takes approximately 2^k steps. Consequently, when using elliptic curves, the same level of security is obtained with shorter keys. Table 2.1 shows key lengths used in symmetric (private-key) ciphers, ECC, and RSA that offer approximately same level of security.

Symmetric cipher key length	ECC key length	RSA key length
80	160	1024
112	224	2048
128	256	3072
192	384	7680
256	512	15360

Table 2.1. Key lengths of private-key ciphers, ECC and RSA that offer approximately same level of security.

2.1 Elliptic Curves

An elliptic curve in $\mathbb{R}$ or $\mathbb{C}$ is the set of points that satisfy the general Weierstrass equation in projective form:

$$Y^2Z + a_1XYZ + a_3YZ^2 = X^3 + a_2X^2Z + a_4XZ^2 + a_6Z^3 \tag{2.1}$$

The curve is regular or *non-singular* if in equation

$$F(X,Y,Z) = Y^2Z + a_1XYZ + a_3YZ^2 - X^3 - a_2X^2Z - a_4XZ^2 - a_6Z^3 \tag{2.2}$$

in every point P of the curve at least one of the three derivates $\frac{\partial F}{\partial X}, \frac{\partial F}{\partial Y}, \frac{\partial F}{\partial Z}$ is different from zero.

The point $(X,Y,Z) = (0,1,0)$ is called the point at infinity of the curve and is indicated as O.

Removing the common Z^3 term yields the affine (non homogeneous) form of the Weierstrass equation:

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \tag{2.3}$$

Unlike its projective counterpart, in affine coordinates the point at infinity does not have a corresponding coordinate value. Instead, the point at infinity is commonly mapped to the point $(x, y) = (0, 0)$, an invalid curve point for $a_6 \neq 0$, which simplifies the implementation of point validity checking. The decision whether to use projective or affine coordinates is based primarily on implementation aspects regarding the availability of memory for storing temporary values and the relative performance of the field inversion and multiplication algorithms used to implement the group operations of point addition and doubling. Two elliptic curves are *isomorphic* if with appropriate change of variables one curve can be transformed into another.

If we define the following relations:

$$\begin{gathered}
d_2 = a_1^2 + 4a_2 \\
d_4 = 2a_4 + a_1a_3 \\
d_6 = a_3^2 + 4a_6 \\
d_8 = a_1^2a_6 + 4a_2a_6 - a_1a_3a_4 + a_2a_3^2 - a_4^2 \\
c_4 = d_2^2 - 24d_4 \\
\Delta = -d_2^2d_8 - 8d_4^3 - 27d_6^2 + 9d_2d_4d_6 \\
j(E) = c_4^3/\Delta
\end{gathered}$$

we define a curve E as *non singular* if and only if $\Delta(E) \neq 0$ and two curves $E1$ and $E2$ *isomorphic* if and only if $j(E1) = j(E2)$. From a cryptographic standpoint, singular curves should be avoided as they compromise the security of the system (the MOV attack [11]).

2.2 Point Addition and Doubling

Given a general affine form of the elliptic curve equation (2.3), one can define a group operation of adding the two points of a curve E. This operation, given two points $P, Q \in E$ returns a point, called the sum of P and Q, $P + Q \in E$ and is defined as:

1. $O + P = P + O = P$ (O serves as the identity element.)
2. $-O = O$
3. If $P = (x, y) \neq O$ then $P = (x, -y - a_1x - a_3)$
4. If $Q = -P$ then $P + Q = O$
5. If $P = Q$ and R is the point of intersection of the line tangent to E in P, then $P + P = -R$ (point doubling)
6. Otherwise given two distinct points $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ lying on E, assuming that $x_1 \neq x_2$, their sum is given as that point R which is the reflection about the x axis of the point of intersection of the line containing P and Q and the curve E. If the two point have the same x coordinate, then the resulting line containing the points will be vertical, and is said to intersect the curve at point at infinity.

It can be demonstrated that $(E, +)$ is an Abelian group in which O is a neutral element and $-P$ is the additive inverse of P. The explicit formulas for calculating the coordinates of $P + Q$ can be found using simple algebraic calculations.
Let $P = (x_1, y_1)$, $Q = (x_2, y_2)$ and $P + Q = (x_3, y_3)$.
If $P \neq Q$, the equation of the tangent line that passes through P and Q is
$y = \lambda x + \beta$ where

$$\lambda = \begin{cases} \frac{y_2 - y_1}{x_2 - x_1} & \text{if } P \neq Q \\ \frac{3x_1^2 + 2a_2x_1 + a_4 - a_1y_1}{2y_1 + a_1x_1 + a_3} & \text{if } P = Q \end{cases}$$

In both cases $\beta = y_1 - \lambda x_1$. After substitution of y in the elliptic curve equation, we obtain a third degree equation, with two solutions x_1 and x_2. The third solution can be obtained directly, without solving the equation, using Newton's identity:

$$x_3 = \lambda^2 + a_1\lambda - a_2 - x_1 - x_2 \tag{2.4}$$

and subsequently

$$y_3 = (-\lambda^2 + a_1)x_3 - \beta - a_3 \tag{2.5}$$

2.3 Scalar Point Multiplication

The product of a point and an integer is an operation which, given a point P and an integer k, produces a point on the curve, indicated as $[k]P$. This operation is the exact analogue of the exponentiation in a group and is defined in the following manner.
Given $k \in Z$ and $P \in E(GF(q))$ we define

$$[k]P = \begin{cases} [k-1]P + P & k > 0 \\ O & k = 0 \\ -[-k]P & k < 0 \end{cases}$$

$[k]P$ has the properties:

- $[0]P = O, [1]P = P$
- $[m+n]P = [m]P + [n]P$
- $[k](P+Q) = [k]P + [k]Q$
- $[m]([n]P) = [mn]P$

Multiplication of a point P by an integer k is repeated addition to the point to itself k times. Calculating elliptic curve scalar point multiplication is equivalent to exponentiation in discrete logarithm systems: given an integer k, it is relatively easy to compute $Q = [k]P$, but it is very difficult and time consuming to reverse the operation, i.e. to find k knowing only P and Q; finding such an integer k is equivalent to calculating the elliptic curve discrete logarithm of Q to the base P.

2.4 Elliptic Curves in Finite Fields $GF(2^m)$

The curve can also be studied in any finite field $GF(q)$, but for cryptographic purposes, we are interested in fields with finite number of elements. The points in $GF(q)$ that satisfy the equation 2.3 are called rational points of the curve in $GF(q)$. Set of these points, together with the point at infinity O, is denoted as $E(GF(q))$. Naturally, if we are working in finite fields, the curve looses all of its ”differential properties”, but keeps many algebraic properties, for example the possibility of defining the sum of points in the same way as in the fields of real or complex numbers. In the fields of characteristic 2, $GF(2^m)$, often used in applications of elliptic curve cryptography, the curve has a different form

$$y^2 + xy = x^3 + ax^2 + b \tag{2.6}$$

obtained by applying an appropriate change of coordinates.
The addition/doubling formula becomes

$$\begin{aligned} x_3 &= \lambda^2 + \lambda + a + x_1 + x_2 \\ y_3 &= \lambda(x_1 + x_3) + x_3 + y_1 \\ \lambda &= \begin{cases} \frac{y_2+y_1}{x_2+x_1} & \text{if } P \neq Q \\ \frac{x_1+y_1}{x_1} & \text{if } P = Q \end{cases} \end{aligned} \tag{2.7}$$

The inverse of the point $P = (x, y)$ in this case is $P = (x, y + x)$.
The above equations show that elliptic curve point addition can be calculated with number of additions, multiplications, squarings, and inversions in the underlying field. For example, doubling of elliptic curve point in $GF(2^m)$ field requires one inversion, two multiplications, one squaring, and six additions (division is performed by calculating multiplicative inverse, followed by multiplication).

The *order* of a point on a curve is the smallest integer k such that $[k]P = O$; of course, such a finite k does not have to exist. In cryptography terms, elliptic curve key size refers to the minimum order of the chosen base point P on the elliptic curve; this order should be slightly smaller than the field size.

2.5 Choosing an Appropriate Elliptic Curve

In the implementation of the elliptic curve cryptosystems, the following factors must be taken into account:

- An elliptic curve E and the finite field $GF(q)$ have to be chosen in a way that the discrete logarithm problem of $E(GF(q))$ is difficult
- The finite field and an efficient representation of the field elements have to be chosen in a way that the operations are easily implemented

- The alternative representations of the curve (projective etc.) can be taken into consideration
- The efficient algorithms for calculating $[k]P$ can be studied.

Various studies have demonstrated that the elliptic curves appropriate for cryptography must have the following properties [39]:

- The number of points on the curve, $\#E(GF(q))$, is divisible by a large prime n
- $\#E(GF(q)) \neq q$, where $q = \#GF(q)$ (field order)
- n does not divide $qk - 1$, for small k
- If $q = 2^m$ is used, then m has to be prime

Special type of curves, known as binary anomalous or Koblitz curves, were also proposed for cryptographic use. These are two elliptic curves over $GF(2^m)$ with coefficients a and b either 0 or 1:

$$E_0 : y^2 + xy = x^3 + 1$$
$$E_1 : y^2 + xy = x^3 + x^2 + 1$$

Koblitz curves have the following property: if (x, y) is a point on a Koblitz curve, then (x^2, y^2) is also a point on that curve. The additional property of Koblitz curves:

$$(x^4, y^4) + 2P = (x^2, y^2)$$

allows for using very efficient $[k]P$ methods. The efficient arithmetic on Koblitz curves is described in [66].

2.6 Elliptic Curve Implementation of Standard Public-Key Algorithms

Basically, almost all public-key methods (except the famous RSA algorithm that does not seem to offer any advantage if translated to elliptic curve scheme) remain valid if instead of integer multiplication operation scalar elliptic curve point multiplication is used. Diffie-Hellman, ElGamal and various forms of digital signature systems using elliptic curves have much shorter key lengths compared to traditional variants of the same cryptosystems, while at the same time offer the same level of security.

2.6.1 Elliptic Curve Diffie-Hellman Key Exchange

Consider Alice and Bob want to agree upon the key for exchanging secret messages. They first choose a finite field, an elliptic curve E defined upon it, and a point P on that curve.

Elliptic Curve Diffie-Hellman Key Exchange (ECDH)

1. To generate her private key, Alice chooses an integer a, and then multiplies the point P by a to get another point $Q = [a]P$ on the same curve. The point Q serves as the public key.
2. Bob does the same: he chooses an integer b as his private key, calculates $R = [b]P$ and publishes R as his public key.
3. When Alice and Bob want to communicate using the same, shared key, Alice takes Bob's public key and multiplies it with her private key, to get another point on the curve $[ab]P$. Bob does the same, so they share the same key $[ab]P$.

Without solving the discrete logarithm problem (finding a knowing P and $[a]P$), there seems to be no other way to compute $[ab]P$ knowing only $[a]P$ and $[b]P$.

2.6.2 Elliptic Curve ElGamal Encryption/Decryption

Alice and Bob first choose a finite field, an elliptic curve E defined upon it, and a point P on that curve. Suppose that the set of messages has been transformed into points of chosen elliptic curve E in some agreed upon way (see [11]).

Elliptic Curve ElGamal Encryption/Decryption (ECELG)

---

1. Alice chooses her private key a and publishes her public key $[a]P$.
2. Bob chooses his private key b and publishes his public key $[b]P$.
3. If Bob wants to send Alice a message $M \in E$, he calculates $b[aP]$ and sends to Alice a pair of elliptic curve points $(bP, M + abP)$.
4. To decipher a message, Alice multiplies the first point in the pair by her private key a and then subtracts the result from the second point in the pair: $M = M + [a]bP - abP$.

2.6.3 Elliptic Curve Digital Signature Algorithm

Suppose that Bob wants to sign a message $M \in E$. As before, Alice and Bob agreed upon finite field, an elliptic curve E (with coefficients a and b), a positive prime integer r dividing the number of points on E, and the base point P of order r on that curve.

Elliptic Curve Digital Signature Algorithm (ECDSA)

Signature Generation

---

1. Bob generates a random integer k and publishes $Q = [k]P = (x_Q, y_Q)$.
2. Then computes an integer $c = x_Q \bmod r$.
3. Bob computes an integer $d = k^{-1}(H(m) + ac) \bmod r$, where $H(M)$ is the message digest obtained by applying the SHA-1 algorithm to the message M.

4. Bob sends to Alice the signature (c, d) and the message M.

Signature Verification

---

1. Alice calculates the integers $h = d^{-1} \bmod r$, $h_1 = H(M)h \bmod r$ and $h_2 = ch \bmod r$.
2. Then computes an elliptic curve point $R = h_1P + h_2Q = (x_R, y_R)$.
3. Alice computes the integer $c' = x_R \bmod r$.
4. If $c' = c$ signature is verified, otherwise it is invalid.

2.7 ECC Standards

Some researchers have expressed concern that the basic problem on which elliptic curve systems are based has not been looked at in as much detail as, say, the factoring problem, on which systems as RSA are based. However, all such systems based on the perceived difficulty of a mathematical problem live in fear of a dramatic breakthrough to some extent, so this issue is not addressed further. Standards for elliptic curve are currently being drafted by various accredited standards bodies around the world; some of this work is summarized below. As these drafts become officially adopted by the appropriate standards bodies, one can expect elliptic curve systems to be widely used by providers of information security.

1. The Elliptic Curve Digital Signature Algorithm (ECDSA) was adopted in January 1999 as an official American National Standards Institute (ANSI) standard X9.62 [3], [4]. The ANSI Financial Services working group is also drafting a standard for elliptic curve key agreement and transport protocols.
2. In February 2000, FIPS 186-1 was revised by National Institute of Standards and Technology (NIST) to include the elliptic curve digital signature algorithm (ECDSA) as specified in ANSI X9.62 with further recommendations

for the selection of underlying finite fields and elliptic curves. FIPS 186-2 [53] recommended 10 finite fields: 5 prime and 5 binary fields. For each of the prime fields, one randomly selected elliptic curve was recommended, while for each of the binary fields one randomly selected elliptic curve and one Koblitz curve was selected.

3. Elliptic curves are in the draft IEEE P1363 [31] (Standard Specifications for Public-Key Cryptography), which includes encryption, signature, and key agreement mechanisms. Elliptic curves over $GF(p)$ and $GF(2^m)$ are both supported. P1363 also includes discrete logarithm systems in subgroups of the multiplicative group of integers modulo a prime, as well as RSA encryption and signatures.

4. ECC Cipher Suites for TLS draft [18] of the Internet Engineering Task Force (IETF) describes additions to TLS to support elliptic curve cryptography. In particular it defines new key exchange algorithms which use Elliptic Curve Digital Signature Algorithm (ECDSA), Elliptic Curve Diffie-Hellman (ECDH) key agreement scheme, and defines how to perform client authentication with ECDSA and ECDH.

5. The ISO/IEC 15946 draft standard [32], [33] specifies various cryptographic techniques based on elliptic curves including signature schemes, public-key encryption schemes, and key establishment protocols.

Elliptic Curve Cryptosystems have no general patents, though some efficient implementation techniques are covered by patents of companies Apple Computer, Certicom [14], and Cylink. However, in all three cases, it is the implementation technique that is patented, not the representation, and there are alternative representations that are not covered by patents. The patent issue for elliptic curve cryptosystems is the opposite of that for RSA and Diffie-Hellman, where the cryptosystems themselves were patented, but the efficient implementation techniques were not.

Chapter 3

Software Implementation of ECC over $GF(2^m)$

3.1 Finite Field Operations

The representation used for the elements of the underlying field $GF(q)$ can have significant impact on the feasibility, cost, and speed of an elliptic curve cryptosystem. The representation used for a particular field, however, does not seem to affect its security. The advantage of elliptic curve systems is that the underlying field and a representation of its elements can be selected so that the field arithmetic (addition, multiplication, and inversion) can be optimized. With the current knowledge, elliptic curve systems over prime fields $GF(p)$ appear to provide the same level of security as elliptic curve systems over characteristic two fields $GF(2^m)$ when $p \approx 2^m$. Because it appears that arithmetic in $GF(2^m)$ can be implemented more efficiently in hardware and software than arithmetic in $GF(p)$, elliptic curves over binary finite fields have seen wider use in commercial applications. This is also the reason we chose to explore ECC over $GF(2^m)$ in this dissertation.

3.1.1 Basis Representation

To construct a field with p^m elements, denoted as $GF(p^m)$, a polynomial $f(x)$ of the degree m *irreducible* over $GF(p)$ is taken. A polynomial $f(x)$ is said to be irreducible if it is only divisible by either $a \cdot f(x)$ or $a \cdot 1$ for some $a \in GF(p^m)$. The elements of the $GF(p^m)$ are simply the polynomials of the degree less than m, and the operations of addition, subtraction, multiplication and division over polynomials are done modulo p. A polynomial can be seen as the vector of the coefficients, i.e. the vector $a = (a_0, a_1, \dots, a_{m-1})$ is associated to the polynomial $A(x) = \sum_{i=0}^{m-1} a_i x^i$, where $a_i \in GF(p)$. This is the so-called *polynomial basis representation*.

If we consider polynomial basis representation of the binary fields $GF(2^m)$, addition and subtraction are done as the simple bitwise XOR of the field polynomials (because arithmetic is done modulo 2). Multiplication is performed by computing $a(x)b(x) \bmod f(x)$, where $f(x)$ is the field irreducible polynomial.

A *normal basis* is formed by choosing an element $\alpha \in GF(p^m)$ such that m elements of the set $(\alpha, \alpha^p, \alpha^{p^2}, ..., \alpha^{p^{m-1}})$ are linearly independent, i.e. the degrees of the type α^{p^i}, for $i = 0, 1, ..., m-1$ are all different. The existence of such element is guaranteed for all fields and for every value of p and m.

In normal bases, every element $x \in GF(p^m)$ is denoted as $x = \sum_{i=0}^{m-1} x_i \alpha^{p^i}$, where x_i are the elements of $GF(p)$.

If we restrict our consideration to the case of $p = 2$, i.e. to binary fields $GF(2^m)$, which are most frequently used in applications, the squaring operation in normal bases becomes practically free, because for $x = (x_0, x_1, ..., x_{m-1})$,
$x^2 = (x_{m-1}, x_0, ..., x_{m-2})$ which is the simple bit rotation.

Under certain conditions of m, an optimal normal basis [51] can be found. For this basis, for every $i, j = 0, 1, \dots, m-1$ exist k_i, k_j such that

$$\alpha^{2^i+2^j} = \alpha^{2^{k_i}} + \alpha^{2^{k_j}}$$

and the field multiplication becomes much simpler.

Example: Binary finite field $GF(2^4)$ The elements of the field $GF(2^4)$ are the binary polynomials: $0, 1, x, x+1, x^2, x^2+1, x^2+x+1, x^3, x^3+1, x^3+x, x^3+x+1, x^3+x^2, x^3+x^2+1, x^3+x^2+x, x^3+x^2+x+1$.
Some examples of arithmetic operations in $GF(2^4)$ with the irreducible polynomial $f(x) = x^4+x+1$ are:
Addition: $(x^3+x^2+1)+(x^2+x+1) = x^3+x$
Subtraction: $(x^3+x^2+x)-(x^2+1) = x^3+x$
Multiplication:$(x^3+x^2+x)\cdot(x^2+1) = (x^5+x+1) \bmod (x^4+x+1) = x^2+1$
Inversion: $(x^3+x^2+x)^{-1} = x^2$ since $(x^3+x^2+x)\cdot x^2 \bmod (x^4+x+1) = 1$.

3.1.2 Software Implementation of Finite Field Operations in Polynomial Basis

In all implementations in practical use, ECC requires the use of arithmetic algorithms that operate on operands that are much larger than the microprocessor's word size (i.e., some hundred bit long operands on a 32-bit architecture). Handling these operands, and performing the required mathematical functions upon them in an efficient manner requires the use of multi-precision $GF(2^m)$ arithmetic. Each m-bit operand is stored on a w-bit processing architecture (where w is 8, 16, 32, 64) as a $t = \lceil m/w \rceil$ word array (m-bit arithmetic is reduced to w-bit digit arithmetic which can be performed using the processor's ALU).
The field element $A = (a_0, a_1, ..., a_{m-1})$ is stored in memory as an array of t w-bit words: $A = (A[t-1]A[t-2], ..., A[1]A[0])$, where the rightmost bit of $A[0]$ is a_0, and the leftmost $s = wt - m$ bits of $A[t-1]$ are unused (set to 0).
In the following sections, we will present algorithms suitable for word-level binary finite field arithmetic in software.

3.1.2.1 Addition in $GF(2^m)$

The addition/subtraction of the two polynomials is done as bitwise exclusive-or (XOR, $\oplus$), and thus requires t word operations.

Algorithm 1 $GF(2^m)$ addition/subtraction

```
INPUT: two $GF(2^m)$ elements $A = (A[t-1]...A[0])$ and $B = (B[t-1]...B[0])$ of degree max $m-1$
OUTPUT: $C = A + B = (C[t-1]...C[0])$
    for ($i = 0$ to $t-1$)
        $C[i] = A[i] \oplus B[i]$
    end for
```

3.1.2.2 Multiplication in $GF(2^m)$

The product of two binary field elements can be obtained by first multiplying them as polynomials, which gives as the result a polynomial of the degree less than or equal to $2(m-1)$, and then calculating the rest of the division with the irreducible field polynomial $f(x)$. Fast methods for field multiplication first multiply the two polynomials, and then reduce the result modulo $f(x)$. We will present several efficient polynomial multiplication methods, which were used in this dissertation.

Pencil-and-Paper Polynomial Multiplication

Pencil-and-paper algorithm is the modification of the classical shift-and-add multiplication taken from [36] and adapted for multiplication of binary polynomials.

Algorithm 2 Pencil-and-Paper polynomial multiplication

```
INPUT: binary polynomials $a(x) = (A[t-1]...A[1]A[0])$ and
$b(x) = (B[t-1]...B[1]B[0])$ of the degree max $m-1$
OUTPUT: polynomial product $(C[2t-1]\ldots C[1]C[0]) = A \otimes B$
    for $(i = 0$ to $2(t-1))$
        $C[i] \leftarrow 0$
    end for
    for $(i = 0$ to $t-1)$
        for $(j = 0$ to $t-1)$
            $P, Q \leftarrow A[i] \otimes B[j]$
            $C[i+j] \leftarrow C[i+j] \oplus P$
            $C[i+j+1] \leftarrow C[i+j+1] \oplus Q$
        end  for
    end for
```

In this method, $\otimes$ denotes the word-level polynomial multiplication, i.e. the routine that polynomially multiplies individual words $A[i]$ and $B[j]$ of $a(x) = (A[t-1]...A[1]A[0])$ and $b(x) = (B[t-1]...B[1]B[0])$ to produce a two-word result. A simple way to produce the word-level polynomial product is to scan the coefficients of $B[j]$ from b_{w-1} to b_0 and to add the partial product $A[i]b_k$ to the running sum. The partial product is either 0 (if $b_k = 0$) or the multiplicand word $A[i]$ (if $b_k = 1$). After each partial product addition, the product must be multiplied by x (i.e. shifted one bit to the left) to align it for the next partial product.

Algorithm 3 A word level shift-and-add polynomial multiplication $\otimes$

```
INPUT: single $w$-bit words $a = A[i]$ and $b = B[j]$ of the polynomials $a(x) = (A[t-1]...A[1]A[0])$ and $B(x) = (B[t-1]...B[1]B[0])$, where $w$ is the processor word length. $b_k$ is the $k$-th bit of $B[j]$.
OUTPUT: two words $C1, C2$ of the polynomial product $A[i] \otimes B[j]$
    $(C1, C2) \leftarrow 0$
    for ($k = w - 1$ downto 0)
        if $b_k \neq 0$ then $(C1, C2) \leftarrow (C1, C2) + A[i]$
        $(C1, C2) \leftarrow x(C1, C2)$
    end for
```

Comb Polynomial Multiplication

In [15] Comba proposed a means of accelerating the implementation of multiplication by minimizing the number of external memory references that are required during the course of execution. Comba's method proposes to eliminate the write-back operation by changing the order of partial product generation/accumulation such that each word of the result is computed in its entirety sequentially, starting with the least significant word. Hence, only the values $A[i]$ and $B[j]$ need to be read from memory. Original Comba's method can be improved significantly at the expense of some storage overhead, by first computing $u(x)b(x)$ for all polynomials $u(x)$ of degree less than W (window length).

Algorithm 4 Left-to-right comb method with windows of length W

```
INPUT: Binary polynomials $a(x) = (A[t-1]...A[1]A[0])$ and $b(x) = (B[t-1]...B[1]B[0])$
of degree at most $m-1$, window length $W$, processor word length $w$
OUTPUT: $c(x) = a(x)b(x)$
    Precomputation
    Compute $B_u = u(x)b(x)$ for all $u(x)$ of degree at most $W$
    $C \leftarrow 0$
    **for** $k = w/W - 1$ **downto** $0$
        **for** $j = 0$ **to** $t-1$
            Let $u = (u_{W-1}...u_1u_0)$ where $u_i$ is bit $(Wk+i)$ of $A[j]$
            $C[j] \leftarrow C[j] + B_u$
        **end for**
        **if** $k \neq 0, C \leftarrow Cx^W$
    **end for**
```

Karatsuba-Ofman Polynomial Multiplication

The Karatsuba-Ofman Algorithm [35] uses a recursive divide-and-conquer approach for multiplying two polynomials that reduces the number of single-precision multiplications that must be performed by replacing a multiplication with several additions (given that addition can usually be performed faster than multiplication on conventional microprocessors).

To multiply two binary polynomials $a(x)$ and $b(x)$ of degree at most $m-1$, we first split up each of them into two polynomials of degree at most $(m/2)-1$ (in case m is odd, the polynomials are first prepended with zeros):
$A(x) = A_1(x)X + A_0(x), B(x) = B_1(x)X + B_0(x)$, where $X = x^{m/2}$.
Then

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

The product can be derived from three products of degree $(m/2-1)$ (i.e., a m-bit multiplication is performed by two $m/2$-bit multiplications, one $(m/2+1)$-bit multiplication to handle the product-of-sum term, and several multi-precision additions). The resulting $m/2$ and $(m/2+1)$ bit multiplications can in turn be computed using Karatsuba's algorithm again, leading to a recursive multiplication algorithm. In practice, the number of levels of recursion that are used will ultimately be dictated by the amount of overhead associated with a particular implementation of the algorithm, and the relative performance of the multiplication and addition operations (i.e., the slower the multiplier, the more recursion that should be used). For example, to apply Karatsuba-Ofman algorithm to 192-bit binary polynomials with processor word length $w = 32$, we can apply the following recursions:

$$192 \rightarrow 96 + 96 \rightarrow 32, 32, 32 + 32, 32, 32 \text{ or}$$
$$192 \rightarrow 64 + 64 + 64 \rightarrow 32, 32 + 32, 32 + 32, 32$$

Montgomery Multiplication

Instead of computing $a \cdot b$ in $GF(2^m)$, Koç and Acar in [40] proposed to compute $a \cdot b \cdot r^{-1}$ in $GF(2^m)$, where r is a special fixed element of $GF(2^m)$. A similar idea was proposed by Montgomery in [50] for modular multiplication of integers. The selection of $r(x) = x^m$ turns out to be very useful in obtaining fast software implementations. Thus, r is the element of the field, represented by the polynomial $r(x) \bmod f(x)$, i.e., if $f = (f_m f_{m-1} ... f_1 f_0)$, then $r = (f_{m-1} ... f_1 f_0)$.
The Montgomery multiplication method requires that $r(x)$ and $f(x)$ are relatively prime, i.e., $gcd(r(x), f(x)) = 1$. For this assumption to hold, it suffices that $f(x)$ be not divisible by x. Since $f(x)$ is an irreducible polynomial over the field $GF(2)$, this will always be the case. Since $r(x)$ and $f(x)$ are relatively prime, there exist two polynomials $r^{-1}(x)$ and $f'(x)$ with the property that

$$r(x)r^{-1}(x) + f'(x)f(x) = 1$$

where $r^{-1}(x)$ is the inverse of $r(x)$ modulo $f(x)$. The polynomials $r^{-1}(x)$ and $f'(x)$ can be computed using the Extended Euclidean algorithm.
The word-level algorithm for computing the Montgomery product requires the computation of the w-length polynomial $F_0(x)$ instead of the entire polynomial $F(x)$ which is of length $k = tw$. The efficient inversion algorithm is based on the observation that the polynomial $F_0(x)$ and its inverse satisfy $F_0(x)F_0'(x) = 1 \bmod x_i$ for $i = 1, 2, \ldots, w$.

In order to compute $F_0'(x)$, we start with $F_0'(x) = 1$, and proceed as:

Algorithm 5 Inversion Algorithm

```
INPUT: Processor word length $w, F_0(x)$
OUTPUT: $F_0'(x) = F_0^{-1}(x) \bmod x^w$
    $F_0'(x) = 1$
    **for** $i = 2$ **to** $w$
        $t(x) = F_0(x)F_0'(x) \bmod x^i$
        **if** $t(x) \neq 1$ **then** $F_0'(x) \leftarrow F_0'(x) + x^{i-1}$
    **end for**
```

The Montgomery multiplication of $a(x)$ and $b(x)$ is defined as the product

$$c(x) = a(x) \cdot b(x) \cdot r^{-1}(x) \bmod f(x)$$

The Montgomery multiplication is particularly useful for obtaining fast software implementation of the exponentiation over the field $GF(2^m)$. However, it can also be used for polynomial multiplication $a(x) \cdot \underline{(}x) \bmod f(x)$; note that in this algorithm multiplication is interleaved with reduction, so there is no need to reduce the obtained product afterwards. However, to obtain $c(x) = a(x) \cdot b(x) \bmod f(x)$, the we have to normalize the product, i.e. to use Montgomery multiplication again: $(a(x) \cdot b(x) \cdot r^{-1}(x)) \cdot r(x)^2) \cdot r^{-1}(x)) \bmod f(x) = a(x) \cdot \underline{(}x) \bmod f(x)$.
The following is the algorithm for word-level Montgomery multiplication; $\oplus$ denotes the XOR operation over two words, while $\otimes$ denotes polynomial multiplication of two words of $a(x)$ and $b(x)$ (see Algorithm 3 in Section 3.1.2).

Algorithm 6 Word-level Montgomery method for polynomial multiplication

```
INPUT: Binary polynomials $a(x) = (A[t-1] \dots A[0])$, $b(x) = (B[t-1] \dots B[0])$,
the irreducible polynomial $f(x) = (F[t-1] \dots F[0])$, $F_0'$
OUTPUT: $c(x) = a(x)b(x)x^{-m} \bmod f(x)$
    for $i = 0$ to $t$
        $C[i] \leftarrow 0$
    end for
    $c(x) = 0$
    for $i = 0$ to $t-1$
        for $j = 0$ to $t-1$
            $(H, L) \leftarrow A[j] \otimes B[i]$
            $C[j] \leftarrow C[j] \oplus L$
            $C[j+1] \leftarrow C[j+1] \oplus H$
        end for
        $(H, M) \leftarrow C[0] \otimes F'[0]$
        $(P, L) \leftarrow M \otimes F[0]$
        for $j = 1$ to $t-1$
            $(H, L) \leftarrow M \otimes N[j]$
            $C[j-1] \leftarrow C[j] \oplus L \oplus P$
            $P \leftarrow H$
        end for
        $C[t-1] \leftarrow C[t] \oplus P \oplus M$
        $C[t] \leftarrow 0$
    end for
```

3.1.2.3 Squaring in $GF(2^m)$

Polynomial squaring is much faster that the polynomial multiplication, since it can be obtained by inserting zero bit between each two consecutive bits of the binary representation of $a(x)$:

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

The speed of squaring can be additionally improved if precomputed look-up table is used.

3.1.2.4 Reduction in $GF(2^m)$

Once the polynomial product has been formed, it is reduced by exploiting the fact that, for the field irreducible polynomial $f(x)$

$$f(x) = x^m + \sum_{i=1}^{m-1} f_i x^i + 1 \Rightarrow x^m \equiv \sum_{i=1}^{m-1} f_i x^i + 1 (\bmod f(x))$$

This identity can be used to reduce the $(2m-1)$-bit product via repeated substitution and accumulation until bits $(2m-2)$ to m are all zero, indicating that the result has been properly reduced.

If $f(x)$ is a trinomial or a pentanomial with middle terms close to each other (which is the case for all irreducible polynomials recommended by NIST DSA Standard [53] used in this dissertation), then reduction of $c(x) \bmod f(x)$ can be efficiently performed using the algorithm presented in [11] for trinomials, adapted to work on processor word-level. This algorithm operates on $c(x)$ "in place", eliminating the need for extra storage for the result.
For example, suppose $m = 163$ and $w = 32$ (so $t = 6$), and consider reducing the word $C[9]$ of $c(x)$ modulo $f(x) = x^{163} + x^7 + x^6 + x^3 + 1$. The word $C[9]$ represents the polynomial $c_{319}x^{319} + c_{289}x^{289} + c_{288}x^{288}$. We have:

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

By adding the four columns on the right side of the above congruences, we see that reduction of $C[9]$ can be performed by adding $C[9]$ four times to C, with the rightmost bit of $C[9]$ added to bits 132, 131, 128 and 125 of C.

3.1.2.5 Inversion in $GF(2^m)$

The inverse of an element $a(x) \in GF(2^m)$ is the unique element $g(x) \in GF(2^m)$ such that $a(x)g(x) \equiv 1(\bmod f(x))$. This inverse element is denoted as $a(x)^{-1}$ (the reduction polynomial $f(x)$ is understood from context). Inversion over $GF(2^m)$ can be performed by exploiting the cyclic nature of $GF(2^m)$ and Fermat's theorem to compute the inverse $a(x)^{-1}$ using the formula:

$$a(x)^{-1} = a(x)^{2^m-2} \quad \forall a(x) \in GF(2^m)$$

We used two algorithms for efficiently calculating inverses in $GF(2^m)$: *Extended Euclidean Algorithm* and *Almost Inverse Algorithm*.
The Almost Inverse Algorithm is due to Schroeppel et. al. [64] and defers the numerous single-bit shifting operations present in the Extended Euclidean Algorithm until the end so that they can be performed all at once in a much more efficient manner. The output of Schroeppel's algorithm is the almost inverse, $2^k A^{-1}$, and the degree, k, of the scaling constant that must be divided out to recover A^{-1}.
The algorithms will be presented here as implemented by IEEE-P1363 Standard [31].

Algorithm 7 Extended Euclidean Algorithm

```
INPUT: A non-zero binary polynomial $a$ of degree at most $m-1$
OUTPUT: $a^{-1} \bmod f$
    $u \leftarrow a, v \leftarrow f$
    $g_1 \leftarrow 1, g_2 \leftarrow 0$
    **while** $u \neq 1$
        $j \leftarrow deg(u) - deg(v)$
        **if** $j < 0$ **then** $u \leftrightarrow v, g_1 \leftrightarrow g_2, j \leftarrow -j$
        $u \leftarrow u + z^j v$
        $g_1 \leftarrow g_1 + z^j g_2$
    **end while**
```

Algorithm 8 Almost Inverse Algorithm

```
INPUT: A non-zero binary polynomial $a$ of degree at most $m-1$
OUTPUT: $a^{-1} \bmod f$
  $u \leftarrow a, v \leftarrow f$
  $g_1 \leftarrow 1, g_2 \leftarrow 0, k \leftarrow 0$
  **while** $u \neq 1$ and $v \neq 1$
    **while** $z$ divides $u$
      $u \leftarrow u/z, g_2 \leftarrow zg_2, k \leftarrow k+1$
    **end while**
    **while** $z$ divides $v$
      $v \leftarrow v/z, g_1 \leftarrow zg_1, k \leftarrow k+1$
    **end while**
    **if** $deg(u) > deg(v)$ **then** $u \leftarrow u+v, g_1 \leftarrow g_1+g_2$
    **else** $v \leftarrow v+u, g_2 \leftarrow g_1+g_2$
    **if** $u = 1$ **then** $g \leftarrow g_1$
    **else** $g \leftarrow g_2$
  **end while**
Return $z^{-k}g \bmod f$
```

3.2 Coordinate Representation

If field inversion is much more expensive than multiplication, it is advantageous to represent points using projective coordinates. There are different types of projective coordinates in use:

- *Standard projective coordinates*: The projective point $(X:Y:Z), Z \neq 0$ corresponds to the affine point $(X/Z^2, Y/Z^3)$. The point at infinity O corresponds to $(0:1:0)$.
- *Jacobian projective coordinates*: The projective point $(X:Y:Z), Z \neq 0$ corresponds to the affine point $(X/Z, Y/Z)$. The point at infinity O corresponds to $(1:1:0)$.

- *López-Dahab projective coordinates*: The projective point $(X : Y : Z), Z \neq 0$ corresponds to the affine point $(X/Z, Y/Z^2)$. The point at infinity O corresponds to $(1 : 0 : 0)$.

In all three cases the negative of $(X : Y : Z)$ is $(X : X + Y : Z)$.

Formulas that do not involve field inversions in adding and doubling points in projective coordinates can be derived by first converting the points to affine coordinates, then using formulas 2.7 to add the affine points, and finally clearing denominators. Often in use is the *mixed addition*, when the two points are given in different coordinate systems.
As an example, we give formulas for mixed addition and doubling of points in López-Dahab projective coordinates [44].
Formulas for computing the double $(X_3 : Y_3 : Z_3)$ of the point $(X_1 : Y_1 : Z_1)$ are:

$$Z_3 \leftarrow X_1^2 Z_1^2, X_3 \leftarrow X_1^4 + bZ_1^4, Y_3 \leftarrow bZ_1^4 Z_3 + X_3(aZ_3 + Y_1^2 + bZ_1^4)$$

The sum $(X_3 : Y_3 : Z_3)$ of $(X_1 : Y_1 : Z_1)$ and $(X_2 : Y_2 : 1)$ is

$$\begin{gathered} A \leftarrow Y_2 Z_1^2 + Y_1, B \leftarrow X_2 Z_1 + X_1, C \leftarrow Z_1 B, D \leftarrow B^2(C + aZ_1^2), \\ Z_3 \leftarrow C^2, E \leftarrow AC, X_3 \leftarrow A^2 + D + E, F \leftarrow X_3 + X_2 Z_3, \\ G \leftarrow (X_2 + Y_2) Z_3^2, Y_3 \leftarrow (E + Z_3)F + G \end{gathered}$$

The field operation count for different types of coordinates is listed in Table 3.2.

Coordinates	General Addition	Mixed Addition	Doubling
Affine	2M, 1S, 1I	n/a	2M, 1S, 1I
Standard Projective	17M, 1S	13M, 1S	7M, 3S
Jacobian Projective	15M, 5S	11M, 4S	5M, 5S
López-Dahab Projective	14M, 6S	9M, 4S	3M, 5S

Table 3.1. Cost of point addition and doubling in $GF(2^m)$ (M-multiplications, S-squarings, I-inversions).

3.3 Implementation of Scalar Point Multiplication

This section considers methods for computing $[k]P$, where P is an elliptic curve point, and k is an integer. We will describe some methods that we used in our software implementation of ECC; an excellent survey of point multiplication methods can be found in [28].

3.3.1 Double-and-Add Algorithm

This is an adjustment of the classical algorithm quoted by Knuth [36].

Algorithm 9 Double-and-Add kP

```
INPUT: A point $P$ and an integer $k = (k_{t-1}, \ldots, k_1, k_0)_2$
OUTPUT: $Q = [k]P$
  $Q = O$
  for $i = t - 1$ downto 0
    $Q \leftarrow 2[Q]$
    if $k_i = 1$ then $Q \leftarrow Q + P$
  end for
```

3.3.2 Signed Digits

The integer k is expressed as $\sum_{i=0}^{t-1} k_i 2^i$, where $k_i \in 0, \pm 1$, $k_{t-1} \neq 0$, and no consecutive digits of k are nonzero. This representation is unique and is called the *NAF* (non-adjacent form). The binary-NAF conversion is performed using a simple algorithm, which is reported in [11]. The product of a point and an integer is given by following algorithm:

Algorithm 10 Signed Digits [k]P

```
INPUT: A point $P$ and an integer $k$
OUTPUT: $Q = [k]P$
  $Q = 0$
  for $i = t - 1$ downto $0$
    $Q \leftarrow [2]Q$
    if $k_i = 1$ then $Q \leftarrow Q + P$
    else if $k_i = -1$ then $Q \leftarrow Q - P$
  end for
```

The calculation of $Q - P$ has practically the same cost as $Q + P$ (in fact, $QP = Q + (-P)$ and $-P$ is found practically for free), but the advantage is the number of non-zero digits: in NAF, the average is $t/3$, compared to $t/2$ in the binary representation (where t is the binary length of k).

3.3.3 Modified Montgomery Method

This algorithm is based on idea of Montgomery [50] and is due to López and Dahab [43].

Let $Q_1 = (x_1, y_1)$ and $Q_2 = (x_2, y_2)$. Let $Q_1 + Q_2 = (x_3, y_3)$ and $Q_1 - Q_2 = (x_4, y_4)$. Using the addition formulas 2.7 it can be verified that

$$x_3 = x_4 + \frac{x_2}{x_1+x_2} + (\frac{x_2}{x_1+x_2})^2$$

The x coordinate of $Q_1 + Q_2$ can be computed from the x coordinates of Q_1, Q_2, and $Q_1 - Q_2$.

Algorithm 11 Modified Montgomery [k]P

```
INPUT: A point $P = (x, y)$ and an integer $k = (k_{t-1}...k_1k_0)_2$ with $k_{t-1} = 1$
OUTPUT: $Q = [k]P = (x_3, y_3)$
$X_1 \leftarrow x$, $Z_1 \leftarrow 1$, $X_2 \leftarrow x^4 + b$, $Z_2 \leftarrow x^2$
for $i = t - 2$ downto $0$
    if $k_i = 1$ then
        $T \leftarrow Z_1, Z_1 \leftarrow (X_1Z_2 + X_2Z_1)^2, X_1 \leftarrow xZ_1 + X_1X_2TZ_2$
        $T \leftarrow X_2, X_2 \leftarrow X_2^4 + bZ_2^4, Z_2 \leftarrow T^2Z_2^2$
    else
        $T \leftarrow Z_2, Z_2 \leftarrow (X_1Z_2 + X_2Z_1)^2, X_2 \leftarrow xZ_2 + X_1X_2TZ_1$
        $T \leftarrow X_1, X_1 \leftarrow X_1^4 + bZ_1^4, Z_1 \leftarrow T^2Z_1^2$
    $x_3 \leftarrow X_1/Z_1$
    $y_3 \leftarrow (x + X_1/Z_1)[(X_1 + xZ_1)(X_2 + xZ2) + (x^2 + y)Z_1Z_2](xZ_1Z_2)^{-1} + y$
end for
```

3.3.4 Fixed-base Comb Method

If the point P is fixed and some storage is available, then the point multiplication can be accelerated by precomputing some data. In the fixed-base comb method, the binary representation of k is divided into W bit strings (where W is the so-called window size) of the equal length d (k is padded with zeros if necessary), so that

$$k = K^{W-1} \parallel ... \parallel K^1 \parallel K^0$$

The bit strings K^i are then written as rows, whose columns are processed one at a time. To accelerate the computation, the points

$$[a_{W-1}, ..., a_2, a_1, a_0]P = a_{W-1}2^{(W-1)d}P + ... + a_2 2^{2d}P + a_1 2^d P + a_0 P$$

are precomputed for all possible bit strings $(a_{W-1}, ..., a_0)$.

Algorithm 12 Fixed-base Comb kP

```
INPUT: A point $P$, window width $w$, an integer $k = (k_{t-1}...k_1k_0)_2$ with $k_{t-1} = 1$,
$d = \lceil t/w \rceil$.
OUTPUT: $Q = [k]P$
    Compute $[a_{W-1}, ..., a_2, a_1, a_0]P$ for all bit strings $(a_{W-1}, ..., a_0)$.
    Write $k = K^{W-1} \parallel ... \parallel K^1 \parallel K^0$, where each $K^j$ is a bit string of length $d$. Let
    $K_i^j$ denote the $i$-th bit of $K^j$.
$Q \leftarrow O$
for $i = d - 1$ downto 0
    $Q \leftarrow 2Q$
    $Q \leftarrow Q + [K_i^{W-1}, ..., K_i^1, K_i^0]$
end for
```

3.3.5 Window Methods

The exponent is divided into blocks of maximum r bits that do not finish with zero and are not necessary continuous.

Algorithm 13 Window kP

```
INPUT: A point $P$ and an integer $k$
OUTPUT: $Q = [k]P$
  Precalculation
  $P_i = P$, $P_i = 2P$
  for $i = 1$ to $2^{r-1} - 1$
    $P_{2i+1} = P_{2i-1} + P_2$
  end for
  Main cycle
  $j \leftarrow L$
  $Q \leftarrow O$
  while $j \geq 0$
    if $(k_j = 0)$ then $Q \leftarrow [2]Q, j \leftarrow j - 1$
    else
      let $t$ be a minimum integer such that $k_i = 1$ and $j - t + 1 \leq r$
      $h \leftarrow (k_j, k_{j-1}, ..., k_t)$
      $Q \leftarrow [2j - t + 1]Q + P_t$
      $j \leftarrow t - 1$
  end while
```

Chapter

4

Benchmarks, Methodology and Related Work

4.1 Benchmark Set

In a typical portable wireless systems, cryptographic functions are performed in software due to the ease of implementation and flexibility that accompanies the use of software. Many existing software implementations of public-key cryptography can be found both in academic and commercial domains. Numerous companies such as Certicom [14] provide complete software implementations that can be purchased for use in commercial applications.

The work described in this dissertation began with searching for software-based solution which implements arithmetic functions required by the various ECC algorithms. We used and modified an available open-source C library, Miracl [49]. This library offered all basic cryptographic functions necessary for developing our ECC benchmark set. It consists of over 100 routines that cover all aspects of multiprecision and finite field arithmetic. Inside the library, a data type called *big* for storing multiprecision integers is defined. It is in the form of a pointer to a fixed length array of digits, with sign and length information encoded in a separate 32-

bit integer. Algorithms used to implement arithmetic on the big data type are taken from [36].

The library was subsequently expanded to contain other $GF(2^m)$ multiplication methods (Pencil-and-Paper, Montgomery and Comb) and point multiplication algorithms (Modified Montgomery and Fixed-base Comb). The summary of implemented EC and finite field methods is presented in Table 4.1.

Methods Implemented within Extended Miracl Library	
$GF(2^m)$ Multiplication	Karatsuba-Ofman, Pencil-and-Paper, Comb, Montgomery
$GF(2^m)$ Reduction	Reduction optimized for NIST irreducible polynomials
$GF(2^m)$ Inversion	Extended Euclidean Algorithm, Almost Inverse Algorithm
Coordinates	Affine, Projective (Jacobian, López-Dahab)
$[k]P$ Methods	Signed window NAF, Modified Montgomery, Fixed-base Comb

Table 4.1. Methods implemented within extended Miracl library.

The use of elliptic curve cryptography also involves choosing a finite field, a specific curve on it, and a base point on the chosen curve. The binary finite fields and elliptic curves we used in tests were chosen according to NIST standard [53]. This standard recommend use of certain curves with strong security properties to ease the interoperability between different implementations of security protocols. For binary polynomial fields, the curves were recommended for key sizes of 163, 233, 283, 409, and 571 bits. Apart from the field size, parameter files in use with our benchmarks specify parameters for initializing the curve, setting the base point on a curve, and setting the irreducible polynomial (trinomial or pentanomial) for reduction of polynomial product. We conducted experiments for the first three fields only, which cover the security requests of nowadays and next future applications.

When referring to benchmarks, the abbreviation b denotes an elliptic curve over binary field, while three-digit number indicates the size of the field; for example, $b163$ denotes the use of $GF(2^{163})$ binary field.

An example of $GF(2^m)$ parameter file for $m = 283$ is given in Table 4.2.

the degrees of the irreducible polynomial	$283 \quad 12 \quad 7 \quad 5 \quad 0$
the curve term b (hex)	27b680ac8b8596da5a4af8a19a0303fca97f d7645309fa2a581485af6263e313b79a2f5
the base point coordinate x (hex)	5f939258db7dd90e1934f8c70b0dfec2eed2 5b8557eac9c80e2e198f8cdbecd86b12053
the base point coordinate y (hex)	3676854fe24141cb98fe6d4b20d02b4516ff 702350eddb0826779c813f0df45be8112f4
the base point order	690174634679056378743475586227025 55839812737345013555379383634485463

Table 4.2. An example of $GF(2^m)$ parameter file for $m = 283$.

We set up a series of kernel benchmarks to cover the cryptographic algorithms described in Chapter 2. The benchmark set description is given in Table 4.3.

Elliptic-curve benchmarks in graphs and tables have the following notation:

$$< benchmark > . < b >< threedigitnumber >$$

For example, *ecdh.b163* denotes Elliptic Curve Diffie Hellman key exchange over $GF(2^{163})$.

4.2 Experimental Evaluation

The performance evaluation of our ECC benchmark set is done using a modified version of sim-profile and sim-outorder simulators of the SimpleScalar toolset [65] for the ARM target. The *sim-profile* tool was used for getting the instruction mix of the executed benchmarks, while the *sim-outorder* tool served for a detailed timing simulation of the modeled target. Simulation is execution driven, and accounts for speculative execution, branch prediction, cache misses, etc.
When simulating ”ideal” memory, processor was configured to have the perfect memory: non-existing cache and latency of main memory access of 1 cycle (which

Benchmark Set		
Benchmark acronym	Benchmark Name	Description
ECDH	EC Diffie Hellman Key Exchange	Based on a prime suitable for Diffie-Hellman algorithm calculates a shared key
ECDSIGN	EC Digital Signature Generation	Calculates the message digest of a file given as argument using SHA-1 algorithm, signs the message using the private key and writes the signature into a file
ECDSVER	EC Digital Signature Verification	Calculates the message digest of a file given as argument using SHA-1 algorithm, then reads and verifies the signature using sender's public-key
ECELGENC	EC ElGamal Encryption	Encrypts a point on the curve using ElGamal algorithm
ECELGDEC	EC ElGamal Decryption	Decrypts a point on the curve using ElGamal algorithm

Table 4.3. Benchmark set description.

corresponds to cache hit latency). The sim-outorder tool was modified to add cycle level function profiling (i.e. to produce exact number of execution cycles for every procedure in the code, based on the output of gcc-objdump). Some unimplemented system calls for ARM target are also added, although they were not critical for the execution of the benchmark. Also, the sim-outorder tool was modified to allow for executing the newly added instructions.

We used gcc cross-compilers for ARM instruction set included with SimpleScalar package [65]. Gcc cross-assembler was modified to recognize newly added instructions. All benchmarks were compiled with -O2 level of optimization. We did not use -O3 flag because it could cause function inlining and loop unrolling, thus augmenting code size, which is still a concern for embedded systems. The library file containing elliptic curve primitives was compiled with -S option to produce assembly code, which was then modified to include new instructions.

4.3 The Intel XScale Processor

The simulated processor configuration is modeled after Intel XScale architecture [34] adopted by major PDA manufacturers like Toshiba, Fujitsu and HP.
The Intel XScale microarchitecture is based on a core which compliant with ARM ISA version 5TE. The microarchitecture surrounds the core with instruction and data memory management units, instruction, data, and mini-data caches, MMUs, BTB, and MAC coprocessor. This RISC core can be integrated with peripherals to develop smaller, more cost-effective handheld devices with long battery life, with the performance to run rich multimedia applications.
The pipeline of Intel XScale is composed of integer, multiply-accumulate (MAC), and memory pipes. The integer pipe has seven stages:

- fetch 1 (F1);
- fetch 2 (F2);
- decode (ID);
- register file/shift (RF);
- ALU execute (X1);
- state execute (X2);
- integer writeback (WB).

The MAC pipe has six to nine stages that use the first four stages of the integer pipe (F1 through RF) and then finish with MAC stages M1, M2, M3, M4, and data cache writeback.
In certain situations, the pipeline may need to be stalled because of register dependencies between instructions. Only the destination of MAC operations and memory loads are scoreboarded; the destinations of ALU instructions are not scoreboarded. The Intel XScale core pipeline also makes extensive use of bypassing to minimize data hazards. Bypassing allows results forwarding from multiple sources, eliminating the need to stall the pipeline. The Intel XScale pipeline issues a single instruction per clock cycle. Instruction execution begins at the F1 pipestage and completes at the WB pipestage. Although a single instruction may be issued per clock cycle,

all three pipelines (MAC, memory, and main execution) may be processing instructions simultaneously. If there are no data hazards, then each instruction may complete independently of the others. Each pipestage takes a single clock cycle or machine cycle to perform its subtask with the exception of the MAC unit.
The Multiply-Accumulate (MAC) unit executes the multiply and multiply-accumulate instructions supported by the Intel XScale core. The MAC is not truly pipelined, as the processing of a single instruction may require use of the same datapath resources for several cycles before a new instruction can be accepted. No more than two instructions can occupy the MAC pipeline concurrently. When the MAC is processing an instruction, another instruction may not enter the first stage of the multiplier M1 unless the original instruction completes in the next cycle. The MAC can achieve throughput of one multiply per cycle when performing a 16 by 32 bit multiply.
The mini-data cache of Intel XScale can contain frequently changing data streams, which prevents core stalls caused by multicycle accesses to external memory. This is essentially the victim cache [30]. The mini-data cache relieves the data cache of data thrashing caused by frequently changing data streams. The 2KByte mini-data cache is 32-Set/2-way associative, where each set contains 2-ways and each way contains a tag address, a cache line (32 bytes with one parity bit per byte) of data, two dirty bits (one for each of two 16-byte groupings in a line), and a valid bit. The mini-data cache uses a round-robin replacement policy, and it cannot be locked.
The ARM target of the SimpleScalar set supports the ARM7 integer instruction set (which is the subset of 5TE ISA), and the timing of the model has been validated against a Rebel NetWinder Developer workstation [6] by the developers of the simulator. Details of the Intel XScale processor configuration used in simulations are given in Table 4.4.

4.4 Related Work

Among popular books on cryptography are those of Schneier [63] and Menezes [47]. These books describe the basics of private and public-key cryptography, hash and digital signature schemes. A very useful overviews of elliptic curve cryptography is [39].

Fetch queue (instructions)	4
Branch prediction	8k (bimodal, 2Kbyte 4-way BTB)
Fetch and Decode width (instr)	1
Issue width (instr)	1 (in order)
ITLB	32-entry, fully associative
DTLB	32-entry, fully associative
Functional units	1 int ALU, 1 int MUL/DIV
Instruction L1 cache	32 kByte, 32-way
Data L1 cache	32 kByte, 32-way
L1 cache hit latency	1 cycle
L1 cache block size	32 Kbyte
L2 cache	none
Mini-data cache	2 KByte
Memory latency (cycles)	24, 96
Memory bus width (bytes)	4

Table 4.4. Intel XScale processor configuration in SimpleScalar.

The recent popularity of elliptic curve cryptography, and its extensive use of finite fields theory, has led to several recently published works of both software and hardware implementations. In this survey, we will mention only the works regarding ECC over $GF(2^m)$, although extensive literature can also be found on ECC over prime fields $GF(p)$.

J. López and R. Dahab are the authors of many articles describing efficient methods for arithmetic in finite fields and over elliptic curves [42], [43], [44].
Acar [1] in his Ph.D. thesis proposes fast techniques for Montgomery multiplication, with application on binary finite fields, and investigates the design tradeoffs in software implementation of ECC on different platforms. Halbutoğullari [27] in his Ph.D. dissertation proposes fast parallel algorithms for finite field arithmetic.
De Win et al. in [16] describe an implementation of ECDSA, for both $GF(p)$ and $GF(2^m)$, on a 200 MHz Pentium-Pro and make comparisons with other signature algorithms such as RSA and DSA. The elliptic curve code was mainly written in C++ and for $GF(p)$ the same multi-precision routines were called for RSA and

DSA.
An extensive study of efficient methods for elliptic curve arithmetic in binary finite fields for NIST-recommended curves on Intel Pentium II can be found in [28]. This article served us as a reference for our ECC implementation.
In [25], authors give the first estimate of performance improvements that can be expected by adding ECC support in SSL protocol. They demonstrate that the use of ECC cipher suites can offer significant performance benefits to SSL clients and servers, especially as security needs increase. Weimerskirch et. al. in [68] present algorithms that are especially suited to high-performance devices like large-scaled server computers. They show how to perform an efficient field multiplication for operands of arbitrary size, and how to achieve efficient field reduction for dense irreducible polynomials.
There are various articles on implementing elliptic curve cryptography in embedded systems. In [69] authors describe how an elliptic curve cryptosystem can be implemented on very low cost microprocessors with reasonable performance. In [67] authors implemented elliptic curves over binary fields on a Palm OS device (with NIST-recommended random and Koblitz curves over $GF(2^{163})$). Using Koblitz curves they measured the timings of an ECDSA signature verification to be less than 0.9 seconds.
A workload characterization of some public-key and private-key algorithms, including their elliptic curve equivalents for binary polynomial fields is found in [20]. They characterize operations in Diffie-Hellman, Digital Signature, and ElGamal elliptic curve methods, and demonstrate that these algorithms can be implemented efficiently with a very simple processor.

There are many papers that describe hardware implementations of elliptic curve operations. The majority of these papers consider elliptic curves over binary fields. The first reported work is that of Agnew [2], in which a $GF(2^{155})$ field is used in implementing an elliptic curve coprocessor. Their work utilized optimal normal basis over GF(2^{155}) and a high-performance programmable control processor. Rosner's Master's thesis [59] uses composite field arithmetic in polynomial basis representation to implement a complete elliptic curve point multiplication on $GF((2^n)^m)$ with $nm \leq 130$ on Xilinx FPGA XC400 family of devices. One of the best performing implementation of ECC over $GF(2^m)$ is reported in [56]. The architecture is scalable in terms of area and speed, and uses reconfigurable hardware to deliver optimized circuitry for different elliptic curves and finite fields. Also, the proposed architecture is neither restricted to use polynomials on extension degrees of a special form, nor it favors particular fields (i.e. fields for which optimal normal bases exist). Timings are provided for the field $GF(2^{167})$. Okada, Torii, Itoh and Takenaka [54] describe an FPGA implementation for elliptic curves over $GF(2^{163})$. Bednara et al. [10] present a new generic ECC cryptoprocessor architecture that can be adapted to various area/performance constraints and finite field sizes. Hardware designs intended to minimize power consumption were considered in J. Goodman's Ph.D. thesis [21]. Gura et.al. in [26] designed a programmable hardware accelerator to speed up point multiplication for elliptic curves over binary polynomial fields. The accelerator is based on a scalable architecture capable of handling curves of arbitrary field degrees up to $m = 255$. In addition, it delivers optimized performance for a set of commonly used curves through hardwired reduction logic.

The work that served as the inspiration and the starting point for this dissertation is [13]. In this paper, authors explored techniques to improve the performance of eight commonly used symmetric key cipher algorithms and introduced new instructions (for fast substitutions, general permutations, rotates, and modular arithmetic) that improve the efficiency of the analyzed symmetric cryptoalgorithms.
Grossshaedl in [22] investigates the potential of application-specific instruction set extensions for long integer arithmetic. He defines two special instructions that can be executed on an optimized multiply/accumulate unit and that are simple to incorporate into common RISC architectures executing RSA algorithm.

Chapter 5

Instruction Set Extensions for ECC over Binary Finite Fields

The goal of our study was to explore possible instruction set extensions of a typical embedded RISC processor instruction set that are useful for efficient elliptic curve cryptography operations. Instead of measuring the performance on 8-bit or 16-bit microprocessors, which are commonly in use in devices like smart cards, we opted for a 32-bit ARM processor because it is used in many embedded devices such as PDAs and mobile phones, and in the near future it will probably be used in smart cards. When considering instruction set extensions, we had in mind only minimal modifications of the processor's core, so that extending it in this way could be economically more attractive than using dedicated cryptographic coprocessors, which is the usual solution for improving ECC performance.

For the primary analysis of the ECC code, we chose binary finite fields, Karatsuba polynomial multiplication, Almost Inverse Algorithm for field inversion, and fast reduction method for NIST recommended fields (see Section 3.1.2.4). The coordinates used were Jacobian projective (Section 3.2). Most of these methods were already defined in the Miracl library [49].

5.1 Instruction Mix

To determine in which instruction class the possible candidates for instruction set extensions might be found, we firstly analyzed the instruction mix of elliptic curve benchmarks (Figure 5.1) using the sim-profile simulator from the SimpleScalar toolset [65].

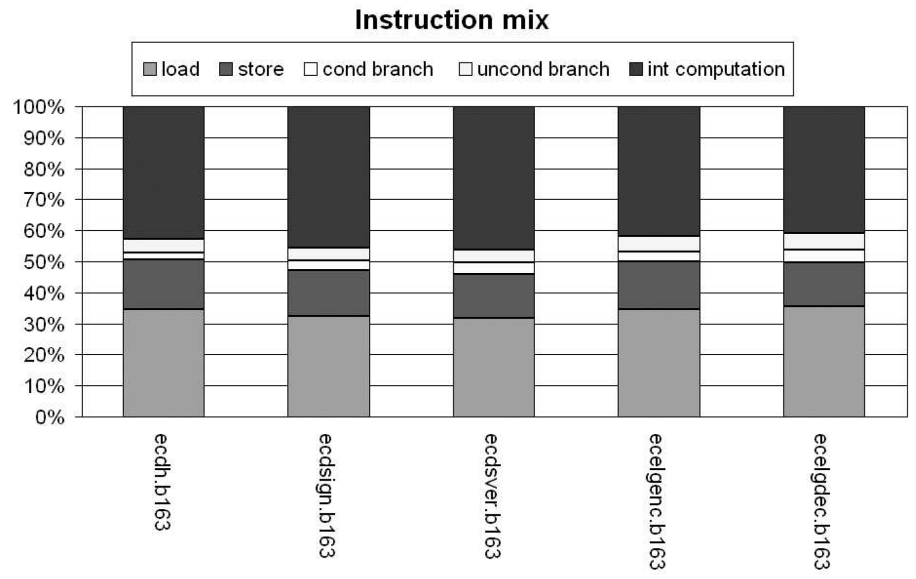

Figure 5.1. Instruction Mix of ECC benchmarks over $GF(2^{163})$. Since the results were essentially identical for different key lengths, we presented only one key length.

The instruction mix shows a very large percentage (approximately 40%) of integer instructions, as well as load and store operations (approximately 50%). A similar distribution is expected since finite field operations translate into a large number of logical operations (XOR, shift etc.), and result in a large number of register-memory transfers to operate on m-bit data (in our case, m can be 163, 233 or 283 and is much larger than 32 bit register width in ARM). The share of load and store operations would have been even higher if some 8-bit or 16-bit processor was used, since it would require even more memory transfers; obviously, 64-bit architectures offer advantages in this sense. To determine the impact of so many memory accesses, we analyzed the cache behavior in further detail; this analysis is shown in Section 6.2.

5.2 Function Cycle Level Profile

In our analysis, we wanted to see if we could substitute some of the functions with some native instructions. To discover which were those candidate functions, we profiled the ECC benchmarks to find out which functions are mostly affecting the execution time. To provide an accurate, cycle-level profiling of the functions, we modified the Simplescalar simulator as described in Section 4.2. This analysis was carried out for all ECC benchmarks that we considered (Figures 5.2, 5.3, 5.4, 5.5, 5.6).
Procedures shown in figures are:

- *karmul2* (recursive Karatsuba algorithm for $GF(2^m)$ polynomial multiplication);
- *mr_bottom4* (the base case in recursive calls of *karmul2* function);
- *mr_mul2* (word-level polynomial multiplication);
- *add2* ($GF(2^m)$ addition, i.e., XOR operation over m-bit polynomials);
- *square2* and *mr_sqr2* ($GF(2^m)$ and word-level squaring respectively);
- *reduce2* ($GF(2^m)$ reduction modulo irreducible polynomial);
- *copy* ($GF(2^m)$ polynomial assignment operation);
- *hashing*, *shs_transform* and *shs_process* (functions used for calculating message digest in digital signature algorithm).

From Figures 5.2 to 5.6, we see that finite field operations, which form the basis of ECC, appear as the most time consuming (all higher-level EC procedures ultimately translate to finite field operations and do not consume much cycles themselves). In particular, the *mr_mul2* procedure, which multiplies two 32-bit binary finite field polynomials and produces a 64-bit product consumes 35% of the total execution time in average for all benchmarks, and reaches 55% for Diffie-Hellman benchmark (5.2). The *mr_mul2* procedure is translated into about 400 dynamic instructions (roughly 500 cycles), which correspond to about 12 instructions per bit to perform 32-bit polynomial multiplication.

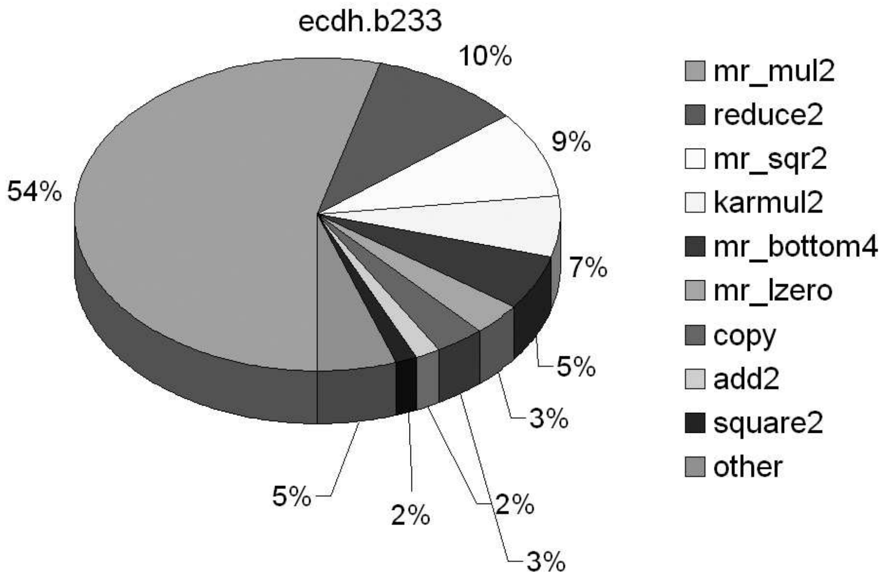

Figure 5.2. Breakdown of execution time (cycle level profiling of program functions) for Diffie-Hellman key exchange. The percentage remains practically invariated when changing key length; thus, in this and all the subsequent figures of this chapter, we presented the function cycle profile only for the key length of 233 bits. Also, in these figures, we have shown only the procedures that take significant part of total execution time.

Elliptic curve Diffie-Hellman shows higher influence by *mr_mul2* than other benchmarks because it is made up exclusively of basic elliptic curve operations, without any other significant computation: in particular, *ecdh* benchmark uses two scalar point multiplications to allow a communication party to form a shared key with the other party.

The second and the third benchmark use ECC to perform digital signature operations (signature and verify), and thus have a non negligible activity in non-EC operations. In digital signature algorithm, more than 60% of the total execution time is spent in:

- reading the message file (18%) and
- calculating its hash (about 45%), i.e. 160-bit digest (functions *shs_transform*, *shs_process*, and *hashing* in Figures 5.3, 5.4).

Figures 5.3 and 5.4 also show that digital signature and verification benchmarks have a very similar distribution of function usage. This is reasonable because ver-

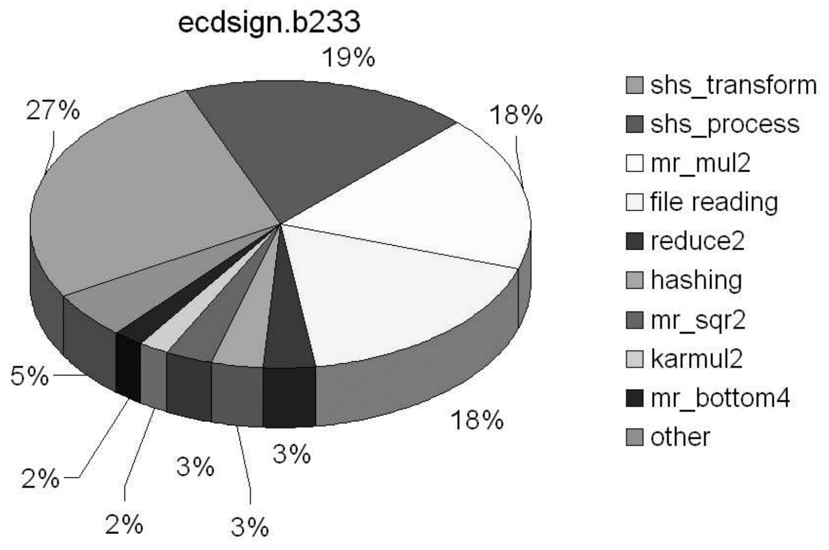

Figure 5.3. Breakdown of execution time (cycle level profiling of program functions) for Digital Signature Generation.

ification procedure is made up of the signature calculation, followed by a comparison of calculated and received digital signatures. However, even in the case of digital signature, the most time-consuming finite field operation is the word-level polynomial multiplication (18%).

ElGamal encryption/decryption (the *ecelgenc* and *ecelgdec* benchmarks in Figures 5.5 and 5.6) essentially traduce into scalar point multiplications, and their function cycle profiles are therefore very similar to Diffie-Hellman key exchange (Figure 5.2). The larger share of *mr_mul2* in total execution time of ElGamal encryption is due to the fact that encryption employs one scalar point multiplication more in respect to decryption in the implementation of our benchmark (see Section 2.6.2).

Overall, the impact of other finite field operations (addition, squaring, reducing with irreducible polynomial and inversion) is at most 20% in *ecdh* benchmark, and is generally less important than finite field multiplication. The fact that inversion in finite field does not take a noticeable part in total execution time is clear, if one takes into consideration that projective coordinates are used (in this case, inversion is traded for more multiplications) (see Section 3.2).

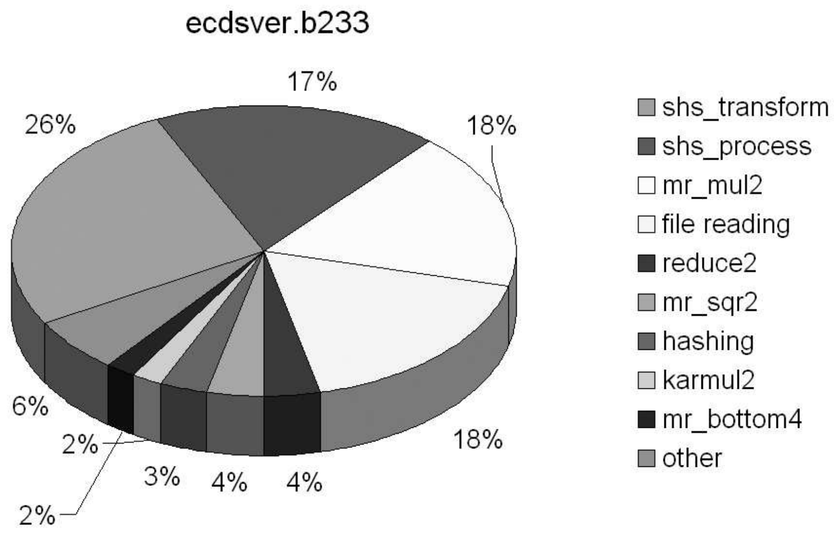

Figure 5.4. Breakdown of execution time (cycle level profiling of program functions) for Digital Signature Verification.

5.3 Instruction Set Extensions

Based on the previous analysis, which showed that the word-level polynomial multiplication is the most time-critical operation for elliptic curve cryptography, we decided to extend ARM instruction set with an instruction for polynomial word-level multiplication in binary finite fields. We called this instruction MULGF (MULtiplication in Galois Fields). The appropriate calls of C procedure for 32-bit polynomial multiplication in software were substituted with a single MULGF instruction.
The MULGF instruction multiplicates two 32-bit polynomials and yields a 64-bit product. The absence of this instruction forced us to use ”shift-and-add” method in software (*mr_mul2* procedure or Algorithm 2 in Chapter 3), which delivers poor performance because of its bit-level operation. Polynomial multiplication is essentially identical to integer multiplication, except that all carries are suppressed; it is best understood on a simple example.

Example of polynomial multiplication on $GF(2^8)$

Suppose that we want to multiply two $GF(2^8)$ polynomials:

$$a(x) = (x^7 + x^6 + x^4 + x^2 + 1) = (11010101)$$
$$b(x) = (x^6 + x^5 + x^4 + x) = (01110010)$$

$(x^7 + x^6 + x^4 + x^2 + 1)(x^6 + x^5 + x^4 + x) = x^{13} + x^{12} + x^{11} + x^8 + x^{12} + x^{11} + x^{10} + x^7 + x^{10} + x^9 + x^8 + x^5 + x^8 + x^7 + x^6 + x^3 + x^6 + x^5 + x^4 + x = = x^{13} + x^9 + x^8 + x^4 + x^3 + x = (0010001100011010)$

```
(11010101)(01110010) =   01110010
                          01110010
                           00000000
                            01110010
                             00000000
                              01110010
                               00000000
                                01110010
                          --------------
                          10001100011010
```

ARM5 Instruction Set implemented in Intel XScale core includes instructions which perform 32 x 32 → 64 long integer multiply [5]. These instructions, called UMULL (unsigned long multiplication) and SMULL (signed long multiplication) multiply the values of two registers and store the result in the third and fourth register specified in the instruction. The MULGF instruction can be implemented in the same format:

$$MULGF < RdLo >, < RdHi >, < Rm >, < Rs >$$

where
$< RdLo >$ stores the lower 32 bits of the result

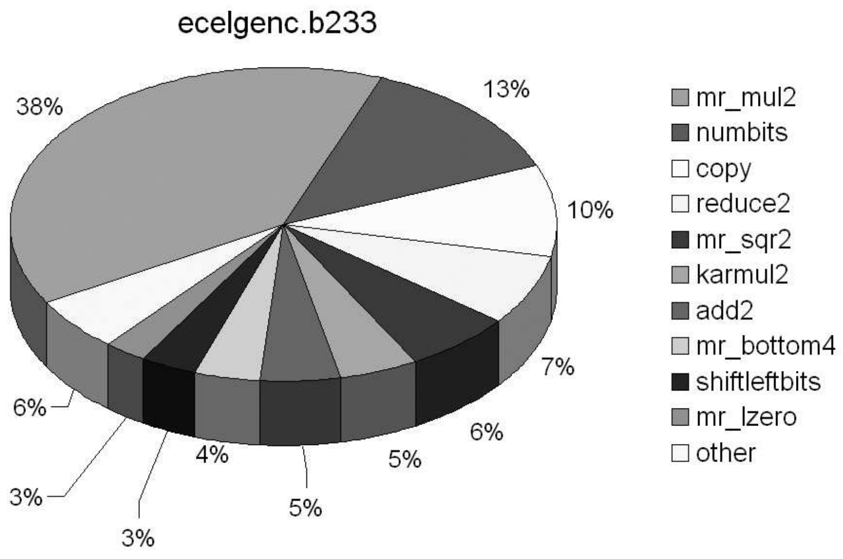

Figure 5.5. Breakdown of execution time (cycle level profiling of program functions) for ElGamal Encryption.

$< RdHi >$ stores the higher 32 bits of the result

$< Rm >$ holds the value to be "polynomially" multiplied with $< Rs >$

$< Rs >$ holds the value to be "polynomially" multiplied with $< Rm >$

If MULGF instruction is available, than word-level polynomial squaring can also be implemented easily in hardware, to replace the appropriate software implementation. Since the software implementation of the word-level polynomial squaring (procedure *mr_sqr2*) takes approximately 5% in average for all benchmarks, this should give additional performance improvement over pure software implementation (Figures 5.2 to 5.6). The squaring operation can be done by using the MULGF instruction with identical register operands ($< Rs > = < Rm >$), i.e. as multiplying each word with itself.

Furthermore, a more detailed inspection of the EC code revealed that *mr_mul2* procedure is sometimes called in constructions like the following:

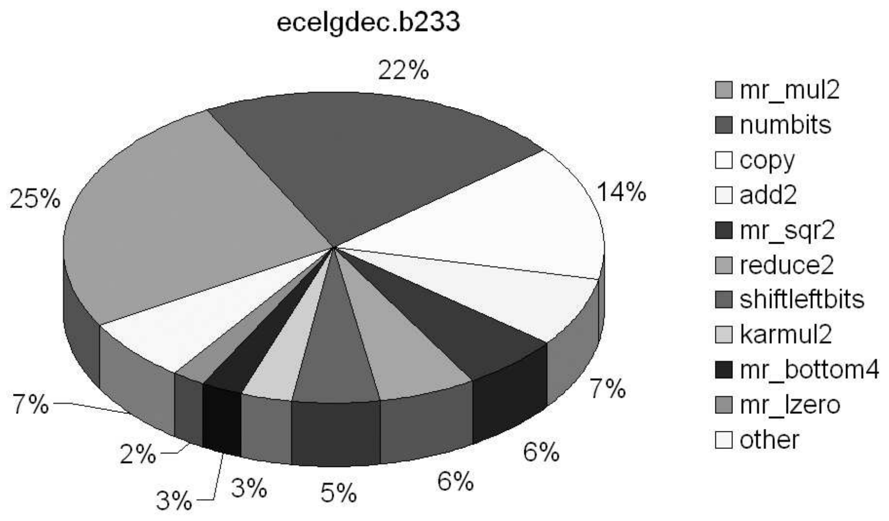

Figure 5.6. Breakdown of execution time (cycle level profiling of program functions) for ElGamal Decryption.

Algorithm 1 Typical use of mr_mul2 procedure in EC code

$A[i]$ and $B[j]$ are i-th and j-th word of the $GF(2^m)$ polynomials $a(x)$ and $b(x)$, respectively. $c(x)$ is the result of polynomial multiplication, $C[k]$ is the k-th word of the product, $\oplus$ is exclusive-or (XOR) operation, while al and bl are word lengths of $a(x)$ and $b(x)$.

```
for ($i = 0$ to $al$)
    for ($j = 0$ to $bl$)
        $P, Q \leftarrow mr\_mul2(A[i], B[j])$
        $C[i+j] \leftarrow C[i+j] \oplus P$
        $C[i+j+1] \leftarrow C[i+j+1] \oplus Q$
    end for
end for
```

The fact that the ARM architecture has a multiply-accumulate unit, which is able to execute long multiply-accumulate instructions (UMLAL, SMLAL), makes the previous code a very obvious candidate to replace with GF multiply-accumulate instruction, which we will call MLAGF:

$$MLAGF < RdLo >, < RdHi >, < Rm >, < Rs >$$

The MLAGF instruction ”polynomially” multiplies the values of registers $< Rm >$ and $< Rs >$ to produce a 64-bit result. The intermediate result is then is polynomially ”added” (xor-ed) to a 64-bit value held in $< RdHi >$ and $< RdLo >$, and the final product is written back to $< RdHi >$ and $< RdLo >$.

5.4 A Possible Hardware Implementation of the Proposed ISA Extensions

Polynomial multiplication is more simple than integer multiplication, because it does not have carries to worry about. The proposed instructions could be relatively easy incorporated into existing multiplier architecture, by adding the circuitry that will decide what shall be done with carry.
Proposals for designing unified integer and binary finite field multipliers already exist in literature [61], [62], and consist in replacing full-adders in multiplier by dual-field adders, which are able to switch between integer and binary-field mode.

A dual-field adder is basically a full-adder equipped with a capability of doing bit addition both with and without carry (Figure 5.7). It has an input called *fsel* (field select) that enables this functionality. When *fsel*=1, the dual field adder performs as the conventional full-adder, i.e does arithmetic in the field $GF(p)$. When *fsel*=0, on the other hand, the output *cout* is forced to 0 regardless of the values of the inputs. The output *sout* produces the result of bitwise XOR of three input values.
As in the standard full-adder circuit, the dual field adder has two XOR gates connected serially. Thus, its propagation time is not larger than that of full adder. Their areas differ slightly, but this does not cause a major change in the whole circuit.

Virtually, all modern processor architectures use Booth multiplication [29], [5]; this is the case for ARM also. A Booth recoded multiplier examines three bits of the

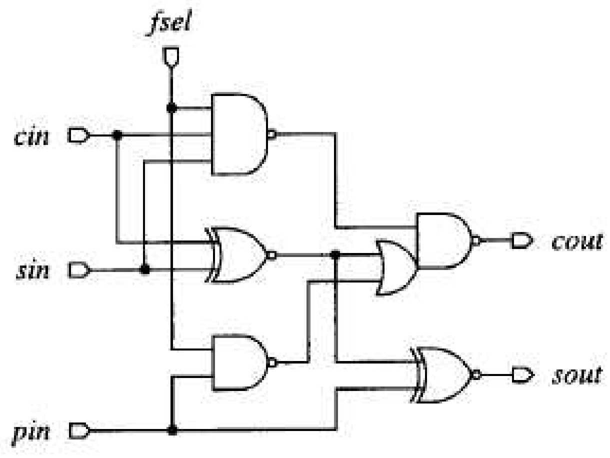

Figure 5.7. A dual-field adder.

multiplicand at a time to determine whether to add 0, 1, -1, 2, or -2 to the rank of the multiplicand. The structure of such multiplier does not allow for simple exchange of the full-adder with a dual-field adder. The study of multiplier architectures is beyond the scope of this dissertation, so we limited ourselves only to the basic idea.

A special functional unit that executes word-level polynomial squaring could be also integrated into processor's datapath. This unit should wire individual bits of the 32-bit input register to two 32-bit output registers, in a way that these bits are alternated with zeros, which could be done in only one cycle. However, having in mind that:

- word-level polynomial squaring takes in average 5% of the total execution time in our benchmarks;
- the processor configuration studied has a fully pipelined 3-cycle multiplier;
- the processor has separate integer and multiply-accumulate pipelines (Section 4.3)

performance gain over a MULGF squaring implementation would be minimal. For this reason, we measured the performance gain over a baseline software implementation when GF squaring is implemented using the new MULGF instruction.

Chapter

6

Performance Analysis of ISA Extensions for Different ECC Implementations

6.1 ECC Benchmark Set Execution with MULGF Instruction Set Extension

The impact of adding the MULGF instruction for word-level polynomial multiplication in finite field on the execution time and number of dynamically executed instructions is shown in Figures 6.1 and 6.2.

When projective coordinates are used, the number of dynamically executed instructions, as well as the execution time, are lower by approximately one-third in average, in comparison with the execution time without MULGF instruction (Figure 6.1). Here, for calculating the average improvement in execution time, all key sizes are taken into account, and MULGF is used only for multiplication. MULGF instruction was modeled to have a delay of three cycles, as the integer multiplier unit of ARM processor. This average result is in line with expectations, in fact, as the software implementation takes 500 cycles, the 3-cycle hardware implementation allows to reduce by a factor of 3/500 the time spent into word-level polynomial multiplication: from 34% on average (5.2 to 5.6) down to a negligible quota (i.e. less than 0.5%). The improvement in execution time is more significant for Diffie-Hellman (54% in number of instructions and 55% for execution time in $GF(2^{233})$) and ElGamal algorithms (48% for encryption and 37% for decryption in number of instructions, i.e. 39% and 35% in execution time in $GF(2^{233})$), where 32-bit polynomial multiplication is more used. The improvement for digital signature algorithm is more modest (19% in instruction number and 17% in execution time for the same key length), but still significant.

When MULGF instruction is used for polynomial squaring also, number of instructions and the total execution time are lower by additional 5% in average respect to the case when MULGF is used for multiplication only. This result is also in line with expectations, considering the similar share of the software word-level squaring procedure (*mr_sqr2* procedure in Figures 5.2 from 5.6).

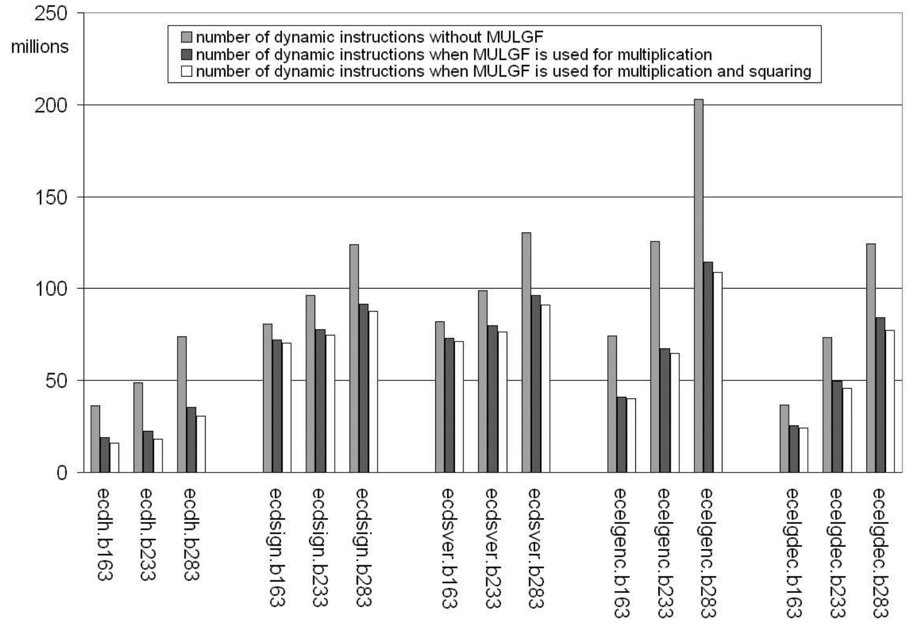

Figure 6.1. Number of dynamic instructions for projective coordinates before and after adding the MULGF instruction for word-level polynomial multiplication.

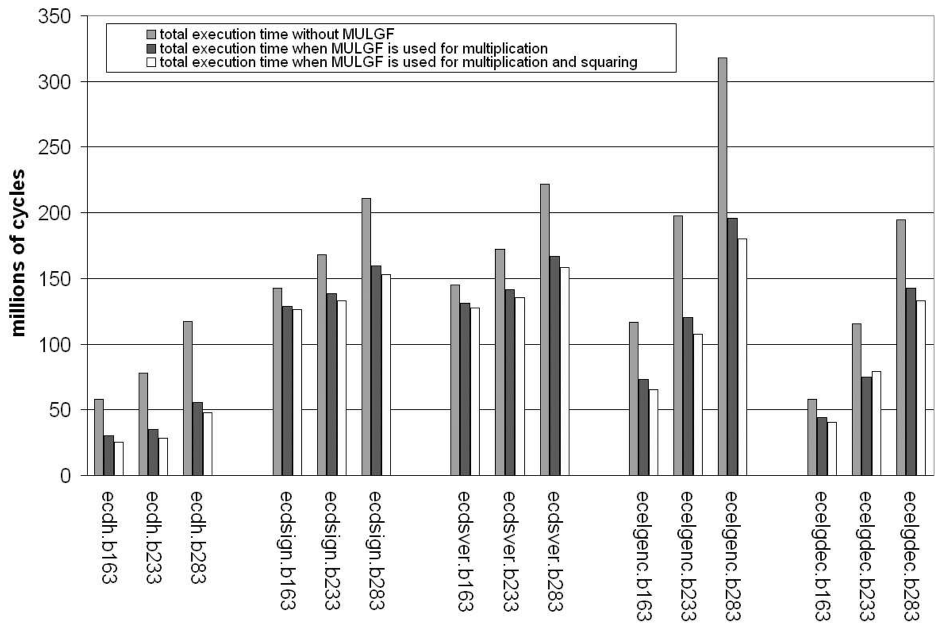

Figure 6.2. A comparison of the total execution time (in millions of cycles) for projective coordinates before and after adding the MULGF instruction.

6.2 Memory Performance with MULGF Extension

In order to analyze the benchmark influence on the cache subsystem, we have explored various cache configurations with cache size ranging from a few hundred bytes to 64KByte. Among them, we have selected a couple of interesting cases. Firstly, we measured instruction and data cache miss rates of ECC benchmarks with 32KByte high associativity (i.e., 32-way) instruction and data caches, which are representative of present Intel XScale processors based on ARM cores. The cache miss rates resulted practically negligible. If one considers the similarity of Figures 6.1 and 6.2, it is clear that memory does not influence significantly ECC performance, when the cache size is 32KByte+32KByte, and that the working set of ECC algorithms is small. Secondly, we repeated the experiments with 1KByte+1KByte direct-mapped instruction and data caches, which match the average working-set size of the considered ECC benchmarks, and thus allow analyzing very precisely the effects on the memory hierarchy before and after the MULGF instruction is added.

In particular, Figure 6.3 demonstrates the CPI with instruction and data caches of 1KByte+1KByte before and after adding the MULGF instruction divided into two parts: the first one due to processor operation, and the second one is due to memory operations.
The total CPI is lower in average by 27% after adding MULGF instruction, with largest improvement by 37% and 44% for digital signing and verification, respectively. This highlights two points:

- average CPI of the ECC benchmarks is similar to the software implementation of MULGF (*mr_mul2*) one;
- execution time reduction derives from the lower number of instructions executed (processor elaboration) and from improved memory CPI (lower number of accesses to instruction and data cache).

Figure 6.4 shows instruction and data cache miss rates for the two cases. From this figure, we see that while instruction cache misses are lower, data cache misses grow after adding MULGF instruction. This happens because, after adding MULGF, the instructions of the software implementation of MULGF (*mr_mul2* and *mr_sqr2*, Figures 5.2 to 5.6) are not loaded anymore in the instruction cache, and the benchmark working set can fit better in it. In the data cache, miss rate increases because the number of misses remains practically the same, but the number of cache accesses decreases. In fact, *mr_mul2* and *mr_sqr2* functions perform many operations on a few local variables and thus their memory accesses are essentially hits.

6.3 The Impact of MULGF Extension with Different Finite Field Multiplication Methods and Coordinate Representations

Finally, in our analysis we considered also the use of affine coordinates in ECC and studied the performance of affine-ECC with MULGF instruction present (Figure 6.5). As expected, our experiments show that adding MULGF has smaller impact when affine coordinates are used, because of the smaller number of finite field multiplications used (in affine coordinates, inversion in the finite field cannot be

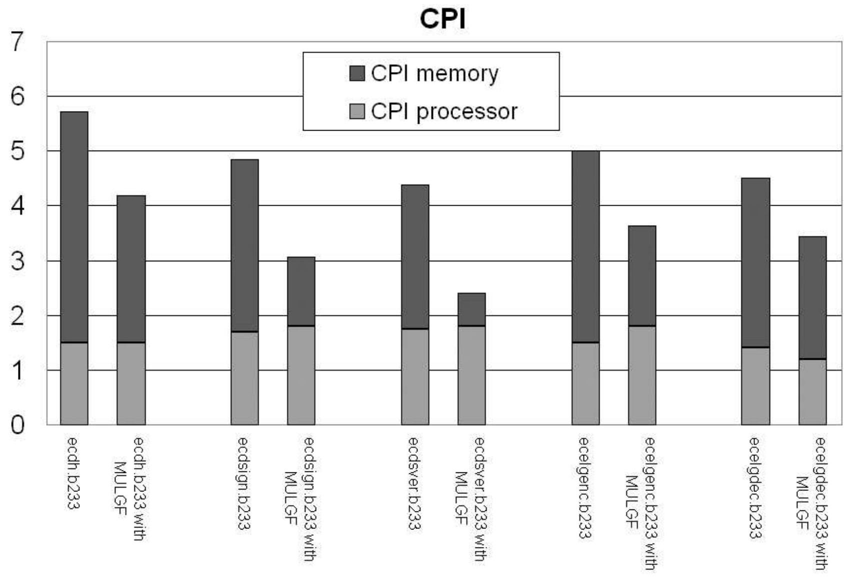

Figure 6.3. CPI (divided into processor and memory shares), before and after adding the MULGF instruction, with 1KB+1KB direct mapped instruction and data caches.

avoided, but some multiplications can be saved).

In case when affine coordinates are used, number of instructions and the execution time are lower in average by 13% for all benchmarks, if MULGF is adopted for multiplication and squaring. In addition, according to our experiments, projective coordinates are advantageous over affine ones both in a pure software implementation and when MULGF instruction is available. The improvement of the total execution time obtained by projective-MULGF implementation over affine coordinates without MULGF is approximately fourfold in average for all benchmarks, in both instruction number and execution time (Figure 6.5).

Next, we explored the effects of the MULGF instruction when different field multiplication methods are used. Figure 6.6 shows total execution time of different GF multiplication methods, together with the reduction. Even with MULGF instruction added, the Karatsuba polynomial multiplication (Section 3.1.2.2) remains the most efficient one for ARM platform. The MULGF lowers the execution time of Karatsuba, Pencil-and-Paper, and Montgomery finite field multiplication methods for approximately 30%. The fact that the Montgomery multiplication has the longest execution time is expected, considering that Montgomery multiplica-

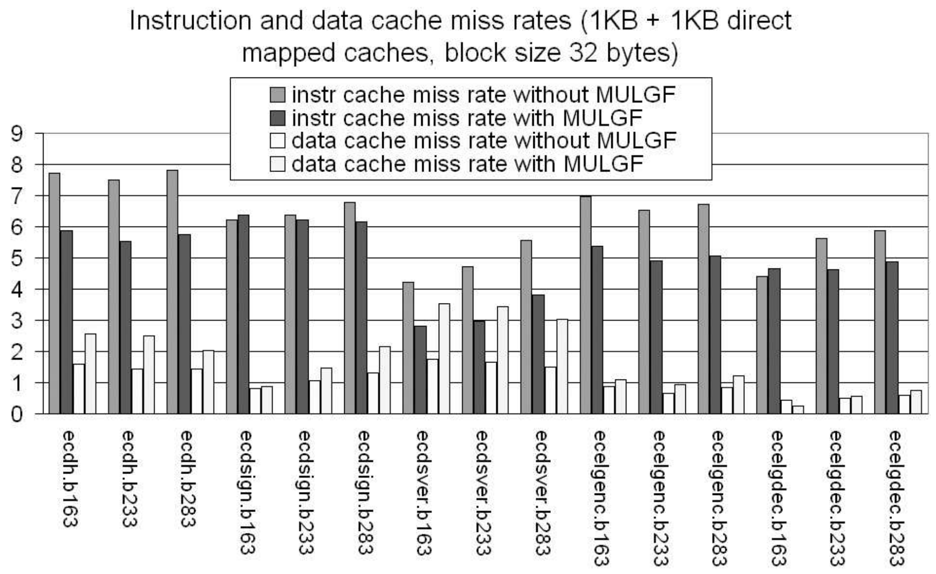

Figure 6.4. Instruction and data cache miss rates in percentages.

tion is done twice, in order to normalize the product (Section 3.1.2.2). Finally, since Comb polynomial multiplication does not use *mr_mul2*, adding MULGF has no effect at all. However, the Comb method takes longer that Karatsuba multiplication even without MULGF.

Figure 6.6 directly translates to execution time of the benchmarks, i.e. the shortest execution time is achieved when Karatsuba multiplication with MULGF is used.

6.4 The Effects of MLAGF Extension

The polynomial multiply-accumulate extension (MLAGF) has sense only when Pencil-and-Paper and Montgomery field multiplication methods are used, because they use the loop as in Algorithm 1 presented in Chapter 5.

We measured the impact of adding MLAGF when these two field multiplications are used. The experiments showed that the execution time of all the benchmarks for these two specific field multiplication methods improves in average by 0.2%. This result can be understood if one recalls that the simulated Intel XScale architecture has separate integer and multiply-accumulate pipelines, and therefore is able to

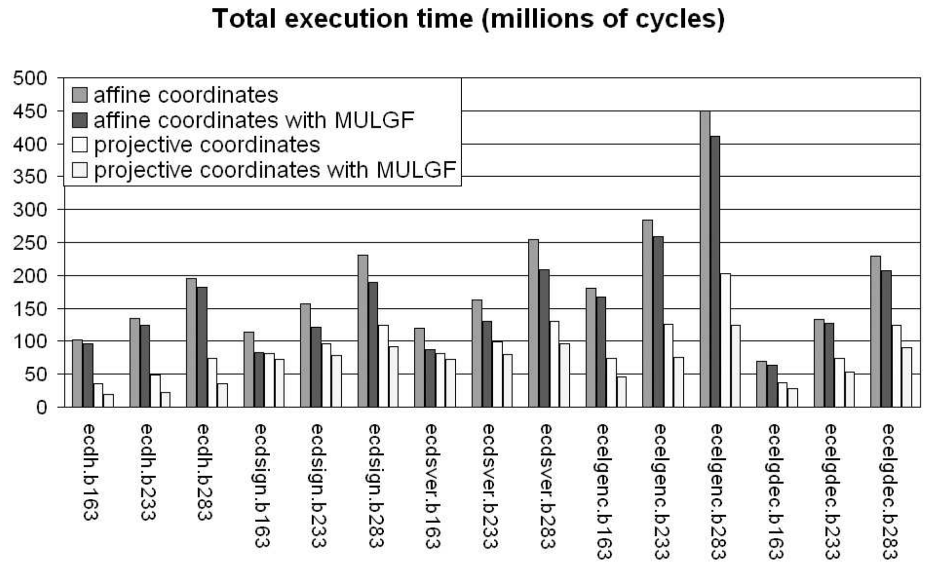

Figure 6.5. The impact of MULGF on the total execution time of the benchmarks with affine and projective coordinates.

schedule multiply-accumulate loop without stalling the pipeline even without the MLAGF extension. However, even if extending the ISA with MLAGF is not justified enough for the Intel XScale architecture that was studied in this dissertation, it would probably be justified for some less sophisticated architecture implementing simple, non-optimized software ECC.

6.5 Work in Progress

As we mentioned in Chapter 4, the initial software implementation of EC methods in Miracl library used as the base for our benchmark set was subsequently extended. In particular, we integrated the implementation of López-Dahab projective coordinates [42], as well as two additional scalar point multiplication methods: Modified Montgomery (Section 3.3.3) and Fixed-base Comb (Section 3.3.4). We are currently exploring the effects of ISA extensions when different projective coordinates and $[k]P$ methods are used. We are also currently in process of implementing the finite field operations in normal bases, with the intention to explore their efficiency and possible ISA extensions. We hope that this work could be used

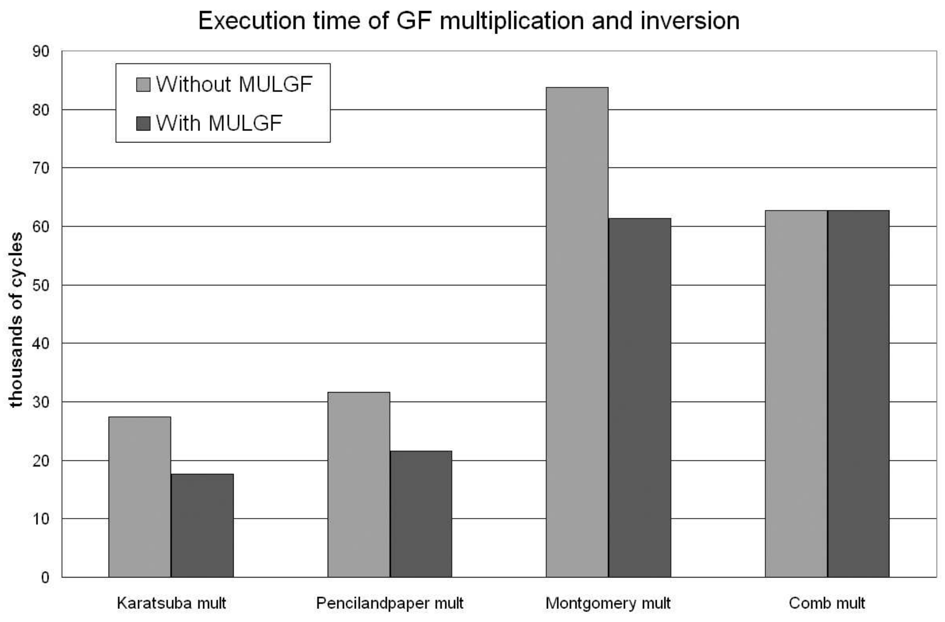

Figure 6.6. Total execution time of different finite field multiplication methods over $GF(2^{163})$. The cycles include the reduction procedure also.

as a guideline for choosing particular software ECC implementation suitable for embedded architectures such as Intel XScale.

We are also exploring the execution performance of ECC benchmarks when changing the memory latency, number of functional units in processor etc. For future work, we plan to generate hardware model of the proposed modified multiplier to estimate its latency, area, and power requirements, to be used for architectural tradeoff studies.

Recently, powerful attacks such as power analysis have been proposed as a means of attacking hardware-based cryptographic applications. These attacks have yielded extremely worrying results, breaking many existing systems (such as the smartcard systems commonly used in portable applications) in a matter of minutes. In the future, we would like to explore possible software and hardware countermeasures to these attacks.

Conclusion

Elliptic Curve Cryptography is becoming more and more used as an alternative to ”standard” public-key methods. Its major advantage is the fact that it uses shorter keys at security level equivalent to ”standard” public-key algorithms. This characteristic makes ECC especially well suited for implementation in embedded systems. As security issues become more and more pronounced in the next few years, we expect that elliptic curves will provide a viable solution for public-key cryptography in embedded devices.

The concept of instruction set extensions is a very attractive solution for improving application performance, because instead of using expensive hardware accelerators, requires only minimal modifications of the processor’s datapath.

In this dissertation, we presented an evaluation of instruction set extensions of an embedded processor for elliptic curve cryptography over binary finite fields $GF(2^m)$ that can efficiently replace a coprocessor that is typically used for improving performance of elliptic curve cryptography. Binary finite fields $GF(2^m)$, whose elements are generally represented as binary polynomials, present a data type which is not well supported in traditional, integer-oriented processor architectures. When the key arithmetic operations of this data type are implemented in software, we found that a very hight fraction of the execution time of elliptic curve cryptography algorithms is dominated by a few elementary operations. Based on the analysis of typical elliptic curve cryptography benchmarks, we proposed to extend the ISA for word-level polynomial multiplication in binary finite fields, called MULGF. We evaluated the impact of such extension on ECC performance in Intel XScale processor (ARM architecture). MULGF instruction for word-level polynomial multiplication can be added to existing ARM-like hardware by integrating a polynomial multiplier into existing datapath. Adding of this instruction is more justified when projective coordinates are used instead of affine, because when projective coordinates are used, finite field inversion is avoided at the price of higher number of finite field multiplications. In case when MULGF instruction is used for polynomial multiplication and squaring, it improves the execution time on average by 37%, and decreases the number of dynamically executed instructions by 37%.

Elliptic curve cryptography algorithms do not require large caches because of their reduced working set. Thus designs can use relatively small caches (i.e., 1KByte size) without affecting ECC performance significantly. Adding MULGF instruction lowers the instruction memory miss rate, while data cache miss rates increase because data working set (and consequently the number of misses) remains the same, while number of accesses decreases.

Our results and findings are applicable to a broad variety of modern processors. For example, a minimalist cryptographic processor could use the ISA extensions we considered for achieving additional performance, without integrating the features of more complex architectures. A general purpose processor may add the discussed ISA extensions to its base instruction set to achieve higher ECC performance. Elliptic curve cryptography implemented on a processor with extended ISA can thus result in a very efficient implementation with increased speed performance over the pure software solution. Finally, the proposed extensions can be useful in other applications where finite field arithmetic is used, and not exclusively for elliptic curve cryptography.

Bibliography

[1] T. Acar, *High-speed Algorithms and Architectures for Number-Theoretic Cryptosystems*, Ph.D. Thesis, Oregon State University, 1998.

[2] G. B. Agnew, R. C. Mullin, S. A. Vanstone, "An Implementation of ECC over $GF(2^{155})$," *IEEE Journal on Selected Areas in Communication*, vol. 11, 1993.

[3] ANSI X9.62, "Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)," 1999.

[4] ANSI X9.63, "Public Key Cryptography for the Financial Services Industry: Elliptic Curve Key Agreement and Key Transport Protocols," working draft, October 2000.

[5] ARM Corporation, *ARM Architecture Reference Manual*, Prentice-Hall, New York, 1996.

[6] T. Austin, E. Larson, D. Ernst, "Simplescalar: An Infrastructure for Computer System Modeling," *IEEE Computer*, vol. 35, issue 2, pp. 56–59,2002.

[7] M. Aydos, T. Yanik, Ç. K. Koç, "High-speed Implementation of an ECC-based Wireless Authentication Protocol on ARM Microprocessor," *IEEE Proceedings: Communications 148*, pp. 273–279, October 2001.

[8] D. V. Bailey, C. Paar, "Efficient Arithmetic in Finite Field Extensions with Application in Elliptic Curve Cryptography," *Journal of Cryptology*, vol. 14, no. 3, pp. 153-176, 2001.

[9] L. E. Bassham, “Efficiency testing of ANSI C implementations of round 1 candidate algorithms for the Advanced Encryption Standard,” *Third Advanced Encryption Standard (AES) Conference*, pp. 136–148, 2000.

[10] M. Bednara, M. Daldrup, J. V. Gaten, J. Shorokllahi, J. Teich, “Reconfigurable Implementation of Elliptic Curve Crypto Algorithms,” in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, 2002.

[11] I.F. Blake, G. Seroussi, N. P. Smart, *Elliptic Curves in Cryptography*, Cambridge University Press,1999.

[12] J. Buchmann, J. Loho, J. Zayer, “An implementation of the general number sieve,” in *Advances in Cryptology – Crypto’93*, pp. 159–166, Springer-Verlag, 1994.

[13] J. Burke, J. McDonald, T. Austin, “Architectural Support for Fast Symmetric-Key Cryptography,” in *Proceedings of the ACM/IEEE Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pp. 178–189, 2000.

[14] Certicom tutorial on Elliptic Curve Cryptography, available online at http://www.certicom.com/resources/ecc_tutorial

[15] P. G. Comba, “Exponentiation Systems on the IBM PC,” *IBM Systems Journal*, vol. 29, no. 4, pp. 526–538, 1990.

[16] E. De Win, S. Mister, B. Prennel, M. Wiener, “On the performance of signature based on elliptic curves,” in *Proceedings of the Third International Symposium on Algorithmic Number Theory*, Lecture Notes in Computer Science, no. 1423, pp. 252–266, Springer-Verlag, 1998.

[17] W. Diffie, M. E. Hellman, “New directions in cryptography,” *IEEE Transactions on Information Theory*, vol. 22, pp. 644–654, November 1976.

[18] ECC Cipher Suites for TLS, Internet draft, draft-ietf-tls-ecc-01.txt, work in progress, 2001.

[19] T. ElGamal, “A public-key cryptosystem and a signature scheme based on discrete logarithms,” *IEEE Transactions on Information Theory*, vol. 31, no. 4, pp. 469–472, July 1985.

[20] A. M. Fiskiran, R. B. Lee, “Workload Characterization of Elliptic Curve Cryptography and other Network Security Algorithms in Constrained Environments,” in *Proceedings of the 5th Annual Workshop on Workload Characterization*, pp. 127–137, 2002.

[21] J. Goodman, *Energy Scalable Reconfigurable Cryptographic Hardware for Portable Applications*, Ph.D. Thesis, Massachusetts Institute of Technology, 2000.

[22] J. Grosschadl,“Architectural Support for Long Integer Modulo Arithmetic on Risc-Based Smart Cards,” *International Journal of High Performance Computing Applications*, Vol. 17, No. 2, pp. 135–146, 2003.

[23] J. Guajardo, R. Bluemel, U. Krieger, C. Paar, “Efficient Implementation of Elliptic Curve Cryptosystems on the TI MSP430x33x Family of Microcontrollers,” in *Proceedings of PKC’01*, Lecture Notes in Computer Science, no. 1992, pp. 365–382, Springer-Verlag, 2001.

[24] J. Guajardo, C. Paar, “Efficient Algorithms for Elliptic Curve Cryptosystems,” *Advances in Cryptology - Crypto’97*, Lecture Notes in Computer Science, no. 1294, pp. 342–356, Springer-Verlag, 1997.

[25] V. Gupta, S. Gupta, S. C. Chang, “Performance Analysis of Elliptic Curve Cryptography for SSL,” in *Proceedings of ACM Workshop on Wireless Security*, pp. 87–94, Atlanta, USA, September 2002.

[26] N. Gura, S. Chang Shantz, H. Eberle, S. Gupta, V. Gupta, D. Finchelstein, E. Goupy, D. Stebila, “An End-To-End Systems Approach to Elliptic Curve Cryptography,” Sun Microsystems Labs, 2001.

[27] A. Halbutoğullari, *Fast Bit-Level, Word-Level and Parallel Arithmetic in Finite Fields for Elliptic Curve Cryptosystems*, Ph.D. Thesis, Oregon State University, 1998.

[28] D. R. Hankerson, J. C. López Hernandes, A. J. Menezes, "Software Implementations of Elliptic Curve Cryptography over Binary Finite Fields," *Cryptographic Hardware and Embedded Systems – CHES 2000*, Lecture Notes in Computer Science, pp. 1–24, Springer-Verlag, 2000.

[29] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 2^{nd} edition, Morgan Kaufmann Publishers, 1997.

[30] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 3^{rd} edition, Morgan Kaufmann Publishers, 2002.

[31] IEEE P1363 Standard Specifications for Public-Key Cryptography, available online at http://grouper.ieee.org/groups/1363/

[32] ISO/IEC 14888-3, "Information Technology Security Techniques - Digital Signature with Appendix - Part 3: Certificate Based Mechanisms," 1998.

[33] ISO/IEC 15946, "Information Technology - Security Techniques - Cryptographic Techniques Based on Elliptic Curves," Committee Draft, 1999.

[34] The Intel XScale Microarchitecture Technical Summary, available online at ftp://download.intel.com/design/intelxscale/XScaleDatasheet4.pdf

[35] A. Karatsuba, Y. Ofman, "Multiplication of Multidigit Numbers on Automata," *Soviet Physics - Koklady*, vol. 7, pp. 595–596, 1963.

[36] D. A. Knuth, *The Art of Computer Programming 2, Seminumerical Algorithms*, Addison-Wesley, Second Edition, 1981.

[37] N. Koblitz, *Elliptic Curve Cryptosystem*, Mathematics of Computation, no. 48, pp. 203–209,1987.

[38] N. Koblitz, *A Course in Number Theory and Cryptography*, Springer-Verlag, New York, 1987.

[39] N. Koblitz, A. Menezes, S. Vanstone, "The state of Elliptic Curve Cryptography," *Design, Codes and Cryptography*, vol. 19, pp. 173–193, 2000.

[40] Ç. K. Koç, T. Acar, "Montgomery Multiplication in $GF(2^k)$," *Design, Codes and Cryptography*, vol. 14, no. 1, pp. 59–67, January 1998.

[41] H. Li, C. N. Zhang, "Efficient Cellular Automata Based Versatile Multiplier for $GF(2^n)$," *Journal of Information Science and Engineering*, vol. 18, pp. 479–488, 2002.

[42] J. López, R. Dahab,"Improved Algorithms for Elliptic Curve Arithmetic in $GF(2^n)$," in *Proceedings of SAC'98*, Lecture Notes in Computer Science, no. 1556, pp. 201–212, Springer-Verlag, 1998.

[43] J. López, R. Dahab, "Fast Multiplication on Elliptic Curves over $GF(2^m)$ without Precomputation," in *Proceedings of CHES'99*, Lecture Notes in Computer Science, no. 1717, pp. 316–327, Springer-Verlag, 1999.

[44] J. López, R. Dahab, "High-speed Software Multiplication in F_{2^m}," in *Proceedings of Indocrypt 2000*, Lecture Notes in Computer Science, no. 1977, pp. 203–212, Springer-Verlag, 2000.

[45] J. López, R. Dahab, "An Overview of Elliptic Curve Cryptography," Technical report TR-IC-00-10, University of Campinas, Brasil, 2000.

[46] A. J. Menezes, *Elliptic Curve Public Key Cryptosystems*, Kluwer Academic Publishers, Boston, 1995.

[47] A. J. Menezes, P. C. Van Oorschot, S. A. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 1997.

[48] V. Miller, "Uses of Elliptic Curves in Cryptography," in *Advances in Cryptology – Crypto'95*, Lecture Notes in Computer Science, no. 218, pp. 417–426, 1986.

[49] Miracl big integer library, available online at http://indigo.ir/ mscott

[50] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, vol. 44, pp. 519–521, 1985.

[51] R. Mullin, I. Onyszchuk, S. Vanstone, R. Wilson, "Optimal normal bases in $GF(p^n)$," *Discrete Applied Mathematics*, no. 22, pp. 149–161, 1988.

[52] National Institute of Standards and Technology, “Secure Hash Standard,” FIPS Publication 180-1, 1995.

[53] National Institute of Standards and Technology, “Digital Signature Standard,” FIPS Publication 186-2, 2000.

[54] S. Okada,N. Torii,K. Itoh, M. Takenaka, “Implementation of Elliptic Curve Cryptographic Coprocessor over $GF(2^m)$ on an FPGA,” in *Proceedings of Cryptographic Hardware and Embedded Systems - CHES 2000*, Lecture Notes in Computer Science, no. 1965, pp. 25–40, Springer-Verlag, 2000.

[55] OpenSSL, available online at http://www.openssl.org

[56] G. Orlando, C. Paar, “A High-Performance Reconfigurable Elliptic Curve Processor for $GF(2^n)$,” in *Proceedings of CHES 2000*,2000.

[57] J. Daemen, V. Rijmen, “AES Proposal: Rijndael,” AES submission to NIST, 1998.

[58] R. Rivest, A. Shamir, L. Adleman, “A method for obtaining digital signatures and public-key cryptosystems,” *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, February 1978.

[59] M. Rosner, “Elliptic Curve Cryptosystems on Reconfigurable Hardware,” Master’s thesis, Worchester Polytechnic Institute, 1998.

[60] RSA Laboratories’ FAQs about today’s cryptography, available online at http://www.rsasecurity.com/rsalabs/faq

[61] A. Satoh, K. Takano, “A Scalable Dual-Field Elliptic Curve Cryptographic Processor,” *IEEE Transactions on Computers*, vol. 52, no. 4, pp. 449–460, April 2003.

[62] E. Savaş, A. F. Tenca, Ç. K. Koç, “A Scalable and Unified Multiplier Architecture for Finite Fields $GF(p)$ and $GF(2^n)$,” in *Cryptographic Hardware and Embedded Systems – CHES 2000*, Lecture Notes in Computer Science, pp. 227–292, Springer-Verlag, 2000.

[63] B. Schneier, *Applied Cryptography*, John Wiley & Sons, Inc., Second edition, 1996.

[64] R. Schroeppel, H. Orman, S. O'Malley, O. Spatscheck, "Fast Key Exchange with Elliptic Curve Cryptosystems," in *Proceedings of Advances in Cryptology – Crypto'95*, Lecture Notes in Computer Science, no. 963, pp. 43–56, Springer-Verlag, 1995.

[65] SimpleScalar architectural simulator, available online at http://www.simplescalar.com

[66] J. Solinas, "Efficient Arithmetic on Koblitz Curves," *Design, Codes, and Cryptography*, no. 19, pp. 195–249, 2000.

[67] A. Weimerskirch, C. Paar, S. Chang Shantz, "Elliptic Curve Cryptography on a Palm OS Device," in *Proceedings of ACISP 2001*, Springer-Verlag, 2001.

[68] A. Weimerskirch, D. Stebila,S. Chang Shantz, "Generic $GF(2^m)$ Arithmetic in Software and its Application to ECC," in *Proceedings of the Eighth Australasian Conference on Information Security and Privacy (ACISP 2003)*, Wollongong, Australia, 2003.

[69] A. Woodbury, D. Bailey, C. Paar, "Elliptic Curve Cryptography on Smart Cards without Coprocessors," in *Proceedings of the Fourth Smart Card Research and Advanced Applications (CARDIS 2000) Conference*, Bristol, UK, 2000.