Design, Analysis, and FPGA prototyping of High-Performance Arithmetic for Cryptographic Applications
Literature Review - Part 1: Cryptographic Algorithms

Mostafa Abd-El-Barr, Alaaeldin Amin and Turki F. Al-Somani

Computer Engineering Department, King Fahd University
of Petroleum & Minerals, Dhahran 31261, Saudi Arabia,
E-mail: {mostafa, amin, tsomani}@ccse.kfupm.edu.sa

Abstract

This report presents a brief survey on secret key and public key cryptography algorithms. These include: block ciphers, stream ciphers, RSA, ElGamal and Elliptic Curve Cryptosystems (ECC). Since ECC achieved security levels comparable to those of traditional public key cryptosystems using smaller keys (160 bits), this work focus more on ECC. Finally, this work also presents a survey on scalar multiplication algorithms used in ECC.

1 Introduction

During the last century, digital communication has become a major part of peoples every day's life. Industrialized countries are evolving towards information societies, where bank cards, mobile phones and wireless Internet access are available to every citizen. Moreover, the value of information keeps growing, while it is being subjected to an increased number of threats such as eavesdropping of communication lines and theft of sensitive data. This clearly demonstrates that there is a strong need for techniques to protect information and information systems. Cryptography is the science of protecting information. A cryptographic algorithm is a mathematical function that uses a key to encipher information. Without knowledge of the key,

deciphering is not possible. Contemporary cryptography deals with more issues than plain encryption. Mathematical concepts that can be used to construct digital signatures or protocols for entity authentication, are well established.

In typical scenario, two entities which are called party#1 and party#2, want to exchange messages securely over an insecure channel. The adversary can have several goals including eavesdropping the communication or altering it. Eavesdropping is a threat to the confidentiality of the messages, i.e., no unintended third part can understand the content. Altering is a threat to the integrity of the message, i.e., no-one can change the message while it is in transit. In addition, the adversary could also try impose one of the communicating parties. Therefore, it is required to authenticate (i.e. get assurance of the identity) the communicating parties. The last of the four basic security services deals with non-repudiation, i.e., the property that the sender cannot deny a message was sent.

This work is organized as follows: Section 2 briefly introduces secret key cryptography. Section 3 provides a brief introduction to public key cryptography. This includes RSA, ElGamal and ECC. Section 4 focuses more on ECC while Section 5 introduces a survey on scalar multiplication algorithms. Finally, Section 6 concludes the report.

2 Secret Key Cryptography

Numerous applications use (software) implementations of cryptographic algorithms to provide security at low cost. The most important requirement for such algorithms, besides their resistance against attacks, is that their use causes minimum performance degradation to the application that uses them. Three types of secret key cryptographic primitives are discussed in this section. *Block ciphers* are used to encrypt data. If block ciphers are not fast enough for an application, *stream ciphers* can be used as alternative. In order to ensure integrity of data, modification detection codes (MDC) which are also called *hash functions* are used. For authentication, message authentication codes (MACs) are used.

2.1 Block Ciphers

A *block cipher* is defined as a set of Boolean permutations operating on n-bit vectors. This set contains a Boolean permutation for each value of a key k. In other words, a block cipher is a length preserving transformation which

takes an element (a plaintext) from the set of plaintexts and transforms it into an element (the ciphertext) from the set of ciphertexts under the influence of a key [1].

A block cipher usually consists of several operations (transformations), which form the encryption algorithm. To allow efficient implementation, block ciphers apply the same Boolean transformation several times on a plaintext. The Boolean transformation is called *round function*, and the block cipher is called an *iterated block cipher*. For each round, a key has to be used. To generate such round keys K_i from the cipher key, a *key schedule* algorithm is applied to the cipher key. If the *round key* is exclusive or-ed (x-ored) to an intermediate value, then the cipher is called *key-alternating block cipher*.

Modern block ciphers are usually based on two different types of designs. *Feistel ciphers* operate on $2m$ input bits. The plaintext is split into a right and a left half, L and R, and the round function f only acts on one of the halves.

$$\begin{aligned} L_i &= R_{i-1} \\ R_{i-1} &= L_{i-1} \oplus f(R_{i-1}, K_i) \end{aligned}$$

As a consequence of the Feistel structure, it takes two rounds before all plaintext bits have been subject to the round transformation. Decryption can be done in the same way as encryption with the round keys supplied in reverse order. Moreover, the round function f does not have to be a permutation. The traditional DES (Data Encryption Standard) [2] consists of an algorithm of that type. Another popular construction is called a *substitution permutation network* (SPN). This construction emphasizes on separating confusion (substitution) and diffusion (permutation) in the cipher. In many recent SPN ciphers, the permutation layer has been replaced by an affine transformation that is chosen in such a way that a high level of diffusion is guaranteed. Rijndael[1] which is an algorithm using the new AES (Advanced Encryption Standard) [3] is an example for this design. Feistel ciphers can also be considered as a kind of SPNs.

Whenever a message being longer than the block size needs to be encrypted, the block cipher is used in a certain mode of operation. For the DES, four modes have been standardized [4]. The *electronic code book* (ECB) mode corresponds to the usual use of a block cipher; the message is split into blocks and each block is encrypted separately with the same key. In the *cipher block chaining* (CBC) mode, each ciphertext block is x-ored (chained) with the next plaintext block before being encrypted with

the key. In the *output feedback* (OFB) mode and the *cipher feedback* (CFB) modes, a keystream is generated and x-ored with the plaintext. Hence, the latter two modes work as a synchronous additive stream cipher.

There is ongoing work to define new modes. Five confidentiality modes are specified in [5] for use with any approved block cipher, such as the AES algorithm.

2.2 Stream Ciphers

Stream ciphers encrypt individual characters, which are usually bits, of a plaintext one at a time. They use an encryption transformation which varies with time. Hence, in contrast to block ciphers, the encryption depends not only on the key and the plaintext, but also on the current state. As mentioned before at the end of the previous section, a block cipher can be turned into a stream cipher by using a certain mode of operation (such as CFB or OFB). There are no stream ciphers that are standardized today. However, the de-facto standard is the RC4 stream cipher [6].

Synchronous stream ciphers generate a keystream independently of the plain- text messages and of the ciphertext. Sender and receiver must therefore be synchronized; they must use the same key and operate at the same state within that key. A ciphertext digit that is corrupted during transmission does not influence any other ciphertext bit.

An *asynchronous* stream cipher is a stream cipher in which the keystream is generated as a function of the key and a fixed number of previous ciphertext bits. Because the keystream is dependent on only a few previous ciphertext bits, self-synchronization is possible even if some of the transmitted ciphertext bits are corrupted.

2.3 MDCs

An MDC (hash function) takes an input of arbitrary length and compresses (digests) it to an output of fixed length, which is called the *hash value*. Cryptographic hash functions satisfy the following additional properties:

1. *preimage resistance*: it should be computationally infeasible to find a preimage to a given hash value,
2. *second (2nd) preimage resistance*: it should be computationally infeasible to find a 2nd preimage to a given input,
3. *collision resistance*: it should be computationally infeasible to find two different inputs with the same hash value.

Hash functions are used to ensure the integrity of data. This can be done by using the data as input to the hash function and storing its output. Later on, to verify that the input data has not been altered, the hash value is recomputed using the data at hand and compared with the original hash value. Another application for hash functions is to use them in digital signature schemes. Hash functions of the secure hash algorithms (SHA) family have been standardized by NIST [7].

2.4 MACs

Hash functions which involve a secret key are called message authentication codes (MACs). The output of such a keyed hash function is also called MAC. In contrast to to MDCs they can also be used to guarantee data origin authentication (i.e. corroborate the source of information) and data integrity. Most contemporary MACs are constructed based upon either a block cipher or a hash function.

In order to ensure authenticity of data, an entity computes a MAC on the data by using the private key. In order to verify the authenticity of the data later on, the MAC can be recomputed by anyone who may access the private key. Standardized MACs are the data authentication code (DAC) [8] and the new keyed-hash message authentication code (HMAC) [9].

3 Public Key Cryptography

In public key cryptography, secret keys are replaced by keypairs consisting of a *private key*, which must be kept confidential, and a *public key*, which is made available to the public by for example publishing it in a directory. Anyone who wishes to send a message to the owner of a certain keypair will take the public key, encrypt the message under this public key and send it to the owner of the keypair. The owner can decrypt the message by using the private key.

This idea works out in practice because the private and the public key are linked in a mathematical way (by a mathematical function) such that knowing the public key does not allow the recovery of the private key. Several mathematical functions which are useful for public key cryptography are known today. Amongst others, the most important hard mathematical problems are:

- *IFP* (integer factorization problem): given a positiv integer n, find its prime factorization $n = p_1^{e_1} p_2^{e_2} ... p_k^{e_k}$

- *GDLP* (generalized discrete logarithm problem): given a finite cyclic group G of order n, a generator α of G, and an element $\beta \in G$, find the integer $x, 0 \leq x \leq n-1$, such that $a^x = \beta$.
- *DHP* (Diffie-Hellman problem): given a prime p, a generator α of Z_p^*, and elements $\alpha^a \bmod p$ and $\alpha^b \bmod p$, find $\alpha^{ab} \bmod p$.

In the subsequent sections, we discuss a reference cryptosystem for each of the above given hard mathematical problems. Thereafter, we sketch two other important applications for public-key cryptography, namely key agreement and digital signatures.

3.1 RSA

The Rivest-Shamir-Adleman (RSA) algorithm [10] was the first public key encryption algorithm invented. Under certain assumptions, the RSA algorithm is based on the IFP. RSA keys are generated as follows: one selects two large secret prime numbers p and q and computes the public RSA modulus $n = pq$. Then one chooses a public encryption exponent e which satisfies $gcd(e, (p-1)(q-1)) = 1$. The private key, i.e. the decryption exponent, d can then be calculated by solving $ed = 1 (\text{mod } (p-1)(q-1))$. Hence the public key is the tuple (e, n) and the private key is the triple (d, p, q).

Suppose that someone wishes to encrypt a message for the entity which is associated with the public key (e, n). Then, the message needs to be represented as a number $m < n$. The ciphertext is computed by raising m to the power e : $c = m^e (\text{mod } n)$. This message can be decrypted by exponentiation with the private key $m = c^d (\text{mod } n)$.

3.2 ElGamal

The ElGamal encryption algorithm[11] is based on the DLP in the finite field Z_p^*. An advantage of ElGamal is that some public parameters can be shared by a group of users. These so-called *domain parameters* are a large prime p such that p-1 is divisible by another prime q, and an element $g \in Z_p^*$ of order divisible by q. The private key is chosen to be an integer d. The public key e can then be computed by solving $e = g^d \pmod p$.

Suppose that someone wishes to encrypt a message for the entity which is associated with the public key (e, p, q). Then, the message needs to be represented as a number $m < p$. In a first step of the encryption procedure, a random key k is generated. The ciphertext c consists of a pair $c = (c_1, c_2) = (g^k, me^k)$. To decrypt the ciphertext, c_2/c_1^d is computed.

Since each message has a different ephemeral key, encrypting the same message twice will produce two different ciphertexts. Cryptosystems with this property are also called *non-deterministic* or *randomized* cryptosystems.

3.3 ECC

Since its inception, *elliptic curve cryptography* [12] has been the subject of extensive cryptanalysis. Today, elliptic curve cryptosystems (ECCs) are deemed secure for commercial as well as government use and has been included in many standards [13] - [19]. Based on today's crypt-analytical knowledge, elliptic curve cryptosystems achieve security levels comparable to those of traditional public-key cryptosystems, e.g. RSA, ElGamal, and those based on the Diffie-Hellman key agreement algorithm [20], using smaller keys and computationally more efficient algorithms.

The ability to use smaller keys and algorithms that are more computationally efficient than traditional cryptographic algorithms are two of the main reasons why elliptic curve cryptography is becoming popular for use particularly in constrained environments, such as smart cards, cellular phones and personal digital assistant devices, which have limited memory and are battery-powered. The same reasons also make elliptic curve cryptography attractive for high performance systems, such as secure networking devices, whose ability to protect and route traffic is a function of their capacity to establish secure connections. Establishing these secure connections often involves public-key operations. Elliptic curve cryptography is discussed in details in the next section.

4 Elliptic Curve Cryptography

Elliptic Curve Cryptosystem (ECC), which was originally proposed by Niel Koblitz and Victor Miller in 1985 [12], is seen as a serious alternative to RSA with much shorter key size. ECC with key size of 128-256 bits is shown to offer equal security to that of RSA with key size of 1-2K bits. To date, no significant breakthroughs have been made in determining weaknesses in the ECC algorithm, which is based on the discrete logarithm problem over points on an elliptic curve. The fact that the problem appears so difficult to crack means that key sizes can be reduced in size considerably, even exponentially. This made ECC to become a serious challenge to the RSA. The advantage of ECC is being recognized recently where it is being incorporated in my standards [13]-[19]. ECC have gained popularity for

cryptographic applications because of the short key compared with earlier public key cryptosystems such as RSA and ElGamal. They are considered particularly suitable for implementation on smart cards or mobile devices.

4.1 $GF(2^m)$ Arithmetic Background

The finite $GF(2^m)$ field has particular importance in cryptography since it leads to particularly efficient hardware implementations. Elements of the field are represented in terms of a basis. Most implementations use either a *Polynomial Basis* or a *Normal Basis*. For the implementation described in this report, a normal basis is used since it leads to more efficient hardware implementations. Normal basis is more suitable for hardware implementations than polynomial basis since operations are mainly comprised of rotation, shifting and exclusive-OR operations which can be efficiently implemented in hardware. A normal basis of $GF(2^m)$ is a basis of the form $(\beta^{2^0}, \beta^{2^1}, \beta^{2^2}, ...\beta^{2^{m-1}})$, where $\beta \in GF(2^m)$

In a normal basis, an element $A \in GF(2^m)$ can be uniquely represented in the form $A = \sum_{i=0}^{m-1} \alpha_i \beta^{2^i}$,where $a_i \in \{0, 1\}$. $GF(2^m)$ operations using normal basis are performed as follows.

1. Addition and squaring. Addition is performed by a simple bit-wise exclusive-OR (XOR) operation, while squaring is simply a rotate left operation.

2. Multiplication. $\forall\ A, B \in GF(2^m)$, where $A = \sum_{i=0}^{m-1} a_i \beta^{2^i}$ and $B = \sum_{i=0}^{m-1} b_i \beta^{2^i}$, the product $C = A * B$, is given by:

$$C = A * B = \sum_{i=0}^{m-1} c_i \beta^{2^i}$$

then multiplication is defined in terms of a multiplication table $\lambda_{ij} \in \{0, 1\}$.

$$c_k = \sum_{i=0}^{m-1} \sum_{j=0}^{m-1} \lambda_{ij} a_{i+k} b_{j+k} \beta^{2^i} \tag{1}$$

An optimal normal basis (ONB) [21] is one with the minimum number of terms in (1), or equivalently, the minimum possible number of nonzero λ_{ij}. This value is $2m - 1$, and since it allows multiplication

with minimum complexity, such a basis would normally lead to more efficient hardware implementations.

3. Inversion. Inverse of $a \in GF(2^m)$, denoted as a^{-1}, is defined as follows.

$$aa^{-1} \equiv 1 \ mod \ 2^m$$

The algorithm used for inversion is derived from Fermat's Little Theorem

$$a^{-1} = a^{2^m-2} = (a^{2^{m-1}-1})^2$$

for all $a \neq 0$ in $GF(2^m)$. The method chosen was proposed by Itoh and Tsujii [22], based on the following decomposition which minimizes the number of multiplications (squarings are much cheaper in a normal basis).
If m is odd, then

$$2^{m-1} = (2^{\frac{m-1}{2}} - 1)(2^{\frac{m-1}{2}} + 1), \text{ and}$$

$$a^{2^{m-1}} = (a^{2^{\frac{m-1}{2}}-1})^{2^{\frac{m-1}{2}}+1}$$

which can be computed using one field multiplication provided $a^{2^{\frac{m-1}{2}}-1}$ is given. The cost of squaring is ignored because it is insignificant compared to multiplication.
If m is even, then

$$2^{m-1} - 1 = 2(2^{m-2} - 1) + 1 = 2(2^{\frac{m-2}{2}} - 1)(2^{\frac{m-2}{2}} + 1) + 1, \text{ and}$$

$$a^{2^{m-1}} = a^{2(2^{\frac{m-2}{2}}-1)(2^{\frac{m-2}{2}}+1)+1}$$

which requires two field multiplications if $a^{(2^{\frac{m-2}{2}}-1)}$ is given.

4.2 $GF(2^m)$ Elliptic Curve Arithmetic

This section provides a brief introduction to elliptic curve arithmetic. An elliptic curve over a finite field $GF(q)$ defines a set of points (x, y) that satisfy the elliptic curve equation together with the point $\mathcal{O}$, known as the "point at infinity"[12]. The "point at infinity" does not satisfy the elliptic curve equation. The coordinates x and y of the elliptic curve points are elements of the field $GF(q)$, where $q = pm$ and p is prime. This report will focus on elliptic curves defined over $GF(2^m)$, where m is prime. Hardware implementations, in this case, have both higher security and computation efficiency with addition requiring no carry propagation, squaring reduced to a simple rotate operation, and no discrimination is necessary between positive and negative numbers [16]. Only non-supersingular curves will be considered since they are more secure than supersingular curves. Supersingular elliptic curves are special class of curves with some special properties that make them unstable for cryptography [23]. Equation (2) defines the elliptic curve equation for fields $GF(2^m)$.

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

where $a, b \in GF(2^m)$ and $b \neq 0$.

The set of discrete points on an elliptic curve form an abelian group (commutative group), whose group operation is known as point addition. Bounds for the number of discrete points n on an elliptic curve over a finite field $GF(q)$ are defined by Hasse's theorem given in Equation (3), where the symbol n represents the number of points on the elliptic curve and where $q = pm$ represents the number of elements in the underlying finite field.

$$q + 1 - 2\sqrt{q} \leq n \leq q + 1 + 2\sqrt{q} \tag{3}$$

Elliptic curve *point addition* is defined according to the "chord-tangent process". Point addition is described as follows:

Let P and Q be two distinct points on an elliptic curve E defined over the real numbers with $Q \neq -P$ (Q is not the additive inverse of P). The addition of P and Q is the point R ($R = P + Q$); where R is the additive inverse of S, and S is a third point on the elliptic curve intercepted by the straight line through points P and Q. For the curve under consideration, R is the reflection of the point S with respect to the *x-axis*; that is, if R is the point (x, y), S is the point $(x, -y)$. The addition operation just described is illustrated in Figure 1.

When $P = Q$ and $P \neq -P$, the addition of P and Q is the point R ($R = 2P$); where R is the additive inverse of S, and S is the third point

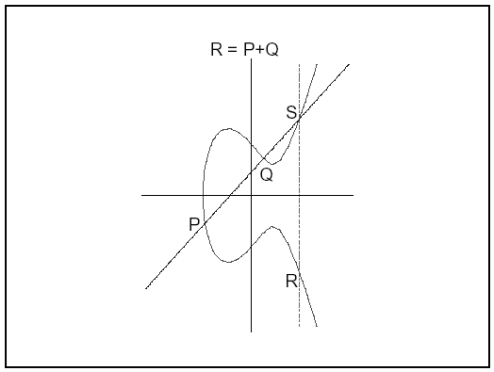

Figure 1: Point Addition

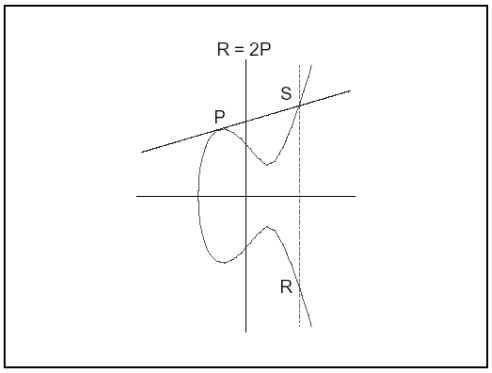

Figure 2: Point Doubling

on the elliptic curve intercepted by the straight line tangent to the curve at point P. This operation is referred to as *point doubling*, and is shown in Figure 2.

The "point at infinity", $\mathcal{O}$, is the additive identity of the group. The most relevant operations involving $\mathcal{O}$ are the following:

- The addition of a point P and $\mathcal{O}$ is equal to P ($P + \mathcal{O} = P$)
- The addition of a point P and its additive inverse $-P$ is equal to $\mathcal{O}$ ($P - P = \mathcal{O}$). If P is a point on the curve, then $-P$ is also a point on the curve.

Point addition and the point doubling operations are generally com-

puted using algebraic formulas derived from the geometrical operations just described. Assuming affine coordinates, a point on the curve is represented by two coordinates, x and y. For an elliptic curve E defined over $GF(2^m)$, $E : y^2 + xy = x^3 + ax^2 + b$, let $P = (x_1, y_1) \in E$; then $-P = (x_1, x_1 + y_1)$.

- If $Q = (x_2, y_2) \in E$, $P \neq Q$ and $Q \neq -P$, then $P+Q = (x_3, y_3)$, where

$$x_3 = (\tfrac{y_1+y_2}{x_1+x_2})^2 + (\tfrac{y_1+y_2}{x_1+x_2}) + x_1 + x_2 + a$$

$$y_3 = (\tfrac{y_1+y_2}{x_1+x_2}) \cdot (x_1 + x_3) + x_3 + y_1$$

- If $P = Q = (x_1, y_1)$, then $2P = P + P = (x_3, y_3)$, where

$$x_3 = x_1^2 + \tfrac{b}{x_1^2}$$

$$y_3 = x_1^2 + (x_1 + \tfrac{y_1}{x_1})x_3 + x_3$$

Projective coordinates are used to eliminate the number of inversions [23]. For elliptic curve defined over $GF(2^k)$, many different forms of formulas are found [24]-[27] for point addition and doubling. The projective coordinates (Pr), so called homogeneous coordinates, takes the form $(x, y) = (X/Z, Y/Z)$ [25], while the Jacobian coordinates takes the form $(x, y) = (X/Z^2, Y/Z^3)$ [26] and the Lopez-Dahab coordinates takes the form $(x, y) = (X/Z, Y/Z^2)$ [27]. From the Jacobian coordinates, two other coordinates where proposed. These are: the Chudnovsky Jacobian coordinates (J^c) representing the point with the quintuple (X, Y, Z, Z^2, Z^3) and the Modified Jacobian coordinates (J^m) representing the point with the quadruple (X, Y, Z, aZ^4). Mixed coordinates was proposed in [24] leading to better performance.

Point subtraction is a useful operation in some algorithms. This operation can be performed with the point addition or point doubling formulas using the additive inverse of the point to be subtracted. For example, the point subtraction $P-Q$ can be computed using the point addition operation where: $P - Q = P + (-Q)$. The additive inverse of a point $P = (x, y)$ is the point $(x, x + y)$ for curves defined over the $GF(2^m)$ fields.

The point operation used by elliptic curve cryptosystems is referred to as point multiplication. This operation is also referred to as scalar point

multiplication. The point multiplication operation is denoted as kP, where k is an integer number and P is point on the elliptic curve. The operation kP represents the addition of k copies of point P as shown in Equation (4).

$$kP = \underbrace{P + P + \cdots + P}_{k \ times \ P} \tag{4}$$

Elliptic curve cryptosystems are built over cyclic groups. Each group contains a finite number of points, n, that can be represented as scalar multiples of a generator point: iP for $i = 0, 1, ..., n-1$, where P is a generator of the group. The order of point P is n, which implies that $nP = \mathcal{O}$ and $iP \neq \mathcal{O}$ for $1 < i < n-1$. The order of each point on the group must divide n. Consequently, a point multiplication kQ for $k > n$ can be computed as $(k \ mod \ n)Q$.

4.3 Elliptic Curve Scalar Multiplication

Scalar multiplication is the basic operation for ECC. Scalar multiplication in the group of points of an elliptic curve is the analogous of exponentiation in the multiplicative group of integers modulo a fixed integer m. Computing kP can be performed using a straightforward double-and-add approach based on the binary representation of $k = (k_{l-1}, ..., k_0)$ where k_{l-1} is the most significant bit of k. Other scalar multiplication methods have been proposed in the literature. A good survey is conducted by Gordon in [28]. These scalar multiplication algorithms will be described in detail next Section.

4.4 Elliptic Curve Discrete Logarithm Problem

Elliptic curve cryptosystems base their security on what is known as the elliptic curve discrete logarithm problem. This problem can be stated as follows. Given a known elliptic curve and two known points P and Q, where $Q = kP$, it is computationally infeasible to determine the value of k if the parameters have been carefully chosen. Elliptic curve cryptosystems use cyclic groups with very large number of points. For example, the FIPS 186-2 standard [19] recommends groups for which the number of points ranges from about 2^{163} to about 2^{571} points, depending on the curve and the underlying finite field. The best cryptanalysis algorithms known today, such as the Pollard's rho algorithm [29], computes an elliptic curve discrete logarithm with an average of $O\sqrt{n}$ point operations if the parameters have been carefully chosen, where n represents the number of points of the cyclic group being used.

Using the best cryptanalysis algorithms, determining a discrete logarithm in the curves specified by the FIPS 186-2 standard require over 2^{80} point operations [19]. This is an intractable exponential problem given the current computer technology. It is important to realize that well-chosen curves achieve the required degree of security. Other curves may exhibit structures that facilitate cryptanalysis; for example, curves defined over composite fields of characteristic two [30].

5 Scalar Multiplication Algorithms

This Section describes some of the most popular scalar multiplication algorithms. The algorithms discussed here for scalar multiplications are mostly variants of similar algorithms employed for exponentiation. The most popular algorithms for exponentiation are described in [31]. The algorithms for scalar multiplication covered here are discussed in detail in [14],[28],[32]-[35]. The following two subsections study two main classes of algorithms. These are the generic scalar multiplication algorithms that can be used to compute an arbitrary scalar multiplication and the fixed-point scalar multiplication algorithms, which can be used to compute scalar multiplications involving fixed points. The fixed-point scalar multiplication algorithms are of interest because scalar multiplication with fixed points can be computed much more efficiently than for arbitrary points. In addition, fixed-point multiplication is a common operation in elliptic curve cryptographic algorithms.

5.1 Generic Scalar Multiplication Algorithms

Generic scalar multiplication algorithms can be used to compute scalar multiplications involving arbitrary points. This section discusses six point multiplication algorithms: double-and-add (or binary), w-ary, addition-subtraction, signed w-ary, width-w addition-subtraction scalar multiplication and Montgomery scalar multiplication algorithms.

5.1.1 Double-and-Add Scalar Multiplication Algorithm

One of the simplest scalar multiplication algorithms is the double-and-add point multiplication algorithm. Algorithm 1 shows the left-to-right version of the double-and-add scalar multiplication algorithm. This algorithm inspects the multiplier k, starting with its most significant bit and ending

with its least significant bit. For each inspected bit, the algorithm performs a point double (Step 2.1), and, if the inspected bit is a one, the algorithms also performs a point add (Step 2.2).

The double-and-add scalar multiplication algorithm requires, on average, l point doubles and $l/2$ point additions, where $l \approx \lceil log_2\ k \rceil$. This algorithm also requires the storage of two points, P and Q.

Algorithm 1 Double-and-add scalar multiplication algorithm

Inputs: P, k
Output: kP
Initialization:
1. $Q = \mathcal{O}$
Scalar Multiplication:
2. for $i = l - 1$ down to 0 do
   2.1. $Q = 2Q$
   2.2. if $k_i = 1$ then $Q = Q + P$
   end for
3. return(Q)

5.1.2 *w*-ary Scalar Multiplication Algorithm

The w-ary scalar multiplication algorithm, Algorithm 2, is a generalization of the double-and-add scalar multiplication algorithm that process w bits of the multiplier k in each iteration.

The first main step of Algorithm 2 is the recoding of the multiplier k in radix 2^w with $\lceil l/w \rceil$ digits in the range $[0, 2^w)$: $k = \sum_{i=0}^{\lceil l/w \rceil - 1} k'_i 2^{wi}$ with $k'_i \in [0, 2^w)$. This representation can be derived directly from the binary representation of k; for example, the number $k = (01101100)_2$ is recoded as $k = (1230)_4$ in radix 4. The second main step of Algorithm 2 is the precomputation of the values iP for $i \in [2, 2^w)$. The basic idea is to compute these values once and then use the precomputed values as necessary in the sacalr multiplication operation.

The third main step of Algorithm 2 is the actual scalar multiplication operation. This operation involves $\lceil l/w \rceil$ iterations. In each iteration, the accumulated value Q is doubled w times (Step 4.1). If the recoded digit k'_i is nonzero, then the precomputed point $P_{k'_i}$ is added to the accumulated point (Step 4.2.1). Note that as in Algorithm 2 the recoded digits are consumed starting with the most significant digit and ending with the least significant

Algorithm 2 w-ary scalar multiplication algorithm

Inputs: P, k
Output: kP
Recoding of k : $k = \sum_{i=0}^{\lceil l/w \rceil - 1} k'_i 2^{wi},\ k'_i \in [0, 2^w)$.
1. for $i = 0$ to $\lceil l/w \rceil - 1$ do
   1.1. $k'_i = k \bmod 2^w$
   1.2. $k = k - k'_i$
   1.3. $k = k/2^w$
   end for
Initialization:
2. $Q = \mathcal{O}$, $P_1 = P$
Precomputations: $P_i = iP,\ i \in [0, 2^w)$.
3. for $i = 1$ to $2^{w-1} - 1$ do
   3.1. $P_{2i} = 2P_i$
   3.2. $P_{2i+1} = P_{2i} + P$
   end for
Scalar Multiplication:
4. for $i = \lceil l/w \rceil - 1$ down to 0 do
   4.1. $Q = 2^w Q$
   4.2. if $k'_i \neq 0$ then
      4.2.1. $Q = Q + P_{k'_i}$
   end for
5. return(Q)

one.

The precomputation effort of the w-ary scalar multiplication algorithm requires approximately 2^{w-1} point doubles and 2^{w-1} point additions. The scalar multiplication phase requires on average l point doubles and l/w point additions, when assuming that $k'_i \neq 0$ for all i and where $l \approx \lceil log_2\ k \rceil$. In total, the algorithm requires approximately $2^{w-1} + l$ point doubles and $2^{w-1} + l/w$ point additions. The algorithm also requires the storage of approximately 2^w points.

For a given set of parameters, it may be advantageous to compute $2^w P$ directly in Step 4.1 with a closed expression instead of computing it with w individual point doubles $(2(2(2(...\ 2P))))$.

The w-ary scalar multiplication algorithm is a fixed-size windowing algorithm. Sliding window algorithms are extensions of the w-ary scalar multiplication algorithm that use variable-size windows. The complexity of point double operations is typically lower than that of point addition operations, which suggests that larger speedups could be obtained in scalar multiplication algorithms.

5.1.3 Addition-Subtraction Scalar Multiplication Algorithm

The addition-subtraction scalar multiplication algorithm, shown in Algorithm 3, is an extension of the double-and-add scalar multiplication algorithm that computes scalar multiplications using point additions, point subtractions, and point doubles. By incorporating point subtractions, whose computational complexities are quite similar to those of point additions, this algorithm achieves a lower computational complexity than the double-and-add point multiplication algorithm with a relatively small increase in complexity.

The addition-subtraction point multiplication algorithm can be realized with different signed digit representations. This work considers the use of the non-adjacent form representation (NAF) described in [36]. Using this representation a multiplier $k = \sum_{i=0}^{l-1} k_i 2^i$ is uniquely recoded as $k = \sum_{i=0}^{l} k'_i 2^i$ with $k'_i \in [-1, 1]$, where the recoded representation does not contain contiguous nonzero digits and where the average number of nonzero digits is $l/3$. The non-adjacent form (NAF) representation of an l-bit multiplier k is at most $l + 1$ digits long.

The first main step of the addition-subtraction scalar multiplication algorithm is the recoding of the multiplier k. The second main step is the scalar multiplication operation. The scalar multiplication operation consists of l

Algorithm 3 Addition-subtraction scalar multiplication algorithm

Inputs: P, k
Output: kP
Recoding of k : $k = \sum_{i=0}^{l} k'_i 2^i,\ k'_i \in [-1, 1]$.
1. for $i = 0$ to l do
  1.1. if k mod 2=1 then
    1.1.1 $k'_i = 2 - k \bmod 2^2$
  1.2. else
    1.2.1 $k'_i = 0$
  1.3 $k = k - k'_i$
  1.4 $k = k/2$
  end for
Initialization:
2. $Q = \mathcal{O}$
Scalar Multiplication:
3. for $i = l$ down to 0 do
  3.1. $Q = 2Q$
  3.2. if $k'_i = 1$ then
    3.2.1. $Q = Q + P$
  3.3. if $k'_i = -1$ then
    3.3.1. $Q = Q - P$
  end for
4. return(Q)

+ 1 loop iterations. In each loop iteration an accumulated point is doubled (Step 3.1). Also in each iteration of the loop, if the recoded digit under inspection is a one, a point is added to the accumulated point (Step 3.2.1). If the value of the recoded digit is negative one (-1), a point is subtracted from the accumulated point (Step 3.3.1).

The addition-subtraction point multiplication algorithm requires, on average, l point doubles and $l/3$ point additions. This algorithm also requires the storage of two points, P and Q.

5.1.4 Signed w-ary Scalar Multiplication Algorithm

The signed w-ary scalar multiplication algorithm, shown in Algorithm 4, is an extension of the w-ary scalar multiplication algorithm that computes scalar multiplications using point additions, point subtractions, and point doubles.

The first main step of Algorithm 4 is the recoding of the multiplier k in radix 2^w with $\lceil (l+1)/w \rceil$ digits in the range $[-2^{w-1}, 2^{w-1}) : \ k \sum_{i=0}^{\lceil (l+1)/w \rceil - 1} k'_i 2^{wi}$. For example, using this representation, the number $k = (11100100)_2$ can be recoded as $k = (10\bar{2}10)_4$ in radix 4, where $\bar{2} = -2$. (Note that the signed w-ary scalar multiplication algorithm can be implemented using different signed digit representations.)

The second main step of Algorithm 4 is the precomputation of the values iP for $i \in [2, 2^{w-1}]$. The basic idea is to compute these values once and then use the precomputed values as necessary in the scalar multiplication operation. The additive inverse of a point is generated as necessary when performing point subtractions.

The third main step of Algorithm 4 is the actual point multiplication operation. This operation involves $\lceil (l+1)/w \rceil - 1$ iterations. In each iteration, the accumulated value of Q is doubled w times (Step 5.1). If the recoded digit k'_i is greater than 0, then the point $P_{|k'_i|}$ is added to the accumulated point (Step 5.2.1). If the recoded digit k'_i is negative, then the point $P_{|k'_i|}$ is subtracted from the accumulated point (Step 5.3.1). Note that as in the w-ary scalar multiplication algorithm the recoded digits are consumed starting with the most significant digit and ending with the least significant one.

The precomputations of the signed w-ary scalar multiplication algorithm requires approximately 2^{w-2} point doubles and 2^{w-2} point additions. The scalar multiplication phase requires approximately l point doubles and l/w point additions (including subtractions), when assuming that $k'_i \neq 0$ for all i and where $l \approx \lceil log_2 \ k \rceil$. In total, the algorithm requires approximately

$2^{w-2} + l$ point doubles and $2^{w-2} + l/w$ point additions (including subtractions). The algorithm also requires the storage of 2^{w-1} points.

In comparison with the w-ary scalar multiplication algorithm, the signed w-ary scalar multiplication algorithm requires the precomputation and storage of half as many points.

Algorithm 4 Signed w-ary scalar multiplication algorithm

Inputs: P, k
Output: kP
Recoding of k : $k = \sum_{i=0}^{\lceil (l+1)/w \rceil - 1} k'_i 2^{wi},\ k'_i \in [-2^{w-1}, 2^{w-1})$.
1. for $i = 0$ to $\lceil (l+1)/w \rceil - 1$ do
   1.1. $k'_i = k \bmod 2^w$
   1.2. if $k'_i \geq 2^{w-1}$ then
      1.2.1 $k'_i = -(2^w - k'_i)$
   1.3. $k = k - k'_i$
   1.4. $k = k/2^w$
   end for
Initialization:
2. $Q = \mathcal{O}$, $P_1 = P$
Precomputations: $P_i = iP,\ i \in [1, 2^{w-1}]$.
3. for $i = 1$ to $2^{w-2} - 1$ do
   3.1. $P_{2i} = 2P_i$
   3.2. $P_{2i+1} = P_{2i} + P$
   end for
4. $P_{2^{w-1}} = 2P_{2^{w-2}}$
Scalar Multiplication:
5. for $i = \lceil (l+1)/w \rceil - 1$ down to 0 do
   5.1. $Q = 2^w Q$
   5.2. if $k'_i > 0$ then
      5.2.1. $Q = Q + P_{k_i}$
   5.3. if $k'_i < 0$ then
      5.3.1. $Q = Q - P_{|k_i|}$
   end for
6. return(Q)

5.1.5 Width-w Addition-Subtraction Scalar Multiplication Algorithm

Algorithm 5 shows the width-w addition-subtraction scalar multiplication algorithm described in [35].

The first main step of the algorithm is the recoding of the multiplier k (Steps 1 1.4). In this algorithm, the multiplier $k = \sum_{i=0}^{l-1} k_i 2^i$ with $k'_i \in [0, 1]$, is recoded using a width-w non-adjacent form (w-NAF) as follows: $k = \sum_{i=0}^{l} k'_i 2^i$ where $k'_i \in (-2^{w-1}, 2^{w-1})$ and k'_i is odd.

The second main step of the algorithm is the precomputation of the points iP for odd values of i in the range $[3, 2^{w-1})$ (Steps 3-5.1).

The third main step of the algorithm is the scalar multiplication process. This is an iterative process in which one digit of the recoded k is inspected in each iteration. In each iteration the accumulated point is doubled (Step 6.1). If the scanned digit is greater than zero, the point $P_{\lfloor k'_i/2 \rfloor}$ is added to the accumulated point (Step 6.2.1). If the scanned digit is negative, the point $P_{\lfloor |k'_i|/2 \rfloor}$ is subtracted from the accumulated point (Step 6.3.1).

The precomputation phase of the width-w addition-subtraction scalar multiplication algorithm requires one point double and $2^{w-2} - 1$ point additions. The scalar multiplication phase requires on average approximately l point doubles and $l/(w + 1)$ point additions (including subtractions), where $l \approx \lceil log_2\ k \rceil$. In total, the algorithm requires approximately l point doubles and $2^{w-2} + l/(w+1)$ point additions (including subtractions). The algorithm also requires the storage of approximately 2^{w-2} points.

5.1.6 Montgomery Scalar Multiplication Algorithm

The Montgomery scalar multiplication algorithm for $GF(2^m)$ discussed here uses a variant of the double-and-add point multiplication algorithm. The algorithm is based on the observation that the x coordinate of the sum of two points P_1 and P_2, whose difference is known to be $P(P_2 - P_1 = P)$, can be computed using the x coordinates of the points P, P_1, and P_2. The y coordinate of the point P_1, which contains the scalar multiplication result at the end of the scalar multiplication process, can be recovered using the x coordinates of P, P_1, and P_2 together with the y coordinate of P (see Algorithm 6).

The description in the previous paragraph corresponds to the affine coordinates version of the algorithm, which is unattractive for implementation because it requires the computation of inverses for each group operation.

Algorithm 5 Width-w Addition-Subtraction Scalar Multiplication Algorithm

Inputs: P, k
Output: kP
Recoding of k : $k = \sum_{i=0}^{l} k'_i 2^i,\ k'_i \in (-2^{w-1}, 2^{w-1})$.
1. for $i = 0$ to l do
  1.1. if $k \bmod 2^w = 1$ then
    1.1.1. $k'_i = k \bmod 2^w$
    1.1.2. if $k'_i \geq 2^{w-1} = 1$ then
      1.1.2.1 $k'_i = -(2^w - k'_i)$
  1.2. else
    1.2.1. $k'_i = 0$
  1.3. $k = k - k'_i$
  1.4. $k = k/2$
  end for
Initialization:
2. $Q = \mathcal{O}$, $P_1 = P$
Precomputations:
3. $P_0 = P$
4. $T = 2P$
5. for $i = 1$ to $2^{w-2} - 1$ do
  5.1 $P_i = P_{i-1} + T$
end for
Scalar Multiplication:
6. for $i = l$ down to 0 do
  6.1. $Q = 2Q$
  6.2. if $k'_i > 0$ then
    6.2.1. $Q = Q + P_{\lfloor k'_i/2 \rfloor}$
  6.3. if $k'_i < 0$ then
    6.3.1. $Q = Q - P_{\lfloor |k'_i|/2 \rfloor}$
  end for
7. return(Q)

Algorithm 6 Montgomery Point Multiplication Algorithm

Inputs: P, k
Output: kP
Initialization:
1. $k = (k_{l-1}...k_1k_0)_2$
2. $P_1 = P$, $P_2 = 2P$
Scalar Multiplication:
3. for $i = l - 2$ down to 0 do
   3.1. if $k_i = 1$ then $P_1 = P_1 + P_2$, $P_2 = 2P_2$
   3.2. else $P_2 = P_1 + P_2$, $P_1 = 2P_1$
end for
4. return$(Q = P_1)$

An attractive projective coordinates version of the algorithm was also introduced in [37].

5.1.7 Comparisons between generic scalar multiplication algorithms

This section provides a brief comparisons between generic scalar multiplication algorithms. The comparisons are based on the algorithm complexity and the storage requirements. The complexity is specified in terms of point additions and point doubles. While the storage requirements are specified in terms of the number of points that needs to be stored.

The complexity and the storage requirements of generic scalar multiplication algorithms are summarized in Table 1. It is clear from Table 1 that the generic scalar multiplication algorithms require about one point double per bit in the binary representation of k. It is also clear that the generic scalar multiplication algorithms differ in how they reduce the number of point additions which is done by multiplier recoding, precomputation or combination between multiplier recoding and precomputation.

The addition- subtraction point multiplication algorithm reduces the number of point additions over the double-and-add point multiplication algorithm by recoding the multiplier k (point subtractions are treated as point additions). The double-and-add, the addition-subtraction, and the Montgomery scalar multiplication algorithms require no precomputation. The other generic algorithms include combinations of multiplier recoding and precomputation. The number of precomputations in these algorithms grows

Algorithm	Average Complexity ($\log_2 k \approx m$)		Storage Requirments
	# point double	# point add	# points
double-and-add	m	$m/2$	2
w-ary	$2^{w-1} + m$	$2^{w-1} + m/w$	2^w
addition-subtraction	m	$m/3$	2
signed w-ary	$2^{w-2} + m$	$2^{w-2} + m/w$	2^{w-1}
width-w addition-subtraction	m	$2^{w-2} + m/(w+1)$	2^{w-2}
Montgomery	m	m	3

Table 1: Complexity of generic scalar multiplication algorithms

exponentially with the window size, and the number of point additions, excluding the ones required for the precomputations, grows inversely proportional with the window size. The memory requirements grow exponentially, as each precomputed point requires storage.

5.2 Fixed-Point Scalar Multiplication Algorithms

This section discusses the special case of scalar multiplication using fixed points. This operation is used in elliptic curves cryptographic algorithms, such as the analogues of the Diffe-Hellman key agreement algorithm, the ElGamal encryption and digital signature algorithms, and the Digital Signature Algorithm (DSA).

5.2.1 Fixed-Point Windowing Scalar Multiplication Algorithm

The fixed-point windowing scalar multiplication algorithm, described by Algorithm 7, is based on the fixed-base exponentiation algorithm introduced in [38]. Algorithm 7 shows a variant of the fixed-point windowing point multiplication algorithm discussed in [28] that recodes the multiplier k using signed digit representation.

In the fixed-point windowing scalar multiplication algorithm, the multiplier k is recorded as $k = \sum_{i=0}^{\lceil (l+1)/w \rceil - 1} k'_i 2^{wi}$ with $k'_i \in (-2^{w-1}, 2^{w-1})$.. Using this recoding, the scalar multiplication can be expressed as follows: $kP = \sum_{i=0}^{\lceil (l+1)/w \rceil - 1} k'_i (2^{wi} P)$. Because the point P is known, it is possible

to precompute the points $2^{wi}P$ for $i = 1...\lceil (l+1)/w \rceil - 1$. Given the precomputed points, a scalar multiplication can be computed by adding, or subtracting, $\mid k'_i \mid$ copies of $2^{wi}P$ for all $k'_i \neq 0$. The fixed-point windowing scalar multiplication algorithm performs these point additions and point subtractions in an efficient manner.

The first main step of the fixed-point windowing scalar multiplication algorithm is the off-line precomputation of the points $2^{wi}P$ for $i = 1...\lceil (l+1)/w \rceil - 1$(Steps 1-2.1).

The second main step of the fixed-point windowing scalar multiplication algorithm is the recoding of the multiplier k (Steps 3-3.4).

The third main step of the fixed-point scalar multiplication algorithm is the scalar multiplication process (Steps 5-5.3). This is an iterative process that adds a point $k'_i(2^{wi}P)$ by adding the point $2^{wi}P$ to an accumulated point when $k'_i > 0$ or by subtracting the point $2^{wi}P$ when $k'_i < 0$. From the iteration at which the point $2^{wi}P$ is added or subtracted till the last loop iteration, the accumulated point is added to itself k'_i times; therefore, the accumulated point incorporates the point $k'_i(2^{wi}P)$ in its result.

Assuming off-line precomputation, the scalar multiplication requires approximately $2^{w-1} + l/w$ point additions (including subtractions), where $l \approx \lceil log_2\ k \rceil$. The algorithm also requires the storage of approximately $\lceil l/w \rceil$ points.

5.2.2 Fixed-Point Comb Scalar Multiplication Algorithm

The fixed-point comb scalar multiplication algorithm, described by Algorithm 8, is based on the fixed-base comb exponentiation algorithm introduced in [39]. The fixed-point comb scalar multiplication algorithm arranges the scalar multiplier $k = \sum_{i=0}^{l-1} k_i 2^i$ as shown to the left of the vertical bar in Figure 2.4. The arrangement of the multiplier k consists of v two-dimensional arrays, where each two-dimensional array contains h rows and b columns and where $l = ah$ and $a = vb$ (note that the multiplier k can be extended by adding zeros to the most significant bit positions to meet these conditions).

The contribution of digit k_i in the scalar multiplication $kP = \sum_{i=0}^{l-1} k_i 2^i P,\ k_i 2^i P$, is shown in Figure 2.1 as follows. The weight of a multiplier k_i within a row is listed in the top row and the weight of a row is listed to the right of the vertical bar. To determine the contribution of bit k_i multiply the weight at the top of the column containing k_i, the weight of the row that contains k_i, and the value of k_i; this process forms the value $k_i 2^i P$.

Algorithm 7 Fixed-Point Windowing Scalar Multiplication Algorithm

Inputs: P, k
Output: kP
Off-Line precomputation: $P_i = 2^{wi}P$
1. $P_0 = P$
2. for $i = 1$ to $\lceil (l+1)/w \rceil - 1$ do
   2.1. $P_i = 2^{wi}P_{i-1}$
   end for
Recoding of k : $k = \sum_{i=0}^{\lceil (l+1)/w \rceil - 1} k'_i 2^{wi},\ k'_i \in [-2^{w-1}, 2^{w-1})$.
3. for $i = 0$ to $\lceil (l+1)/w \rceil - 1$ do
   3.1. $k'_i = k \bmod 2^w$
   3.2. if $k'_i \geq 2^{w-1}$ then
      3.2.1 $k'_i = -(2^w - k'_i)$
   3.3. $k = k - k'_i$
   3.4. $k = k/2^w$
   end for
Initialization:
4. $A = \mathcal{O},\ B = \mathcal{O}$
Scalar Multiplication:
5. for $i = 2^{w-1}$ down to 1 do
   5.1. for each i for which $k'_i = j$ do
      5.1.1 $B = B + P_i$
   5.2. for each i for which $k'_i = -j$ do
      5.2.1 $B = B - P_i$
   5.3. $A = A + B$
6. return(A)

2^{b-1}	$\cdots$	2	1	
k_{b-1}	$\cdots$	k_1	k_0	P
k_{a+b-1}	$\cdots$	k_{a+1}	k_a	2^aP
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$k_{(h-1)a+b-1}$	$\cdots$	$k_{(h-1)a+1}$	$k_{(h-1)a}$	$2^{(h-1)a}P$
k_{2b-1}	$\cdots$	k_{b+1}	k_b	2^bP
k_{a+2b-1}	$\cdots$	k_{a+b+1}	k_{a+b}	$2^{a+b}P$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$k_{(h-1)a+2b-1}$	$\cdots$	$k_{(h-1)a+b+1}$	$k_{(h-1)a+b}$	$2^{(h-1)a+b}P$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$k_{vb-1=a-1}$	$\cdots$	$k_{(v-1)b+1}$	$k_{(v-1)b}$	$2^{(v-1)b}P$
$k_{a+vb-1=2a-1}$	$\cdots$	$k_{a+(v-1)b+1}$	$k_{a+(v-1)b}$	$2^{a+(v-1)b}P$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$k_{(h-1)a+vb-1=ah-1}$	$\cdots$	$k_{(h-1)a+(v-1)b+1}$	$k_{(h-1)a+(v-1)b}$	$2^{(h-1)a+(v-1)b}P$

Figure 3: Arrangement of multiplier k for the fixed-point comb scalar multiplication algorithm

The fixed-point comb scalar multiplication algorithms makes use of a two dimensional precomputation table consisting of v rows and $2^h - 1$ columns. Each entry in the table is a precomputed point. There is one row in the table for each of the two dimensional arrays in Figure 2.1 (v two dimensional arrays). There is also one column in the table for each binary combination of the h-tuple formed by the digits in the columns of the two-dimensional arrays excluding the h-tuple containing only zeros ($2^h - 1$ columns).

A precomputation table entry in Algorithm 8 is denoted G[*array-index*][*entry-index*], where array index is an integer in the range $[0, v)$ that refers to one of the v two dimensional arrays in Figure 2.4 and where the entry index is an integer in the range $[1, 2^h)$ that refers to a column in the table corresponding to the binary representation of an h-tuple containing digits of k.

In Algorithm 8, a column is pointed to by $I_{s,r} = \sum_{t=0}^{h-1}(k_{ta+bs+r}2^t)$, where s specifies a two-dimensional array and r specifies a column in it. Note that the entry $G[s,\ I_{s,r}]$ in the lookup table contains the point $\sum_{t=0}^{h-1}(k_{ta+bs+r}2^{ta+bs+r})$. A sample precomputation table is shown in Figure 2.2.

The first main step of the fixed-point comb scalar multiplication algorithm is the off-line computation of the precomputation table (Steps 1-2.2.1).

The second main step is the point multiplication process (Steps 4-4.2.2.1).

$s \setminus I_{s,r}$	$2^h - 1$	...	2	1
0	$\sum_{i=0}^{h-1} 2^{ia} P$	...	$2^a P$	P
1	$\sum_{i=0}^{h-1} 2^{ia+b} P$	...	$2^{a+b} P$	$2^b P$
...	...	...	...	...
$v-1$	$\sum_{i=0}^{h-1} 2^{ia+(v-1)b} P$	...	$2^{a+(v-1)b} P$	$2^{(v-1)b} P$

Figure 4: $G[s,\ I_{s,r}]$ precomputation table for the fixed-point comb scalar multiplication algorithm

This is an iterative process consisting of b steps, each of which consists of v sub-steps. In each sub-step, a precomputed point corresponding to a column in one of the v two- dimensional arrays is added to an accumulated point. The v sub-steps adds point corresponding to the same column in each of the v two-dimensional arrays. In each main step, the accumulated point is doubled and to it is added the point computed by the v sub-steps (this is analogous to the double-and-add point multiplication algorithm).

Assuming off-line precomputation, the fixed-point comb scalar multiplication algorithm requires, on average, $b-1$ point doubles and $a-1$ point additions. This algorithm also requires the storage of $v(2^h - 1)$ points.

5.2.3 Comparisons between fixed-point scalar multiplication algorithms

Table 2 summarizes the complexity and the storage requirements of the fixed-point scalar multiplication algorithms. It is clear from Table 2 that the fixed-point windowing scalar multiplication algorithm does not require point doubles. The number of point additions required by this algorithm exhibit growth similar to that of the generic scalar multiplication algorithms that use precomputations. It is also clear that the memory requirement of the fixed-point windowing scalar multiplication algorithm grows inversely proportional with the window size.

The processing time of the fixed-comb scalar multiplication algorithm is controlled by the parameters a and b, which are under user control. The number of point additions is ruled by $a \approx m/h$ and the number of point doubles is ruled by $b = (m/h)/v$. To reduce the number of point additions and point doubles, one will choose a large value for h. To further reduce the

Algorithm 8 Fixed-Point Comb Scalar Multiplication Algorithm

Inputs:
P, k
h - number of blocks in which k is divided, also the number of rows in the precomputation matrix
a - width of the blocks
v - number of sub-blocks in which each block is further subdivided
b - width of the sub-blocks
Output: kP
Off-Line precomputation:
1. for $i = 0$ to $h - 1$ do
   1.1. $P_i = 2^{ai}P$
   end for
Compute precomputation array:
2. for i=1 to $2^h - 1$ do
   2.1. $G[0][i] = \sum_{j=0}^{h-1} i_j P, where i = \sum_{j=0}^{h-1} i_j 2^j,\ i_j \in [0, 1]$
   2.2. for $i = 1$ to $v - 1$ do
      2.2.1. $G[j][i] = 2^{bj} G[0][i]$
      end for
   end for
Initialization:
3. $A = \mathcal{O}$
Scalar Multiplication:
4. for $r = b - 1$ down to 0 do
   4.1. $A = 2A$
   4.2. for $s = v - 1$ down to 0 do
      4.2.1 $I_{s,r} = \sum_{t=0}^{h-1} k_{at+bs+r} 2^i$
      4.2.2 if $I_{s,r} \neq 0$ then
         4.2.2.1. $A = A + G[s][I_{s,r}]$
      end for
   end for
5. return(A)

Algorithm	Average Complexity ($\log_2 k \approx m$)		Storage Requirments
	# point double	# point add	# points
fixed-point windowing	0	$2^{w-1} + m/w$	m/w
fixed-point comb ($m = ah,\ a = vb$)	b	a	$v(2^h - 1)$

Table 2: Complexity of fixed-point scalar multiplication algorithms

number of point doubles, one will choose a large value for v. The choice is likely to be restricted by the memory requirements. The memory required grows exponentially with h and linearly with v.

An advantage of the fixed-comb scalar multiplication algorithm over the fixed-point windowing scalar multiplication algorithm is that the user has full control of the processing time and memory requirements, which allows to make best use of the available resources.

6 Conclusion

In this report we presented a brief survey on secret key and public key cryptography algorithms. Secret key cryptography includes both block ciphers and stream ciphers. While MDC and MACs are used for integrity and authentication respectively. Public key cryptography includes RSA, ElGamal and ECC. ECC use short key length (only 160 bits) without losing the same secutiy levels reached by traditional public key cryptosystems.

In this work we also presented a brief introduction to ECC on $GF(2^m)$ which is more suitable and efficient in hardware implementation specially with normal basis representation. Finally, this report presented a survey on different scalar multiplication algorithms.

Future work includes a survey on side channel attacks (SCA) and ECC. Also, a survey on existing ECC implementations on FPGAs will be covered. This will be followed by new algorithms development and implementations.

Acknowledgment

The authors would like to acknowledge the support of KACST for the Project No. AT-22-17. The authors would like also to acknowledge the support of King Fahd University of Petroleum & Minerals.

References

[1] J. Daemen and V. Rijmen. The Design of Rijndael. Number ISBN 3-540-42580-2 in Information Security and Cryptography. Springer, 2002.

[2] FIPS 46-3. Data Encryption Standard. Federal Information Processing Standard (FIPS), Publication 46-3, National Bureau of Standards, U.S. Department of Commerce, October reaffirmed 1999.

[3] FIPS 197. Advanced Encryption Standard. Federal Information Processing Standard (FIPS), Publication 197, Institute of Standards and Technology, U.S. Department of Commerce, November 2001.

[4] FIPS 81. DES Modes of Operation. Federal Information Processing Standard (FIPS), Publication 81, National Bureau of Standards, U.S. Department of Commerce, December 1980.

[5] SP 800-38A. Recommendation for Block Cipher Modes of Operation - Methods and Techniques, National Bureau of Standards, U.S. Department of Commerce, December 2001.

[6] Bruce Schneier. Applied Cryptography. New York: Wiley 1996.

[7] FIPS 180-2. Secure Hash Standard. Federal Information Processing Standard (FIPS), Publication 180-2, National Institute of Standards and Technology, US Department of Commerce, February 2003.

[8] FIPS 113. Computer Data Authentication. Federal Information Processing Standard (FIPS), Publication 113, National Bureau of Standards, U.S. Department of Commerce, May 1985.

[9] FIPS 198. The Keyed-Hash Message Authentication Code (HMAC). Federal Information Processing Standard (FIPS), Publication 198, National Bureau of Standards, U.S. Department of Commerce, March 2002.

[10] 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, 1978.

[11] T. El Gamal. A Public-Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. Advances in Cryptology: Proceedings of CRYPTO 84, Springer Verlag, pp. 10-18, 1985.

[12] N. Koblitz. Elliptic curve cryptosystems. Mathematics of Computation, 48, pp. 203-209, 1987.

[13] ANSI X9.62 - 1998, Public Key Cryptography for the Financial Services Industry: Curve Digital Signature Algorithm (ECDSA), 1998.

[14] IEEE P1363, IEEE Standard Specifications for Public-Key Cryptography, 2000.

[15] National Institute of Standards and Technology, Recommended Elliptic Curves for Federal Government Use, Appendix to FIPS 186-2, 2000.

[16] Standards for Efficient Cryptography Group/Certicom Research, SEC 1: Elliptic Curve Cryptography, Version 1.0, 2000. http://www.secg.org/

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

[18] Wireless Application Protocol (WAP) Forum, Wireless Transport Layer Security (WTLS) Specification. http://www.wapforum.org/

[19] FIPS 186-2. Digital Signature Standard (DSS). Federal Information Processing Standards Publication 186-2, U.S. Department of Commerce/N.I.S.T. National Institute of Standards and Technology, January 2000.

[20] W. Diffie, M. E. Hellman. New directions in cryptography. IEEE Trans. Inform. Theory, IT-22, pp. 644-654, November 1976.

[21] R. C. Mullin, I. M. Onyszchuk, S. A. Vanstone, and R. M. Wilson, "Optimal normal bases in $GF(p^m)$," Discrete Appl. Math., vol. 22, pp. 149-161, 1988/1989.

[22] T. Itoh and S. Tsujii, "A fast algorithm for computing multiplicative inverses in $GF(2^m)$ using normal bases," Info. Comput., vol. 78, no.3, pp. 171-177, 1988.

[23] A. J. Menezes, "Elliptic Curve Public Key Cryptosystems", Kluwer Academic Publishers, 1993.

[24] Cohen, H., Ono, T., and Miyaji, A. "Efficient elliptic curve exponentiation using mixed coordinates", Advances in Cryptology ASIACRYPT '98 (1998), K. Ohta nd D. Pei, Eds., vol. 1514 of Lecture Notes in Computer Science, pp. 51-65.

[25] K. Koyama and Y. Tsutuoka, "Speeding up elliptic cryptosystems by using signed binary window method", Advances in Cryptology Proc. Of Crypto '92, Lecture Notes in Computer Science, 740 (1993), Springer-Verlag, pp. 345-357.

[26] H. Cohen, A. Miyaji and T. Ono, "Efficient elliptic curve exponentiation", Advances in Cryptology-Proc. Of ICICS '97, Lecture Notes in Computer Science, 1334 (1997), Springer-Verlag, pp. 282-290.

[27] J. Lopez and R. Dahab, "Improved Algorithms for Elliptic Curve Arithmetic in $GF(2^n)$", SAC'98, LNCS Springer Verlag, 1998.

[28] D. Gordon, "A Survey of Fast Exponentiation Methods", Journal of Algorithms, 1998, pp. 129-146.

[29] J. Pollard. Monte Carlo methods for index computation mod p. Mathematics of Computation, 32:918-924, 1978.

[30] J. Guajardo and C. Paar. "Efficient algorithms for elliptic curve cryptosystems", Advances in Cryptology - CRYPTO '97 (LNCS 1294), pp. 342-356. Springer-Verlag, 1997.

[31] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. Handbook of Applied Cryptography. CRC Press, 1997.

[32] D. Hankerson, J. Lopez, and A. Menezes. Software implementation of elliptic curve cryptography over binary fields. In Cryptographic Hardware and Embedded Systems - CHES '00 (LNCS 1965), pages 1-24. Springer-Verlag, 2000.

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

[34] J. Lopez and R. Dahab. An overview of elliptic curve cryptography. Technical Report IC-00-10, Institute of Computing, State University of Campinas, Campinas, Sao Paulo, Brazil, May 2000.

[35] J. Solinas. Improved algorithms for arithmetic on anomalous binary curves. Technical Report CORR-46, University of Waterloo, 1999.

[36] Marc Joye and Christophe Tymen, "Compact Encoding of Non-Adjacent Forms with Applications to Elliptic Curve Cryptography", Public Key Cryptography, vol. 1992 of Lecture Notes, in Computer Science, pp. 353-364, Springer-Verlag, 2001.

[37] J. Lopez and R. Dahab. Fast multiplication on elliptic curves over $GF(2^m)$ without precomputation. In Cryptographic Hardware and Embedded Systems - CHES '99 (LNCS 1717), pages 316-327. Springer-Verlag, 1999.

[38] Brickell, E., Gordon, D., McCurley, K. and Wilson, D., "Fast exponentiation with precomputation", Proc. Eurorypt'92, Balatonfured, Hungary 1992.

[39] C. Lim and P. Lee, "More Flexibility Exponentiation with Precomputation", Advances in Cryptology - Crypto '94, LNCS 839, 1994, 95-107.