AN ABSTRACT OF THE THESIS OF

Jaewon Yoo for the degree of Master of Science in Electrical and Computer Engineering presented on June 4, 2008.

Title: Side Channel Attack Resistant Elliptic Curves Cryptosystem on Multi-cores for Power Efficiency

Abstract approved: _____

Ben Lee

The Advent of multi-cores allows programs to be executed much faster than before. Cryptoalgorithms use long-bit words thus parallelizing these operations on multi-cores will achieve significant performance improvement. However, not all long-bit word operations in cryptosystems are suitable for parallel execution on multi-cores. In particular, long-bit words used in Elliptic Curves Cryptography (ECC) do not efficiently divide by the system word size. This causes some of the cores to be idle, which makes it vulnerable for attackers to guess how many operations occurred and thus what field size is being used.

Multiplication is the most important part of public key cryptosystems. Long-bit word multiplication operations are needed for encryption and decryption. J. Fan et al. proposed using Montgomery multiplication on multi-cores using GF(2^{256}) [25, 26], which is suitable for computer systems with 16-bit or 32-bit word size. Fan's Montgomery multiplication is suitable for most RSA. However, in ECC, some GFs will cause idle cores. For example, suppose GF(2^{131}) is used (which is one of the recommended word size by NIST) on a quad-core with a 32-bit word size, which requires $\lceil 132/32 \rceil = 5$ iterations with the last iteration requiring just a 3-bit operation. This cause three of the cores to be idle during this time causing needless power consumption. The most general and the easiest way to make side channel attacks difficult is to insert dummy in-

structions to cover the idle processors. However, dummy instructions result in extra workloads that lead to performance degradation and increases in power consumption.

In this thesis, we will present a multiplier adjuster technique to improve the execution time and the power consumption for the last unbalanced iteration. By appropriately applying dummy instructions between point-addition and point-doubling operations, a balanced point operation can be achieved in ECC. The performance and power-efficiency of the proposed method on multi-cores are analyzed for each GF used in ECC.

Side Channel Attack Resistant Elliptic Curves Cryptosystem on Multi-cores for Power Efficiency

by
Jaewon Yoo

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented June 4, 2008
Commencement June 2009

Master of Science thesis of Jaewon Yoo presented on June 4, 2008

APPROVED:

Major Professor, representing Electrical and Computer Engineering

Director of the School of Electrical Engineering and Computer Science

Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Jaewon Yoo, Author

ACKNOWLEDGEMENTS

Thanks God!

I wish to express my gratitude to Professor Ben Lee, my advisor, for his encouragement, insightful guidance and patience throughout my MS degree at OSU.

I also want to extend great appreciation to Professor Huaiping Liu, Traylor Roger, and John A. Nairn for their serving in my graduate committee.

I want to thank my country Korea, Korean Air Forces, and Defense Security Command for giving chance to study and fully supporting me in USA. In Specially, DSC and 2nd Service Department for patience about my studies for the last two years.

Finally, I express my deepest appreciation and love to my father, Youngkun Yoo, and my mother, Yangsun Oh, who have been encouraging, caring, and praying for my whole life. I also express my thanks to my God parent, Hubert and Maris Böwer, and his family. In special, I would like to express my deepest gratitude to my wife, Hyeking Jeon who has shown endless patience and love. She supported me fully with victimizing her important period in her life. I thank very much my daughter, Siyeon Yoo for well-breeding with health and happiness.

Thanks Everybody

Jaewon Yoo

TABLE OF CONTENTS

Page

TABLE OF CONTENTS (Continued)

Page

TABLE OF CONTENTS (Continued)

Page

LIST OF FIGURES

Figure Page

LIST OF FIGURES (Continued)

Figure Page

LIST OF FIGURES (Continued)

Figure Page

LIST OF TABLES

Side Channel Attack Resistant Elliptic Curves Cryptosystem on Multi-cores for Power Efficiency

1. Introduction

Information security has always been an important challenge in our society. Information processing, especially in governments and businesses, requires systems to be secure. Protection of critical information has usually been accomplished using *cryptography*. Cryptography is the science of hiding data from eavesdroppers. On the other hand, eavesdroppers want to find data from systems that use cryptography, which is known as *cryptanalysis* or *attack*. Cryptography and cryptanalysis have an antagonistic relationship. One side hides data, while the other side attacks it. In the age of computers, cryptography has been integrated into computer systems. Accordingly, attacks have been accomplished by observing the computer system.

A cryptosystem is also referred to as a *cipher*. For example, one of the well-known crypto-algorithms can be found in Roman Emperor Caesar's letters for his staff generals. Caesar encrypted his orders by shifting the letters some fixed number of positions further down the alphabet so that only the generals who knew the number of shifts could understand them. In the Second World War, German's used a cipher called Enigma to hide messages from the allied forces. With the advent of the Internet, the importance of privacy and security of personal information has increased. Therefore, people began to use various crypto-algorithms in their computers, hand-held devices, smart cards, etc.

The efforts to hide information have also attracted the efforts to break the cryptosystem. The essence of cryptosystems has traditionally been considered as a kind of a mathematical problem. Making a crypto-algorithm impossible to break is the key to security. Even mathematically immune crypto-algorithms have vulnerability in practical devices, i.e., real computer systems. Ideally, a cryptosystem generates an output with two inputs: *plaintext* and *key*. However, real systems have other inputs and output, such as voltage, current, power consumption and electromagnetic emission. These parameters have been ignored in traditional cryptosystems. However, the concept of a *side channel* has been recently introduced to include physical characteristics of cryptosystems. A side channel is another source of information about the plaintext and key. The cryptosystem can be broken using this side channel information, which is called a *Side-Channel Attack* (SCA). One general method of SCA is performing power and timing analysis, since different operations of a cryptosystem generate different power consumption and execution time characteristics.

In traditional cryptosystems, the fastest cipher was considered the best cipher. However, in 1996, the vulnerability of cryptosystems to SCAs was exposed by Kocher [14] who extracted

the key value by monitoring the execution time of a cipher. Since then protection from SCA has become one of the major issue in the design of cryptosystems. The general techniques for protecting against SCAs involve masking at the gate-level, inserting dummy instructions, and using regular behaving algorithms [16, 19, 37]. However, most of these research efforts have been performed on a single-core processor. A multi-core processor is a single-chip multiprocessor that contains two or more processors have been attached for enhanced performance, reduced power consumption, and more efficient simultaneous processing of multiple tasks. Multi-core set-up is somewhat comparable to having multiple, separate processors installed in the same computer, but because the two processors are actually merged into the same chip, the connection between them is faster. Ideally, e.g. a dual-core processor is nearly twice as powerful as a single-core processor, and quad-core is four times better than single-core. However, multi-core processors present new software challenges that must be overcome to fully take advantage of processing capabilities to get twice the performance or even more than a single-core processor. Conventional programs do not take into account the capability of multiprocessing, especially in personnel computers or mobile computing devices. Software developers thought that parallel programming is only needed in large companies that require parallel data processing. Thus, small programs, including cryptography programs, used in PCs or mobile devices have been designed for single-core systems.

With the advent of multi-cores, a number of parallel multiplication algorithms have been developed [25, 26]. However, crypto-algorithms use long keywords, e.g., one of the key sizes in ECC is 163-bit. This requires a 163-bit multiplication on multi-cores, which causes uneven distribution of the load. For example, consider an eight-core system with 32-bit processors. Then, 163-bit divided by 32-bit generates six iterations that can be distributed across the cores. However, this causes two of the cores to be completely idle making it vulnerable to SCAs. Therefore, this thesis analyzes the performance and power characteristics of running crypto-algorithms on multi-cores, and develops methodologies to improve the energy-delay product and at the same time protect from SCAs. We proposed arranging the main operation of crypto-algorithm to make it balanced and inserted dummy instructions to fill the gap which is for not accessed data in normal operations. Also, to reduce the overhead from the dummy instructions, we applied a *multiplier adjuster* for the last iteration in a multiplication operation. Rearrangement of the operation and dummy instructions increased runtime by 7.5% and the energy-delay product (EDP) by 1.5%

on average. This increase reduced to at most a 1.1% and 2.7% overhead for runtime and EDP, respectively.

The thesis organized as follows: Section 2 presents a brief review of some typical cryptosystems with examples: *Data Standard Encryption* (DES), *Rivest-Shamir-Adleman Scheme* (RSA), *Elliptic Curves Cryptography* (ECC). Section 3 presents the background on mathematical tools used in cryptosystems that include *Galois Field* (GF), basic operations in GF, the addition chain and multiplication algorithm. Then a Power model and parallel methods are reviewed. Section 4 introduces the Side channel attack method and defense techniques. In Section 5, related works for this thesis that include multiplication operations on a multi core and projective coordinates. Section 6 shows the proposed method of parallelizing multiplication, inserting dummy instruction and using multiplier adjuster. Section 7 shows the simulation environment, simulation methods, and results for the proposed method. Finally, conclusions are drawn in Section 8.

2. Cryptography

According to Wikipedia, *Cryptography* is the practice and study of hiding information. Until recently, cryptography almost referred exclusively to *encryption*, which is the process of converting ordinary information, called *plaintext*, into meaningless garbage information, called *ciphertext*. Decryption is the reverse process that converts ciphertext to plaintext. The pair of algorithms that performs this encryption and decryption is called a *cipher*, or cryptosystem. The detailed operations of cryptosystems are controlled by both the algorithm and a parameter called a *key*. This is a secret key for a specific message exchange context, and is ideally known only to the communicants.

Cryptosystems can be divided into three groups depending on the types of keys: *Symmetric key* cryptosystem, *public key* (also known as asymmetric key) cryptosystems, and *hybrid* cryptosystems. The first method uses the same key for both encryption and decryption. The second method uses different keys for encryption and decryption. The last method uses the combination of the previous two cryptographic schemes.

The following subsections describe the characteristics of the three algorithms.

2.1. Symmetric Key Cryptography

Until the 1970s, all ciphers were based on *Symmetric Key Cryptography* (*SKC*). The basic idea was simple – a cipher generates a ciphertext based on an input message and the key. Then, the delivered ciphertext with the same key recovers the original message.

Figure 1 illustrates an example SKC. Suppose Alice wants to communicate securely with Bob. First, Alice and Bob have to meet and share a key to use in the cipher. Then, Alice generates a ciphertext with the message and the shared key and sends it to Bob. When Bob receives the ciphertext, Alice's message is reproduced with the shared key.

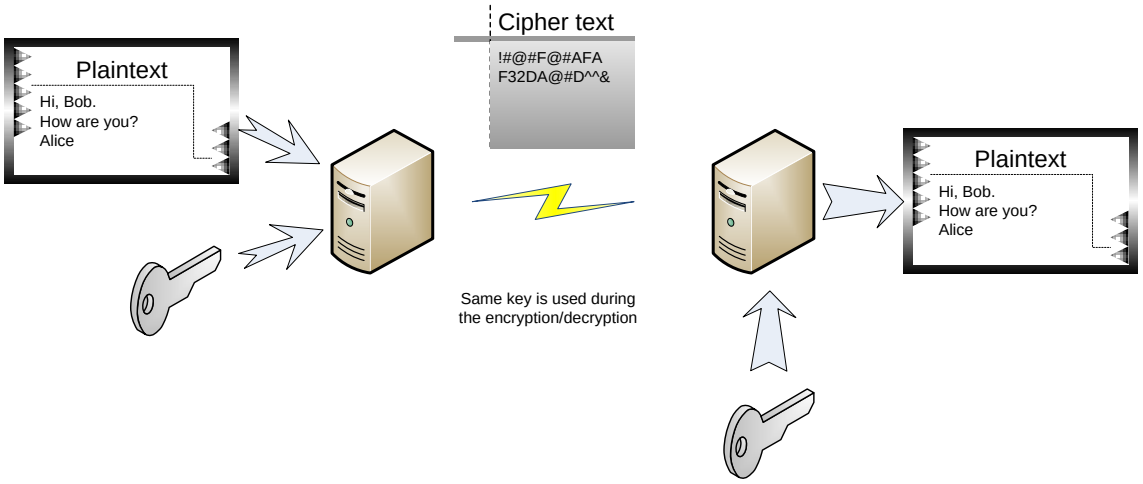


Figure 1: Symmetric Key Cryptosystem

2.2. Public Key Cryptography

In SKC, anyone who wants to communicate must share the same key in a secure way; e.g., face-to-face or through a trusted courier. Furthermore, frequent key exchanges are usually required to maintain secure communications. In addition, each user who wants to perform cryptography must have keys for all other users, which makes key management and distribution a major problem. Before World War II, this was not a big problem because only a few people in Government agencies and the military used SKC. Therefore, key management and distribution was not an issue.

However, with advances in technology, the need for secure communication in our society has become essential. Therefore, key management and distribution has become a major challenge for researchers. In order to reduce the side effects of using symmetric keys, Diffie and Hellman introduced the concept of *public key cryptography* (*PKC*) in 1976 [12]. The concept of public key is to use two different keys during encryption and decryption, called a *public key* and a *private key*. Thus, a user who wants to use this scheme needs to notify to others his public key.

Figure 2 shows an example where Alice wants to send a message to Bob using PKC. Alice first acquires Bob's public key. Then, the message is encrypted to ciphertext using the obtained key and sent to Bob. When Bob receives the ciphertext, it is decrypted to plaintext using Bob's own private key.

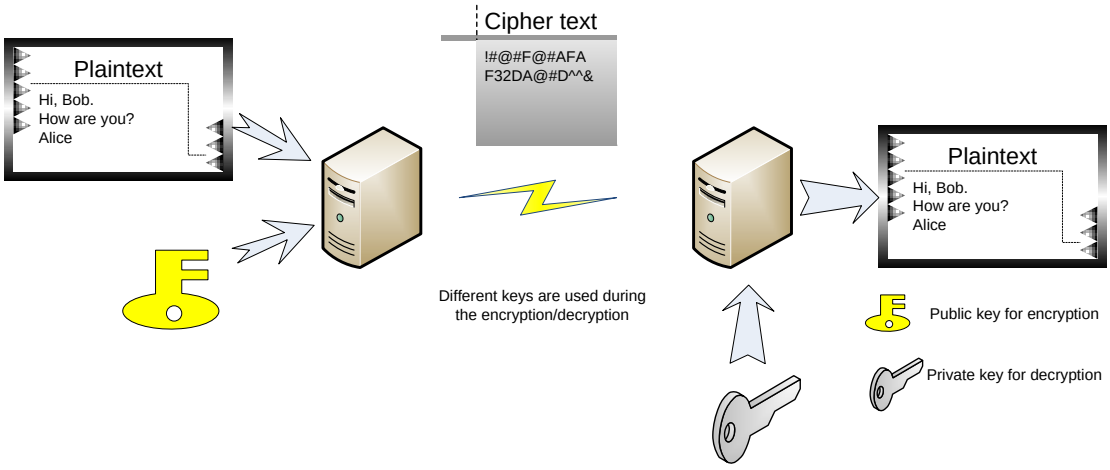


Figure 2: Public Key Cryptosystem

2.3. Hybrid Cryptosystem

Both SKC and PKC have advantages and disadvantages. SKC has a high security level and is easy to implement, but key distribution and management is a problem. PKC solves the aforementioned issue in SKC, but has a big shortcoming – the level of security for PKC compared with SKC of the same key size is lower. *Hybrid Cryptosystems* have been introduced to overcome this weakness by implementing key distribution/management as in PKC and encryption/decryption process based on SKC.

Figure 3 shows an example of a hybrid scheme. First, Alice and Bob generate their own key, called a *session key*, using the PKC scheme. They then exploit the session key with the SKC scheme.

Each user, Alice and Bob, has his or her own private key as same as we saw in the PKC. They need a secret key to operate the system. The secret key must be shared by two parties, i.e., Alice and Bob. However, the secret key is created by the PKC. This key which is created by PKC is called a *session key* in contrast with the normal secret key. Alice gets Bob's public key and then mixes it up with her own private key. Bob also performs the same process with Alice's public key. These key is ideally the same: $K_{pri\_alice} * (K_{pri\_bob} * Base) = K_{pri\_bob} * (K_{pri\_alice} * Base)$ where K_{pri_alice} and K_{pri_bob} are their own private key, Base and * are the shared information and operation, i.e., Base is basic point and * is scalar multiplication in ECC.

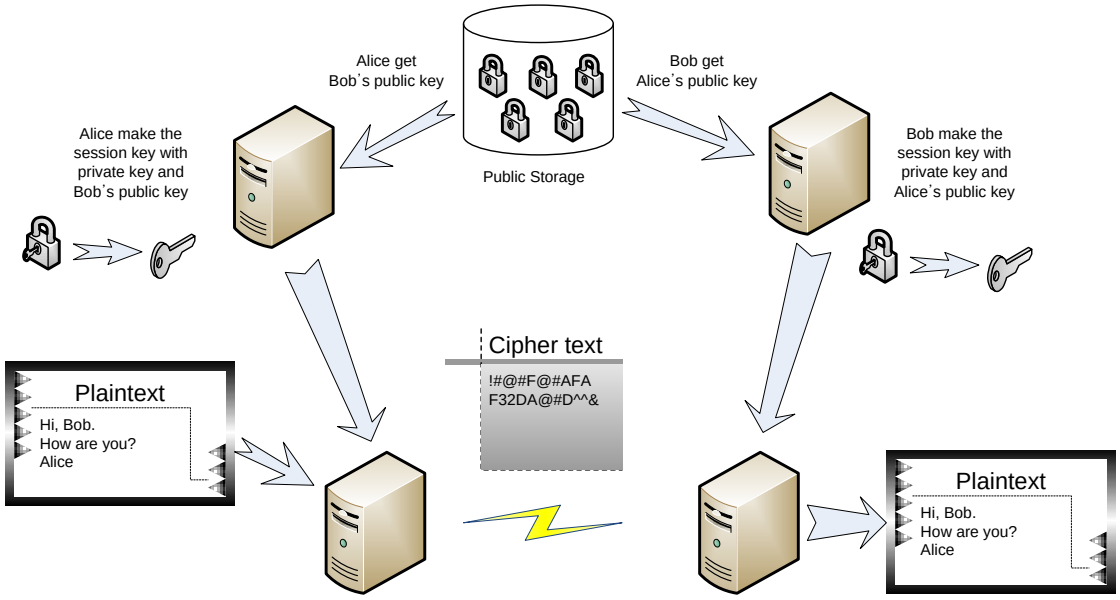


Figure 3: Hybrid Cryptosystem

2.4. Example of Crypto-algorithm

There have been many cryptographic algorithms developed since humans started hiding information. After WWII, most cryptographic applications have been developed for computing environments. This section overviews three modern crypto-algorithms; *Data Encryption Standard* (*DES*), *Rivest-Shamir-Adleman* (*RSA*) scheme, and *Elliptic Curve Cryptography* (*ECC*).

DES is a typical SKC system, while RSA and ECC represent PKC systems.

2.4.1. Data Encryption Scheme (DES)

A team at IBM proposed the DES algorithm [1] in 1975 as a response to the government's needs for secured protection of classified and other sensitive information. DES was adopted in 1977 and became the most widely used encryption scheme. In DES, data is encrypted in 64-bit blocks using a 56-bit key. A plaintext is first divided into 64-bit blocks, and then the blocks are changed bit-by-bit using permutation and substitution, based on subkeys produced from the 56-bit key.

The overall architecture of DES is shown in Figure 4, which accepts a 64-bit block of *plaintext* and a 56-bit *key*. The processing of the plaintext proceeds in three phases; Initial Permutation (IP), 16-Round processing, and Inverse IP. The right-hand portion of Figure 4 shows the way in which the 56-bit key is used. Initially, the 56-bit key is passed through a permutation function, which changes the position of the bits. Then, for each of the 16 rounds, a 48-bit *subkey*

or *roundkey* K_i is produced by the combination of a left circular shift and a permutation [2, 10, 11]. Based on this, each round obtains a different subkey.

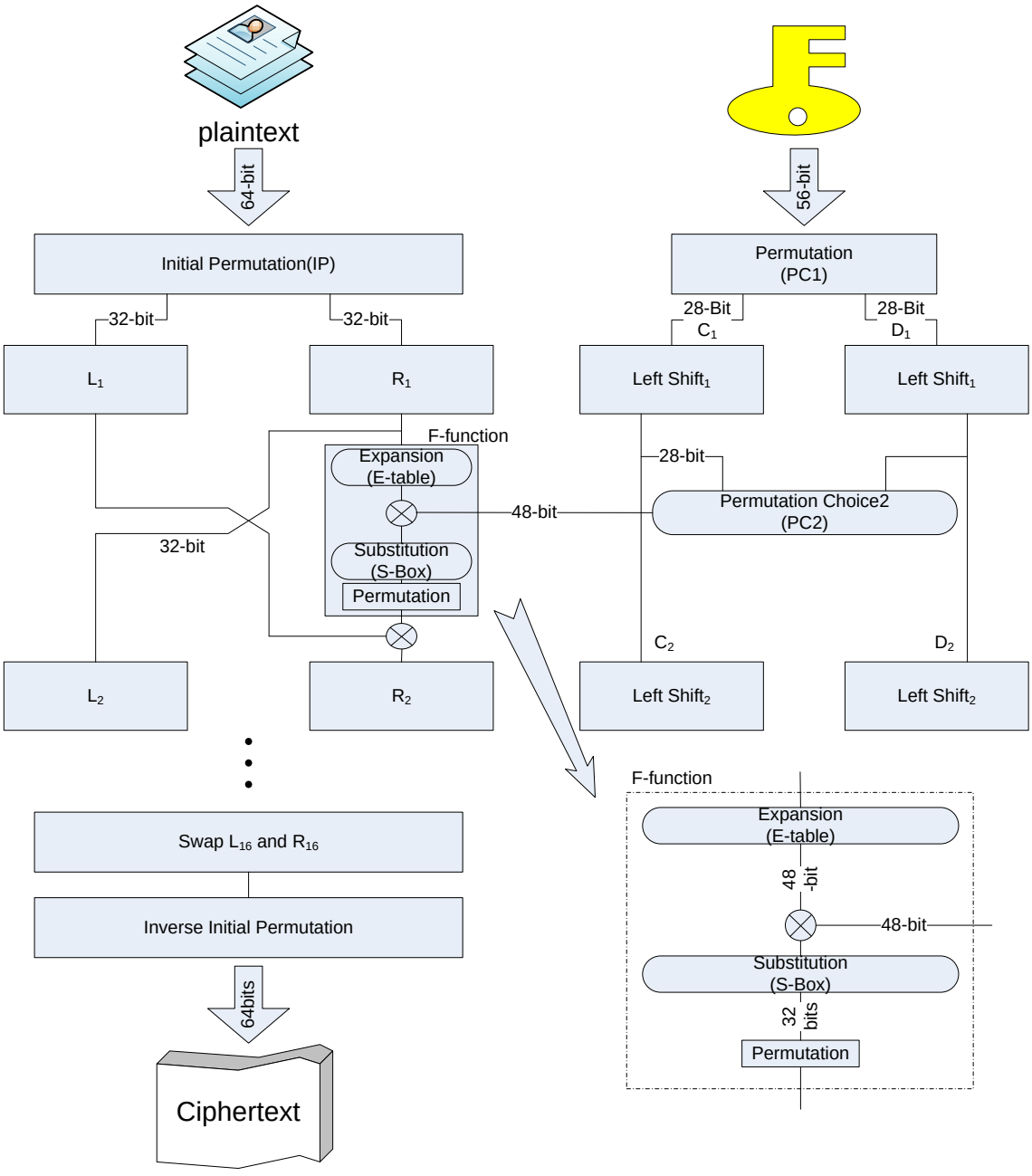


Figure 4: Architecture of DES

Figure 4 also shows the internal structure of a single round. The left (L_{i-1}) and right (R_{i-1}) halves of each 64-bit intermediate value are treated as separate 32-bit quantities. The processing requirement for each round *i* can be summarized as the following formulas:

$$L_i = R_{i-1}$$
$$R_i = L_{i-1} \oplus F(R_{i-1}, K_i)$$

where the subkey K_i is 48 bits and L_{i-1} and R_{i-1} are 32-bit inputs. $R_{i-1}$is then expanded to 48 bits by a predefined extended permutation table (E-table). The extended 48 bits are mixed with the sub-key by exclusive-OR and substituted with 32 bits by S-box. Finally, by passing the permutation table, the substituted 32 bit positions are changed into the next L_i after an exclusive-OR operation with L_{i-1}

The 48-bit subkey is produced by using the 56-bit key as an input and then applying Permuted Choice1 (PC1), circular left shift (LS), and Permuted Choice 2 (PC2). C_{i-1} and D_{i-1} are 28 bits of the left and right sides of the 56-bit key. PC2 chooses 48 bits as a subkey with $C_i=LS(C_{i-1})$ and $D_i=LS(D_{i-1})$. Afterwards, C_i and D_i are used as the inputs for the next round processing.

Simply DES gets a 64-bit data, and then divides it into two parts. One of them is extended using E-table and mixed with the subkey. After this, it is reduced by S-box, and finally its bit position is interchanged. This process repeats 16 times in DES machine.

Suppose we have a simple permutation table such as '2 5 7 8 4 6 3 1', which numbers represent the bit position, instead of a real permutation table since real permutation tables require at least 32 bits. Any input character in the American Standard Code for Information Interchange (ASCII) format can be changed by the permutation table. For examples, the letter 'A' $(0100\ 0001)_2$ is changed to 'È' $(1001\ 0000)_2$. IP, IP-1, PC and PC2 have the same concept of this simple example. The Extended table(E-table) is also similar to the permutation tables. Figure 5 shows the real E-table and an example in DES. As shown in the Figure, there are extra bits such as the 4th, 5th and 32nd positions.

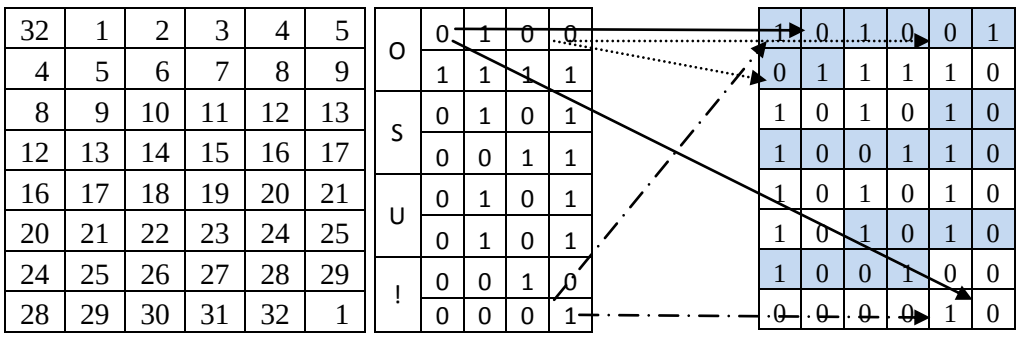


Figure 5: E-table and Example of bits extension

Using the table, the 4 letter word "OSU!", viz., O(0100 1111)$_2$, S(0101 0011)$_2$, U(0101 0101)$_2$, and !(0010 0001)$_2$, is changed to a 48 bit chunk. The extended data chunk is shortened by the Substitution Box. S-box operations are described in Figure 6.

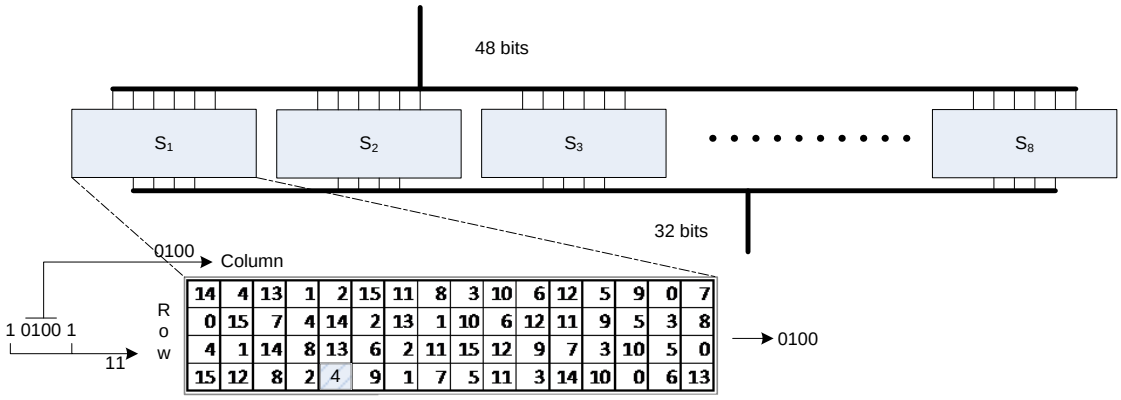


Figure 6: S –box Substitution

S-Box consists of eight substitution tables. The extended 48-bit chunk is divided by 8 and distributed to each substitution table; S_1, S_2, …, S_7, and S_8. The distributed 6 bit chunk is used to choose the substitute data. The 1st and 6th bit are used to select a row and the 2nd to 5th bit are used to select a column as shown in Figure 6. In our example, the first 6 bit chunk $(101001)_2$ from Figure 5 selects $(11)_2$ as a row and $(0100)_2$ as a column.

2.4.2. Rivest, Shamir, and Adleman (RSA) Scheme

In 1976, Diffie and Hellman demonstrated the concept of an algorithm where two parties can communicate without sharing the key [3, 12]. The first implementation of the algorithm was proposed by Rivest, Shamir, and Adleman (RSA) at MIT in 1977. The RSA scheme is a block cipher, and the plaintext and ciphertext are integers between 0 and n-1 for some n.

RSA relies on the difficulty of factoring some number and modular operations based on the following equations:

$$C = M^e \bmod n \tag{2.1}$$

$$M = C^d \bmod n = (M^e)^d \bmod n, \tag{2.2}$$

where C is ciphertext, M is plaintext, n is the product of the two prime numbers, and e and d are the public key and the private key, respectively. How e, d, and n are chosen will be discussed shortly.

For example, suppose Alice wants to communicate with Bob. Bob first generates the RSA key pair (e_{Bob}, d_{Bob}). Alice obtains Bob's public key, e_{Bob}, and encrypts the message M using the obtained key e_{Bob} and modulus n. When Bob receives the ciphertext C, the message is decrypted using the private key d_{Bob} and modulus n.

Before performing encryption and decryption, the RSA algorithm requires a setup operation to choose e, d, and n, which is done using the following steps:

1. Choose two distinct prime numbers p and q
2. Compute $n = pq$
3. Compute Euler's totient function of n : $\phi(n) = (p-1)(q-1)$
4. Select the public key e_{Bob} such that $\gcd(e_{Bob}, \phi(n))=1$
5. If $e_{Bob} < 1$ or $e_{Bob} > \phi(n)$, return to step 4.
6. Compute $d_{Bob} = e_{Bob}^{-1} \pmod{\phi(n)}$ using extended Euclid's algorithm.

The security of RSA comes from the difficulty of solving the *Discrete Logarithm Problem* (DLP). In other words, RSA relies on the difficulty of factoring some numbers, which requires multiplication and squaring. Euler's totient function $\phi(n)$ is defined as the number of integers less than n which are relative prime to n. For example, $\phi(6) = 2$ since 1 and 5 are relative prime to 6. Also $\phi(9) = 6$ since 1, 2,4,5,7, and 8 are relative prime to 9.

The equation of step 6 above can be represented as $e_{Bob}d_{Bob} = 1 \pmod{\phi(n)}$ by multiplying the e_{Bob} for both sides. Since $\gcd(e_{Bob}, \phi(n)) = 1$ by step 4, $\gcd(e_{Bob}, \phi(n))$ can be represented as the following equation by Extended Euclid's algorithm which is described in the background work.

$$e_{Bob}x + \phi(n)y = 1 \pmod{\phi(n)} \qquad (2.3)$$

The x and d_{Bob} are identical since $\phi(n)y \pmod{\phi(n)} = 0$. The d_{Bob} is the multiplicative inverse of $e_{Bob} \pmod{\phi(n)}$. Extended Euclid's algorithm is used in finding out x and y from the equation (2.3).

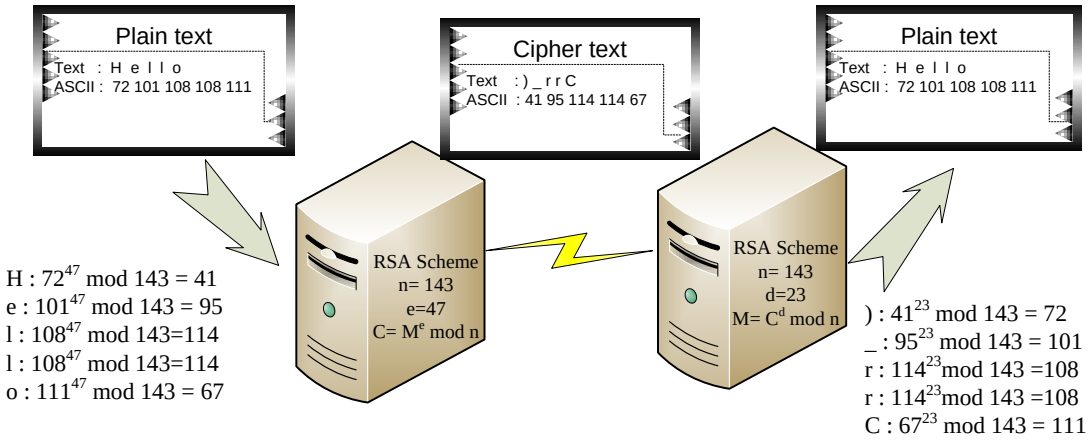
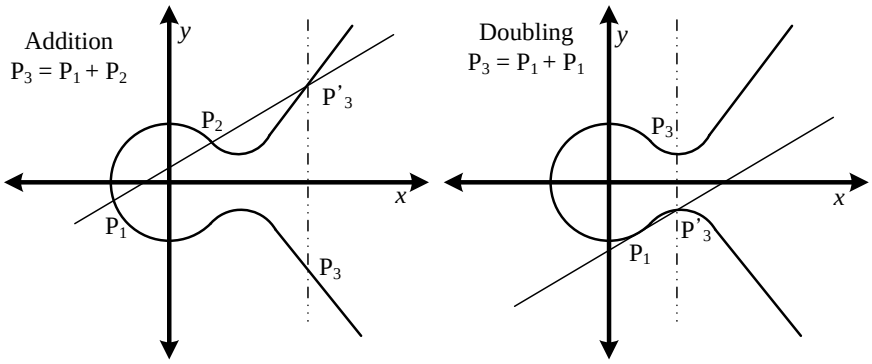


Figure 7: RSA Scheme

Figure 7 shows the RSA scheme. Suppose the encryption process of the sentence "Hello" is performed by RSA. The n is set to 143. Since we choose p = 11 and q =13, as a result $\phi(n)$ will be set to 120. The encryption key and decryption key are 47 and 23, respectively. The first letter 'H' in our example can be represented '72' in ASCII. The number 72 is converted to 41 by RSA algorithm; $41 = 72^{47} \bmod 143$. The same method applied to the other letters. The decryption process is also similar to the encryption process. The encrypted letter '41' is recovered to '72' by the equation; $72 = 41^{23} \bmod 143$. The processes of other encrypted letters are identical.

2.4.3. Elliptic Curve Cryptography (ECC)



$GF(p) \supset d, e$ $a,b,c = 0; p>3; 4d^3+27e^2 \neq 0$	$GF(2^m) \supset c, e$ $a=1; b,d = 0; e \neq 0$
$O + O = O$ $(x,y) + O = (x,y)$ $(x,y) + (x,-y) = O$	$O + O = O$ $(x,y) + O = (x,y)$ $(x,y) + (x,x+y) = O$
Addition over GF(p) $(x_1,y_1) + (x_2,y_2) = (x_3, y_3)$ $\lambda = (y_2-y_1)(x_2-x_1)^{-1}$ $x_3 = \lambda^2-x_1-x_2$ $y_3 = \lambda(x_1-x_3)-y_1$	Addition over GF(2^m) $(x_1,y_1) + (x_2,y_2) = (x_3, y_3)$ $\lambda = (y_2-y_1)(x_2-x_1)^{-1}$ $x_3 = \lambda^2+\lambda+x_1+x_2+c$ $y_3 = \lambda(x_1+x_3)+x_3+y_1$
Doubling over GF(p) $(x_1,y_1) + (x_1,y_1) = (x_3, y_3)$ $\lambda = (3x_1^2+d)(2y_1)^{-1}$ $x_3 = \lambda^2-2x_1$ $y_3 = \lambda(x_1-x_3)-y_1$	Doubling over GF(2^m) $(x_1,y_1) + (x_1,y_1) = (x_3, y_3)$ $\lambda = x_1 + (y_1)(x_1)^{-1}$ $x_3 = \lambda^2 + \lambda + c$ $y_3 = \lambda(x_1+x_3)+x_3+y_1$

Figure 8: Operations in Elliptic curves over $GF(p)$ and $GF(2^m)$

ECC was introduced by Neal Koblitz and Victor S. Millerin in 1985 [6]. ECC relies on the difficulty of performing algebra on an elliptic curve. ECC offers an equal level of security with smaller key sizes than RSA. Thus, the computational requirement is reduced. The security from ECC comes from the difficulty in solving the *Elliptic Curve* DLP (ECDLP) modular random number p. An arbitrary number $n^x (mod\ p)$ has the same result with $n \times n \times \cdots \times n\ (mod\ p)$, i.e., x times multiplication in RSA. Also, arithmetic operation of random point $dP (mod\ p)$ has similar

results with the RSA case. Scalar multiplication of d times point P, dP, gives the identical output as $P + P + \cdots + P$, d times addition in ECC.

In fact, elliptic curves are not ellipses. They are named so because they are described by cubic equations, similar to those used for calculating the circumference of ellipses. In general, cubic equations for elliptic curves take the form

$$y^2 + axy + by = x^3 + cx^2 + dx + e$$

where a, b, c, d, and e are real numbers that satisfy some simple conditions. Also included in the definition of any elliptic curve is a single element denoted as O, called the *point at infinity* or the *zero point*, which is the sum of three points on an elliptic curve that lie on a straight line [2].

There are two major types of elliptic curves in real cryptosystems; *Galois Field* (*GF*) in prime number p, $GF(p)$, and GF in 2^m, $GF(2^m)$. The coefficients of an elliptic curve change depending on the types. They also have different doubling and addition equations. Figure 8 shows the difference of operations between the two types.

As we see in Figure 8, the addition of two points on an elliptic curve always lies somewhere on the curve. Even though the third point as a sum of two points also lie on the elliptic curve, the point addition is not as simple as adding the coordinates of the points. The third point P_3 is defined as follows. A certain point P'_3 in Figure 8 will be met by the straight line - it can be created by two given points - and elliptic curve whether the two points are different or not. P'_3 is reflected across the x-axis. The reflected point also lies on the elliptic curve. This reflected point is P_3 which is the result point of the addition P_1 and P_2. The difference of $GF(p)$ and $GF(2^m)$ comes from distinct algebra operation from each field as we saw in the mathematical section. We will briefly review the Addition and Doubling operations over GF(p).

On the elliptic curves, O plays a role of *Addition Identity*, which acts similar to 0 in general algebra. Since an elliptic curve is symmetric about the x-axis, when we add two distinct points which have the same x value, it does not satisfy the summation sentence as we discussed without *Addition Identity O*. When we add two distinct points with different x values on elliptic curves, the third point will be on the curve. With two points, we can create a linear equation and then find out the third point by interception with the elliptic curves and the lines. Also, in the case of doubling, we can use the tangent of a line by differentiation of the elliptic curves. The same rule also applies to elliptic curves over $GF(2^m)$. For more detail, see [30].

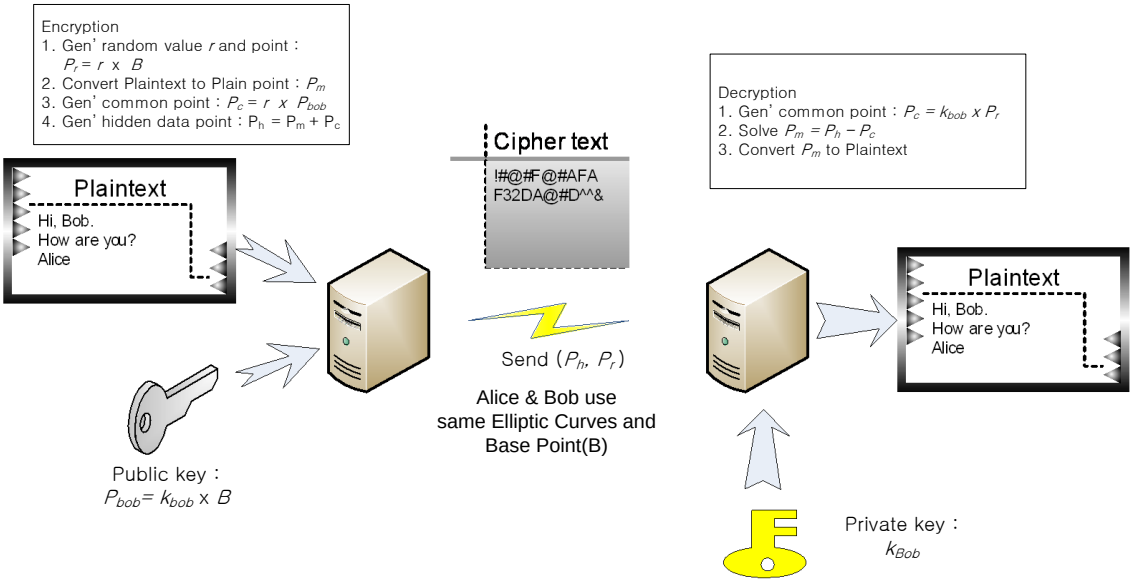


Figure 9: ECC ElGamal Scheme

There are several applications using elliptic curves, but this paper discusses a simple application -the ElGamal scheme in Figure 9. Assume that someone who wants to send a message m has to convert the message into point P_m over the Elliptic curve. The plain point P_m will be encrypted and then decrypted. Each user chooses an arbitrary number k less than n and one point $P=kG$, where k and P are used as a private key and a public key, respectively. Using these keys, plain point P_m can be encrypted as

$$C_m=(rG, P_m + rP_{\text{rcv}}),$$

where r is a random number chosen by the sender and P_{rcv} is the other party's public key. Then, the other party can decrypt the ciphertext C_m to the point P_m by subtracting a common value from:

$$P_{\text{m}} = P_{\text{m}}+rP_{\text{rcv}}–k_{\text{rcv}}rG \text{ (using } rP_{\text{rcv}} = rk_{\text{rcv}}G).$$

Then, the extracted P_m is converted into the message that was originally sent by sender.

3. Background

This section presents a brief overview of the background information necessary to understand the rest of the thesis. First, some mathematical concepts used in cryptosystems are introduced. Then, some basic information on power models are discussed. Finally, methodologies for parallel computing are summarized. For a detailed treatment of these topics, readers are referred to the following: Cryptography [2, 10, 11], Power Model [17], and Side Channel Attack [14, 15]

3.1. Mathematical Concepts Used in Cryptosystem

Cryptosystems cannot be explained without mathematics. Cryptographers often evaluate the security of ciphers by observing their mathematical functions. The fundamental mathematical operations in cryptography consist of addition, subtraction, multiplication, and division with reminder. Using these four mathematical operations, a cryptosystem can change a plaintext into a cipher text and vice versa.

3.1.1. Fundamental Mathematical Operations

Humans use the base-10 number system. However, base-10 is not efficient for computer arithmetic operations, since computers use the base-2 number system. Therefore, a cryptosystem also uses the base-2 number system. However, the number of occurrences of the nonzero integer is not uniform in modular multiplication using the binary system. For example, the results of performing modular multiplications on 3-bit data (i.e., mod 8) are shown in Table 1. As can be seen, there are only four occurrences of 3, but twelve occurrences of 4. Uneven number format is shown to be cryptographically weaker than a uniformly distributed number format [2]. In contrast, *polynomial base* (PB) provides a uniform number format, i.e., the frequency of occurrence of any number is the same.

Table 1: Multiplication modulo 8 and Occurrences

Mult	001(1)	010(2)	011(3)	100(4)	101(5)	110(6)	111(7)
001(1)	001(1)	010(2)	011(3)	100(4)	101(5)	110(6)	111(7)
010(2)	010(2)	100(4)	110(6)	000(0)	010(2)	100(4)	110(6)
011(3)	011(3)	110(6)	001(1)	100(4)	111(7)	010(2)	101(5)
100(4)	100(4)	000(0)	100(4)	000(0)	100(4)	000(0)	100(4)
101(5)	101(5)	010(2)	111(7)	100(4)	001(1)	110(6)	011(3)
110(6)	110(6)	100(4)	010(2)	000(0)	110(6)	100(4)	010(2)
111(7)	111(7)	110(6)	101(5)	100(4)	011(3)	010(2)	001(1)

Integer	1	2	3	4	5	6	7
Occurences	4	8	4	12	4	8	4

A polynomial is a sum of different powers of a variable. For example,

$$x^4 + x^2 + 1 \quad (3.1)$$

$$x^3 + x + 1 \quad (3.2)$$

are polynomials in x. The polynomials (3.1) and (3.2) can be represented in binary format $(10101)_2$ and $(1011)_2$, respectively. A polynomial is not set to any particular constant, i.e., x is not defined. Thus, four operations in PB, i.e., addition, subtraction, multiplication and division, are different from conventional algebra. Before discussing the PB operations, the concepts of fields and group will first be reviewed. This will be followed by a discussion on addition, subtraction, multiplication, and division operations in PB.

Group and Fields

A group is a set of numbers with a set of custom-defined arithmetic operations. The unique rules for arithmetic in groups are the source of the hard problems necessary for cryptographic security. In general, two groups used in cryptosystems are Z_n, the additive group of integers modulo a number n, and Z_p^*, the multiplicative group of integers modulo a prime number p.

The group Z_n operations use only the integers from 0 to $(n - 1)$. Its basic operation is addition, which ends by reducing the result modulo n, i.e., taking the integer remainder when the result is divided by n. One very important feature of arithmetic in a group is that all calculations give numbers that are in the group, which is called closure. The modular reduction by n ensures that all additions result in numbers between 0 and $(n - 1)$.

(11 + 13) mod 15 = 24 mod 15 = 9	(10 * 3) mod 11 = 30 mod 11 = 8
(4 + 11) mod 15 = 15 mod 15 = 0	(4 * 7) mod 11 = 28 mod 11 = 6
(14 + 4) mod 15 = 18 mod 15 = 3	(2 * 8) mod 11 = 16 mod 11 = 5
(9 +10) mod 15 = 19 mod 15 = 4	(9 * 5) mod 11 = 45 mod 11 = 1

Figure 10: Example of Addition in Group Z_{15} and Multiplication in Group Z_{11}^*

The multiplicative group Z_p^* uses only the integers between 1 and $(p - 1)$ and its basic operation is multiplication. A multiplication ends by taking the remainder of a division by a prime number p, which ensures closure.

Figure 10 shows some example additions and multiplications in group Z_{15} and group Z_{11}^*. The additive group Z_{15} uses integers from 0 to 14. If two arbitrary numbers a and b belong to Z_{15}, the result c of its addition also belongs to group Z_{15}, i.e., $(a + b) \bmod 15 = c$. The multiplicative

group Z_{11}^* uses integers from 1 to 10. A multiplication in Z_{11}^* finishes by taking the remainder after the result is divided by 11.

An arithmetic operation is said to be *commutative* if the order of its arguments is insignificant. With ordinary numbers, addition and multiplication are commutative operations; e.g., (2* 9) = (9*2) and (2 + 9) = (9 + 2). However, subtraction and division are not commutative, e.g., (2 – 9) ≠ (9 – 2) and (2 / 9) ≠ (9 / 2).

A group is called *Abelian* if its main operation is commutative. Thus, an additive group is Abelian if $(a + b) = (b + a)$ for all elements a and b in the group. A multiplicative group is Abelian if $(a * b) = (b * a)$ for all elements a and b in the group. The additive group Z_n and the multiplicative group Z_p^* are both Abelian groups.

A *field* is a set of elements with two custom-defined arithmetic operations; most commonly, addition and multiplication. The elements of the field are an additive Abelian group, and the non-zero elements of the field are a multiplicative Abelian group. This means that all elements of the field have an additive inverse, and all non-zero elements have a multiplicative inverse. As is true for groups, other operations can be defined in a field, using its two main operations. A field is called *finite* if it has a finite number of elements. The most commonly used finite fields or *Galois fields* in cryptography are $GF(p)$ (where p is a prime number) and $GF(2^n)$ [28].

Field of *GF*(p)

The $GF(p)$ field consists of numbers from 0 to $(p – 1)$. Its operations are addition and multiplication, which are defined for the groups Z_n and Z_p^*, respectively, and all calculations end with reduction modulo p. The restriction that p be a prime number is necessary so that all non-zero elements have a multiplicative inverse. As with Z_n and Z_p^*, other operations in $GF(p)$ (such as division, subtraction, and exponentiation) are derived from the definitions of addition and multiplication.

Field of GF(2^n)

The $GF(2^n)$ field is attractive for implementation due to their carry-free arithmetic, and the availability of different equivalent representations of the field, which can be implemented and optimized in hardware [28, 29]. There are several ways to describe arithmetic in $GF(2^n)$. *Polynomial basis* (PB) and *optimal normal basis* (ONB) are typical $GF(2^n)$ representation methods. Our proposed method is based on PB, therefore, only the PB representation will be reviewed in this thesis.

A cryptosystem uses various *GF* sizes. For example, ECC usually uses $GF(2^{131})$, $GF(2^{163})$, $GF(2^{193})$, and $GF(2^{233})$, whereas RSA uses $GF(2^{1024})$, $GF(2^{2048})$ and $GF(2^{3072})$. In order to improve security, the field size will become bigger in future.

Polynomial Bases Representation

$GF(2^n)$ has 2^n elements, which are polynomials of degree less than n, with coefficients in $GF(2^n)$, i.e., $\{a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + ... + a_2x^2 + a_1x + a_0 \mid a_i = 0 \text{ or } 1\}$. These elements can be written in vector form as $(a_{n-1} \ldots a_1\ a_0)$. The main operations in $GF(2^n)$ are addition and multiplication. Some computations involve an *irreducible* polynomial $f(x) = x^n + f_{n-1}x^{n-1} + f_{n-2}x^{n-2} + ... + f_2x^2 + f_1x + f_0$, where each f_i is in $GF(2^n)$. The polynomial $f(x)$ cannot be factored into two polynomials over $GF(2^n)$, each of degree less than n. The irreducible polynomial takes the same role as a prime number in the $GF(p)$. The basic arithmetic operations of polynomial bases are shown in Figure 11:

Addition/Subtraction

$(c_{n-1} \ldots c_1\ c_0) = (a_{n-1} \ldots a_1\ a_0) \pm (b_{n-1} \ldots b_1\ b_0)$,

where $c_i = a_i$ xor b_i, $0 \le i \le n\text{-}1$

Division/Modular operation

$(a_{n-1} \ldots a_1\ a_0) = (q_{n-k} \ldots q_0) \times (b_{k-1} \ldots b_1\ b_0) + (r_{k-2} \ldots r_0)$

where $(q_{n-k} \ldots q_0)$ is the quotient, $(r_{k-2} \ldots r_0)$ is the reminder $(n \ge k)$

$(a_{n-1} \ldots a_1\ a_0) \div (b_{k-1} \ldots b_1\ b_0) = (q_{n-k} \ldots q_0)$

$(a_{n-1} \ldots a_1\ a_0) \bmod (b_{k-1} \ldots b_1\ b_0) = (r_{k-2} \ldots r_0)$

Modular Multiplication

$(r_{n-1} \ldots r_1\ r_0) = (a_{n-1} \ldots a_1\ a_0) \times (b_{n-1} \ldots b_1\ b_0) \bmod f(x)$,

where a_i, b_i and $r_i\ (0 \le i \le n\text{-}1)$ is the remainder modulo $f(x)$

$$
\begin{array}{rccccc}
 & a_{n-1} & & \ldots & a_1 & a_0 \\
\times & b_{n-1} & & \ldots & b_1 & b_0 \\
\hline
a_{n-1}b_{n-1} & (a_{n-1}b_{n-2} + a_{n-2}b_{n-1}) & \ldots & & (a_1b_0 + a_0b_1) & a_0b_0 \\
\hline
\bmod & 1 & f_{n-1} & f_{n-2} \ldots & f_1 & f_0 \\
\hline
 & r_{n-1} & & \ldots & r_1 & r_0
\end{array}
$$

Additive Identity/Inverse $(a_{n-1} \ldots a_1\ a_0) + 0 = (a_{n-1} \ldots a_1\ a_0) \bmod f(x)$ 0 is called an *additive identity* if $(a_{n-1} \ldots a_1\ a_0) + (b_{n-1} \ldots b_1\ b_0) = 0 \bmod f(x)$ then $(b_{n-1} \ldots b_1\ b_0)$ is an *additive inverse* of $(a_{n-1} \ldots a_1\ a_0)$
Multiplicative Identity/Inverse $(a_{n-1} \ldots a_1\ a_0) \times 1 = (a_{n-1} \ldots a_1\ a_0) \bmod f(x)$ 1 is called a *multiplicative identity* if $(a_{n-1} \ldots a_1\ a_0) \times (b_{n-1} \ldots b_1\ b_0) = 1 \bmod f(x)$ then $(b_{n-1} \ldots b_1\ b_0)$ is a *multiplicative inverse* of $(a_{n-1} \ldots a_1\ a_0)$

Figure 11: Arithmetic Operations of Polynomial Bases

In order to illustrate PB arithmetic operations, consider the following two polynomial representations

$$x^4 + x^2 + 1 \qquad (a_4\, a_3\, a_2\, a_1\, a_0) = (1\ 0\ 1\ 0\ 1) \tag{3.3}$$

$$x^3 + x^2 + x + 1 \qquad (b_4\, b_3\, b_2\, b_1\, b_0) = (0\ 1\ 1\ 1\ 1), \tag{3.4}$$

where both are elements of $GF(2^5)$. The result of performing addition according to Figure 11 is as follows:

$c_4 = a_4$ xor $b_4 = 1$ xor $0 = 1$ $\qquad$ $c_3 = a_3$ xor $b_3 = 0$ xor $1 = 1$

$c_2 = a_2$ xor $b_2 = 1$ xor $1 = 0$ $\qquad$ $c_1 = a_1$ xor $b_1 = 0$ xor $1 = 1$

$c_0 = a_0$ xor $b_0 = 1$ xor $1 = 0$

$(1\ 1\ 0\ 1\ 0)$ $\qquad$ $x^4 + x^3 + x$

Thus, PB representation $x^4 + x^3 + x$ is the addition of (3.3) and (3.4). In PB representation, subtraction also provides the same results. Multiplication is more difficult than addition/subtraction. The following illustrates the multiplication of the above two polynomials, i.e., $(x^4 + x^2 + 1) \times (x^3 + x^2 + x + 1) \bmod (x^5 + x^3 + 1)$:

$x^4 \times (x^3 + x^2 + x + 1) = x^7 + x^6 + \cancel{x^5} + \cancel{x^4}$

$x^2 \times (x^3 + x^2 + x + 1) = \cancel{x^5} + \cancel{x^4} + \cancel{x^3} + \cancel{x^2}$

$1 \times (x^3 + x^2 + x + 1) = \cancel{x^3} + \cancel{x^2} + x + 1$

	$(x^7 + x^6 + x + 1)$	(11000011)	(3.5)
-	$(x^5 + x^3 + 1) \times x^2$	(10100100)	(3.6)
	$(x^6 + x^5 + x^2 + x + 1)$	(01100111)	(3.7)
-	$(x^5 + x^3 + 1) \times x$	(1010010)	(3.8)
	$(x^5 + x^4 + x^2 + 1)$	(00110101)	(3.9)
-	$(x^5 + x^3 + 1)$	(101001)	(3.10)
	$x^4 + x^3 + x^2$	(11100)	(3.11)

Multiplication is simply a shift and Exclusive-OR as shown in (3.5). However, since multiplication increases the size of the exponent, we need to shorten it to fit into a $GF(2^5)$ word size, i.e., 5 bits. To prevent word size overflow, a modular operation is performed using an irreducible polynomial $x^5 + x^3 + 1$, which is one of the irreducible polynomials in $GF(2^5)$ that has been arbitrarily chosen. The division and modular operation involves a series of shift and Exclusive-OR operations as shown in (3.6) to (3.11) until the reminder becomes an element of $GF(2^5)$. The quotients bits are generated from operations (3.6), (3,8) and (3.10), i.e., $x^2 + x + 1$. The last reminder is the result of modular operation, which is $x^4 + x^3 + x^2$.

The benefit of PB is a uniformly distributed number format system. Back to the 3-bit system, we already saw the binary format is unevenly distributed as shown in Table 1. Consider the same bit system using PB format where irreducible polynomial is $x^3 + x + 1$. The results of multiplication mod $f(x)$ are shown as Table 2.

Table 2: Multiplication mod $f(x)$ where $f(x) = x^3 + x + 1$

Mult	001(1)	010(2)	011(3)	100(4)	101(5)	110(6)	111(7)
001(1)	001(1)	010(2)	011(3)	100(4)	101(5)	110(6)	111(7)
010(2)	010(2)	100(4)	110(6)	011(3)	001(1)	111(4)	101(5)
011(3)	011(3)	110(6)	101(5)	111(7)	100(4)	001(1)	010(2)
100(4)	100(4)	011(3)	111(7)	110(6)	010(2)	101(5)	001(1)
101(5)	101(5)	001(1)	100(4)	010(2)	111(7)	011(3)	110(6)
110(6)	110(6)	111(7)	001(1)	101(5)	011(3)	010(2)	100(4)
111(7)	111(7)	101(5)	010(2)	001(1)	110(6)	100(4)	011(3)

Integer	1	2	3	4	5	6	7
Occurences	7	7	7	7	7	7	7

3.1.2. Addition Chain

Multiplication is one of the big issues in Cryptography. In particular, efficient algorithms for group exponentiation have received much attention due to their role in the cryptosystem. Suppose we want to evaluate M^n for given arbitrary numbers M and n. The computation of M^n is an n-multiplication operation, i.e., the result is equals to M multiplied n times. However, computation cost is expensive because a cryptosystem uses a large number n to increase security. For example, RSA typically uses a 1024-bit number for n. Many mathematicians have tried to find a solution that requires a less number of multiplications, and *Addition Chain* is one of the solutions. Interestingly, Addition Chain is not new. Knuth found material that existed in around 200 B.C. related to the addition chain problem [30]. Indian and Arabic works in the 10th and 11th century also mentioned the problem.

An addition chain is a sequence of integers

$$a_0 \quad a_1 \quad a_2 \quad \dots \quad a_r$$

starting from a_0=1 and ending with $a_r = n$ in such a way that any a_k is the sum of two earlier integers a_i and a_j in the chain, i.e.,

$$a_k = a_i + a_j \qquad \text{for } 0 < i, j < k$$

For example, the computation of M^{15} can be done in two ways as shown below.

$$M \times M \times M \times M \times M \times M \times M \times M \times M \times M \times M \times M \times M \times M \times M = M^{15}$$

$$1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10 \quad 11 \quad 12 \quad 13 \quad 14 \quad 15 \qquad (3.12)$$

$$(((M)^2 \times M)^2 \times M)^2 \times M = M^{15}$$

$$1\,2 \quad 3\,4 \quad 5 \quad 6 \quad 7 \qquad (3.13)$$

As we see from the above result, (3.12) needs 15 multiplications but (3.13) needs just 6 multiplications. Using the Addition Chain is almost two times more efficient than performing multiplications sequentially. In the following subsections, we will show the Binary Addition Chain as an example of performing addition, and then discuss exponentiation.

Binary Addition Chain

The Binary Addition Chain is also called the *Addition-and-Doubling* (AD) *Algorithm*. Suppose we want to compute the encrypted point $S = dP \pmod{f(x)}$, where d is an n-bit key value, i.e., $d = (d_{n-1}, d_{n-2}, \dots, d_0)_2$, P is a given plain point, and $f(x)$ is the irreducible polynomial in GF. A modular $f(x)$ operation is a protection from word size overflow. Figure 12 shows the steps of AD, which processes the bits of d from MSB to LSB.

```
if d_{n-1} = 1 then S = P else S =0
for i from n-2 to 0 {
    S = S + S (mod f(x))
    if d_i =1 then S = S + P (mod f(x))
    }
return S
```

Figure 12: Binary AD Algorithm

Suppose we want to compute 15 (*d*) times *P* by using Addition Chain ignoring modular operation since the results from Point Addition and Point Doubling are already optimized by modular operation. 15 can be expressed as a binary $(01111)_2$ in $GF(2^5)$. Since the MSB is 0, the initial value *S* is set to *O*. The following operations are then performed.

(0 1 1 1 1)	$S = O$	// $S = S + S$	1st iteration
↑	$S = P$	// $S = S + P$	
(0 1 1 1 1)	$S = 2P$	// $S = S + S$	2nd iteration
↑	$S = 3P$	// $S = S + P$	
(0 1 1 1 1)	$S = 6P$	// $S = S + S$	3rd iteration
↑	$S = 7P$	// $S = S + P$	
(0 1 1 1 1)	$S = 14P$	// $S = S + S$	4th iteration
↑	$S = 15P$	// $S = S + P$	

As a result, we obtain S = 15*P* in $GF(2^5)$. The scalar multiplication using the Addition Chain takes 8 additions compared to 15 additions without it. Using the AD method, the *n*-bit Addition Chain requires *n*-1 doubling (i.e., $S + S$) operations, and the number of $S + P$ operations is equal to the number of 1's in *d* (except MSB). In our example, we need 4 doubling and 4 addition operations. This Addition Chain is the one of main operations in the ECC scheme and is also the main source to get the private key. The method to get the private key will be discussed in a later section.

Binary Exponentiation

Binary exponentiation uses an algorithm similar to AD called *Squaring-and-Multiplication* (SM), where doubling changes to squaring and addition changes to multiplication. Figure 13 shows the algorithm for binary exponentiation:

```
if d_{n-1} = 1 then S = P else S = 1
for i from n-2 to 0 {
    S= S × S (mod f(x))
    if d_i = 1 then S = S × P (mod f(x))
    }
return S
```

Figure 13: Binary SM Algorithm

For example, suppose we want to compute 10^{15}, P is set to 10 $(01010)_2$, d is set to15 $(01111)_2$, and modular $f(x)$ is set to 32 $(100101)_2$. Since the MSB of d is 0, the initial value S is set to 1.

(0 1 1 1 1)	S= (11100)	// $S = S \times S$ (mod $f(x)$)	1st iteration
↑	S =(01100)	// $S = S \times P$ (mod $f(x)$)	
(0 1 1 1 1)	S= (10100)	// $S = S \times S$ (mod $f(x)$)	2nd iteration
↑	S =(11010)	// $S = S \times P$ (mod $f(x)$)	
(0 1 1 1 1)	S= (00011)	// $S = S \times S$ (mod $f(x)$)	3rd iteration
↑	S =(11110)	// $S = S \times P$ (mod $f(x)$)	
(0 1 1 1 1)	S= (10011)	// $S = S \times S$ (mod $f(x)$)	4th iteration
↑	S =(11011)	// $S = S \times P$ (mod $f(x)$)	

As a result we obtain 10^{15} in $GF(2^5)$. The binary exponentiation using the addition chain takes 8 multiplications compared to 15 multiplications without it. Using the SM method, we can see that the n-bit addition chain needs n-1 squaring and the number of $S \times P$ operations equal to the number of 1's in d (except MSB). The SM method is the main operation of RSA and also the target algorithm to get a private key.

3.1.3. Montgomery's Method

In 1985, P. L. Montgomery proposed a new modular multiplication algorithm without requiring division by n [24]. *Montgomery Multiplication* (MM) exploits additions and divisions by a power of 2, i.e., shift right or left. This method replaces division by n with division by $r = 2^k$, where 2^k is greater than n, GCD(r, n) should be *one* (*1*) and the inverse of r is less than n. Using this method, computer systems can easily implement the modular divide operation since $r = 2^k$ involves simple shifting operations.

MM is used to compute $Z = A \cdot B \cdot r^{-1} \pmod{n}$, where A and B are the n-residues of a and b with respect to r and r^{-1} is the inverse of r in modulus n, i.e., $A = a \cdot r \bmod n$, $B = b \cdot r \bmod n$. Since r is chosen as a power of 2, r and n should be relative prime. In addition, we need n^{-1} in such a way that $r \cdot r^{-1} - n \cdot n^{-1} = 1$. Montgomery found the following equation, where m is set to $t \cdot (-n^{-1} \bmod r) \bmod r$:

$$\begin{aligned}(t + m \cdot n)/r &= \{t + n \cdot [\, t \cdot (-n^{-1} \bmod r) \bmod r]\}\ /r \\ &= (t + n \cdot t \cdot n^{-1})/r \\ &= t \cdot r^{-1} \bmod n\end{aligned}$$

Using this equation, we can divide by r instead of by n. For example, suppose we want to compute $a \cdot b$ using MM. First, a and b changed to A and B, respectively, and the result of $A \cdot B$ is given as

$$\begin{aligned}A \cdot B &= (a \cdot r)\,(b \cdot r)\, r^{-1} \bmod n \\ &= a \cdot b \cdot r \bmod n\end{aligned}$$

Then, we obtain the result by using one more MM with 1, i.e.,

$$\begin{aligned}(a \cdot b \cdot r \bmod n) \cdot 1 &= (a \cdot b \cdot r\,) \cdot r^{-1} \bmod n \\ &= (a \cdot b) \bmod n\end{aligned}$$

Figure 14 shows the MM algorithm and example.

Input : a, b Output : $a \cdot b \cdot r \bmod n$ n^{-1} & r^{-1} are pre computed. Function: MMmult(A,B) t = A*B $m = t * n^{-1} \bmod r$ $u = (t+m*n)/r$ if $u \geq n$ then return $u-n$ else return u	a = 5, b = 3, n = 7, r =2^4 $r^{-1}r - n^{-1}n = 1$, thus $n^{-1} = 9$, $r^{-1} = 4$ A = $a * r \bmod n$ = 5 * 16 mod 7 = 3 B = $b * r \bmod n$ = 3 * 16 mod 7 = 6 MM(A, B) t = A * B = 18 $m = t * n^{-1} \bmod r$ = 18 * 9 mod 16 = 2 $u = (t + m * n) / r$ = (18 + 2*7) /16 = 2 MM(u,1) $t = u * 1$ = 2 $m = t * n^{-1} \bmod r$ = 2 * 9 mod 16 = 2 $u = (t + m * n) / r$ = (2 + 2*7) /16 = 1

Figure 14: MM Algorithm and Example

Consider the case: 5 × 3 mod 7. It is easily computed with a result of 1. However the operation (mod 7) is not easy to computer as mentioned. Consider the r as 2^4, we can get n^{-1} as a 9 and r^{-1} as a 4 by using the Extended Euclid's algorithm, which are discussed in a later section. Then input value 5 (a) and 3 (b) are changed n-residues 3 (A) and 6 (B), respectively. Using A and B, MM(A, B) computed the $a \times b \times r^{-1} \bmod n$ as a result 2. Actually, our purpose is not the computation of $a \times b \times r^{-1} \bmod n$, but $a \times b \bmod n$. To change $a \times b \times r^{-1} \bmod n$ into $a \times b \bmod n$, one more MM is required with the result of MM(A, B) and 1 as an input. As shown in figure 14, the second MM(2, 1) shows the same value with 5 × 3 mod 7.

MM needs pre-computations of n^{-1} and r^{-1}, and the changing processes of input values are also needed. Finally, one more MM is required to get a correct value. Even though MM needs pre and post processing, it is better than conventional modular multiplication in the case of cryptosystems where a large number of modular multiplications are required.

3.1.4.Chinese Reminder Theorem (CRT)

CRT was discovered by a Chinese mathematician, Sun Tse, in the 1st century [10]. The basic theorem is simple. Let m_1 and m_2 be relatively prime integers. Given x_1 less than m_1 and x_2 less than m_2, there exists a unique integer X less than $m_1 \cdot m_2$ such that

$$X = x_1 \bmod m_1$$

$$X = x_2 \bmod m_2$$

$$X = x_1 + m_1 \cdot [(x_2 - x_1) \cdot m_1^{-1} \bmod m_2]$$

This method is useful in reducing an n-bit modular exponentiation into two half size modular exponentiations with CRT processing.

3.1.5. Euclid's Algorithm

Greatest Common Divisor of *a* and *b*, denoted as GCD (*a*, *b*), is the largest positive integer that divides both *a* and *b*. There are two general ways to determine the GCD. The first method is factoring each number *a* and *b* to find the common factor. For example, if *a* = 1728 and *b* = 135, then GCD can be found as

$1728 = 2^6 3^2$, $135 = 3^3 5$ => GCD(1728, 135) = $3^2 = 9$

Since *a* and *b* have the same factor of 3^2, GCD is 9. However, factoring large numbers is not easy. Therefore, Euclid's Algorithm is used to determine the GCD of large numbers. For any integer *a* and *b*, *a* is greater than or equal to *b*, if not, switch *a* and *b*. The first step is to divide *a* by *b*, hence represent *a* in the following equation:

$$a = q_1 b + r_1 \tag{3.14}$$

where q_1 is the quotient and r_1 is the reminder. If $r_1 = 0$, then GCD is *b*. If not, then continue in the following form until the reminder is zero.

$$\begin{array}{lll} a = q_1 b + r_1 & r_2 = q_4 r_3 + r_4 & \vdots \\ b = q_2 r_1 + r_2 & r_3 = q_5 r_4 + r_5 & r_{k-2} = q_k r_{k-1} + r_k \\ r_1 = q_3 r_2 + r_3 & \vdots & r_{k-1} = q_{k+1} r_k \end{array}$$

Then the GCD (*a*, *b*) = r_k. Euclid's algorithm has the following important characteristics. If one of *a* or *b* is not zero, then there exist integers *x* and *y* such that *ax* + *by* = GCD (*a*, *b*).

The process of finding *x* and *y* is called Extended Euclid's algorithm. Suppose we start by dividing by b into a, that is $a = q_1 b + r_1$, then proceed as above in Euclid's algorithm. We will get the successive quotients, i.e., $q_1, q_2, \ldots, q_{k+1}$. The *x* and *y* can be found by the following equations;

$$x_j = -q_{j-1} x_{j-1} + x_{j-2}, \quad x_0 = 0, x_1 = 1 \quad \text{for } (2 < j < k + 1) \tag{3.15}$$

$$y_j = -q_{j-1} y_{j-1} + y_{j-2}, \quad y_0 = 1, y_1 = 0 \quad \text{for } (2 < j < k + 1) \tag{3.16}$$

The x_{k+1} and y_{k+1} are what we want to find. Suppose we want to find out GCD (1180, 482) and *x* and *y* in the equation $1180 \times x + 482 \times y$ = GCD (1180, 482) by extended Euclid's algorithm. The process of finding the GCD (1182, 482) is like below:

$$\begin{array}{ll} 1182 = 2 \times 482 + 216 & 50 = 3 \times 16 + 2 \\ 482 = 2 \times 216 + 50 & 16 = 8 \times 2 \\ 216 = 4 \times 50 + 16 & \end{array}$$

Thus, we can find out the GCD (1180, 482) = r_k = 2, and then we can also get the quotients q_1 = 2, q_2 = 2, q_3 = 4, q_4 = 3 and q_5 = 8. By using these quotients, we can find out the x and y. The following processes show the way of finding x and y.

$x_0 = 0$	$y_0 = 1$
$x_1 = 1$	$y_1 = 0$
$x_2 = -2x_1 + x_0 = -2$	$y_2 = -2y_1 + y_0 = 1$
$x_3 = -2x_2 + x_1 = 5$	$y_3 = -2y_2 + y_1 = -2$
$x_4 = -4x_3 + x_2 = -22$	$y_4 = -4y_3 + y_2 = 9$
$x_5 = -3x_4 + x_3 = 71$	$y_5 = -3y_4 + y_3 = -29$

The 71 and -29 are what we want. Extended Euclid's algorithm is useful to find *the multiplicative inverse*.

To find the multiplicative inverse of 11111 (mod 12345) as an example, Let us analyze the definition of the multiplicative inverse, i.e., the multiplicative inverse x is defined as the multiplicative identity 1 multiplied by 11111 as in equation (3.17).

$$11111x = 1 \pmod{12345} = 12345y' + 1 \tag{3.17}$$

$$11111x + 12345y = 1 = \text{GCD}(11111, 12345), \text{ Substitute } y = -y' \tag{3.18}$$

from equation (3.18), we can get the quotients q_1 = 1, q_2 = 9, q_3 =246, q_4 = 1, and q_5 =4 by Euclid's algorithm. Then, we can find the number x=2471 as the multiplicative inverse by Extended Euclid's algorithm. To verify it, we substitute x with 247. Then we can check it with following equation (3.19);

$$11111 \times 2471 = 27455281 = 2224 \times 12345 + 1 = 1 \pmod{12345} \tag{3.19}$$

3.1.6. Projective Coordinate

The projective coordinate is used to eliminate the need for performing division. Affine coordinate in ECC as shown in the Cryptography section requires a division operation, i.e., computing λ requires one division in both $GF(p)$ and $GF(2^m)$. In computer systems, a division operation needs more clock cycles than a multiplication operation [28]. Affine coordinate (x, y) can be represented as a projective (Jacobian style) coordinate (X, Y, Z), where $(x, y) = (X/Z^2, Y/Z^3)$. A point doubling and a point addition are transformed as shown in Table 3.

Table 3: Relationship between Affine Coordinates and Projective Coordinates [30,43]

$y^2 + xy = x^3+ax^2+b$ over $GF(2^m)$, $b \neq 0$	Affine Coordinates(x, y)	Projective Coordinates(X, Y, Z)
Point Addition	$(x_1,y_1) + (x_2,y_2) = (x_3, y_3)$ $\lambda = (y_2-y_1)(x_2-x_1)^{-1}$ $x_3= \lambda^2+\lambda+x_1+x_2+c$ $y_3 = \lambda(x_1+x_3)+x_3+y_1$	$(X_1,Y_1,Z_1) + (X_2,Y_2, Z_2) = (X_3, Y_3, Z_3)$ $A = X_1Z_2^2$ $B = X_2Z_1^2$ $C = A + B$ $D =Y_1Z_2^3$ $E = Y_2Z_1^3$ $F = D + E$ $G = Z_1C$ $H = FX_2 + GY_2$ $Z_3 =GZ_2$ $I = F + Z_3$ $X_3 = aZ_3^2 + IF + C^3$ $Y_3 = IX_3 + HG^2$
Point Doubling	$(x_1,y_1) + (x_1,y_1) = (x_3, y_3)$ $\lambda = x_1 +(y_1)(x_1)^{-1}$ $x_3 = \lambda^2 +\lambda + c$ $y_3 = \lambda(x_1+x_3)+x_3+y_1$	$(X_1,Y_1,Z_1) + (X_1,Y_1, Z_1) = (X_3, Y_3, Z_3)$ $Z_3 = X_1Z_1^2$ $A = bZ_1^2$ $B = X_1 + A$ $X_3 = B^4$ $C = Z_1Y_1$ $D = Z_3 + X_1^2 + C$ $E = DX_3$ $Y_3 = X_1^4Z_3 + E$

A division operation in Affine Coordinates does not appear in Projective Coordinates. Instead of division operations, Projective Coordinates need more multiplication than Affine Coordinates, i.e., Point Doubling needs 12 multiplications in Projective Coordinates while it needs only 2 multiplications in Affine Coordinates. Thus, the appropriateness of using projective coordinates is determined by user. Furthermore, a division operation is needed to change Projective Coordinates into Affine Coordinates.

3.2. Power Model

Most of our electric appliances like TVs, game cubes and computers, consume power from power outlets and lots of hand-held devices also use a battery as a power source. This means that power consumption rates are one of the performance standards, which makes limiting power consumption a critical issue for embedded systems. A cryptosystem requires numerous computations,

which means lots of 0-to-1 or 1-to-0 transitions at the gate-level. These gates usually consist of one or more complementary metal-oxide semiconductor (CMOS) logic circuits.

The dominant equation for the CMOS power model is given by

$$P = ACV^2f + tAVI_{short}f + VI_{leak} \quad (3.20)$$

The first term measures the dynamic power consumption, which is proportional to the frequency *f*, the active gates *A*, the capacitance *C*, and the square of voltage V^2. The second terms are related to short-circuit power, which varies with short circuit current I_{short}, and instantaneous time *t*. The last term, VI_{leak}, shows the leakage power by Ohm's law. However, the second and third terms are less significant than the first term in most CMOS circuits.

P is defined as the consumption at a discrete point in time while *energy E* is defined as the power dissipation during the execution time *D*, as shown in the equation:

$$E = P_{avg} \cdot D \quad (3.21)$$

As we know, power uses watts units (W), and energy uses Jules (J = W/s). Even if a processor operates at low power, its energy can be greater than or equal to the energy used when operating at high power. Another example, a single core can finish a job in time 2*t* using power *p* while a dual core takes time t using power 2*p*. The energy between the single and dual core will be the same even though the performance of dual core is better than that of the single core. Energy or instantaneous power, as a measure of efficiency, is not suitable for evaluating the system. Therefore, evaluating the power-performance of a system based only on instantaneous power or energy is not sufficient. Thus, researchers use a new method for power-performance, the *energy-delay product* [18]:

$$\text{EDP} = E \cdot D \ (\text{J}\cdot\text{sec}) \quad (3.22)$$

By using *Energy-Delay product* (EDP), the above example of comparison with single core and dual core can be meaningful. The EDP of the single core is $(2t)^2 \cdot p$ while that of the dual core is $t^2 \cdot 2p$. Thus, dual-core is twice as good as a single core. Notice that smaller EDP means better power-performance efficiency.

There are several ways to improve performance-power efficiency in mobile devices. Research on power reduction consists of techniques at the circuit level, architectural level and software level. We will briefly discuss these methods in the following subsections. For more detail, refer to [17].

3.2.1. Power Reduction at the Circuit Level

We will overview three methods for power reduction at the circuit level. The first method is *clock gating*, which exploits a characteristic that all gates do not need to be clocked at the same time. Therefore, the power consumption can be reduced by turning off the clock tree which branches to unused gates. The second technique is *half-frequency and half-swing clocks*. A gate uses either the rising edge or falling edge. The half-frequency technique uses both edges of the clock to synchronize events. Half-frequency and half-swing clocks reduce power consumption. Traditionally, hardware events such as register file writes occur on a rising clock edge. Half frequency clocks synchronize events using both edges, and they tick at half the speed of regular clocks. Thus, cutting clock switching power reduces by half. Reduced-swing clocks also often use a lower voltage signal and thus reduce power by power equation (3.20) [31]. That is, the half-swing clock technique swings using only half of the supply voltage for making slow frequency. The last method is to omit the clock tree using asynchronous logic. The clock tree consumes 30% of the processor power. Therefore, removing the clock tree using asynchronous logic can significantly reduce power.

3.2.2. Power Reduction at the Architectural Level

We have reviewed power saving features at the circuit level as though they were a fixed foundation upon which programs execute. However, programs exhibit wide variations in behavior. Researchers have been developing hardware structures whose parameters can be adjusted on demand so that one can save power consumption by activating just the minimum hardware resources needed for the code that is executing.

There are various techniques to reduce power consumption in architectural level. Memory systems including cache and main memory are one of the major power consumers, which consists of two types of power loss: dynamic power and leakage power. Whenever a processor accesses memory, it uses dynamic power. Therefore, accessing only the required part of memory is a way to reduce power loss in the memory hierarchy. One memory system power reduction method is adaptive caches, which can be selectively activated based on the application workload. One example of such a cache is the Deep-Submicron Instruction (DRI) cache [32]. This cache permits one to deactivate its individual sets on demand by gating their supply voltages. To decide what sets to activate at any given time, the cache uses a hardware profiler that monitors the application's cache-miss patterns. Whenever the cache misses exceed a threshold, the DRI cache activates previously deactivated sets. Likewise, whenever the miss rate falls below a threshold,

the DRI deactivates some of these sets by inhibiting their supply voltages.

The second example for power reduction is *Dynamic Voltages Scaling* (DVS). DVS algorithms assign different speeds for different tasks. These speeds remain fixed for the duration of each task's execution. When we uses parallel processing for some workloads, some threads, which are parallelized workloads, will be waiting after finishing their own job. We can reduce the power by reducing voltage for those threads that have completed around the same time as the thread that takes the longest time. Accurate branch predictors could be another power reduction method. For example, if a branch predictor makes a wrong prediction, useless instructions waste power. Therefore, we can reduce power, especially dynamic power, by using a better branch predictor.

3.2.3. Power Reduction of Software Level

As we saw in the power consumption equation in (3.20), reducing the supply voltage has a big advantage in power reduction. At the software level power reduction is usually performed by the operating system (OS). OSs control the computer resources like memory systems, file systems, and processes. Some processes have a marginal time to complete computations. If an OS knows the computation deadline, it can reduce power by setting the supply voltage to satisfy the computation deadline.

There are two types of approaches to scaling. One is that the OS provides direct interface for voltage scaling, which processes can use to schedule their own voltage needs. The second approach is the OS has a control of determining voltage needs of each process. For example, a MPEG decoder just has to complete one frame within $1/30^{th}$ second, making it unnecessary to complete its job as soon as possible. Thus, the OS runs the MPEG decoder with a typical voltage at the first time, and then it computes the marginal time to adjust voltage scaling [17].

3.3. Parallel Computing Architectures

Parallel processing simply means concurrent performance at a particular cycle. It exploits thread-level parallelism instead of instruction-level parallelism (ILP). The ILP extracts parallelism in a sequential single program: modern superscalar microprocessors have multiple execution units working in parallel using this with OoO (Out-of-Order) execution. The TLP, however, is explicitly represented by multiple threads of execution that are inherently parallel.

Multiple threads can be executed in parallel on multi cores. This multi-threading generally occurs by time slicing (similar to time-division multiplexing), wherein a single processor switches between different threads, in which case the processing is not literally simultaneous as the single processor is really doing only one thing at a time. This switching can happen so fast that it gives the illusion of simultaneity to an end user. For instance, most PCs several years ago only contained one processor core, but we could run multiple programs at once, such as typing in a document editor while listening to music in an audio playback program. Though the user experiences these activities as simultaneously, in reality, the processor quickly switches back and forth between these separate processes.

On a multiprocessor or multi core system, which are now coming into general use, multithreading can be achieved via parallel processing, wherein different threads and processes can run simultaneously on different processors or cores. In other words, threads are distributed to distinct processors and executed concurrently. It introduces much more efficient performance than multithreading on one core-processor [28].

The idea of using multiple processors both to increase performance and to improve availability dates backs to the earliest computers. Flynn proposed a simple model of categorizing all computers that is still used today. He categorized the parallelism in the instruction and data systems into four types, which will be discussed in this subsection [28].

3.3.1. Single instruction stream, single data stream (SISD)

SISD are implemented in a serial computer. Only one single instruction stream is being acted on by the CPU and only one data stream is being used as input during any one clock cycle. Although SISD is the oldest, it is the most prevalent form of computer until recently [22]. Figure 15 shows an example of SISD execution.

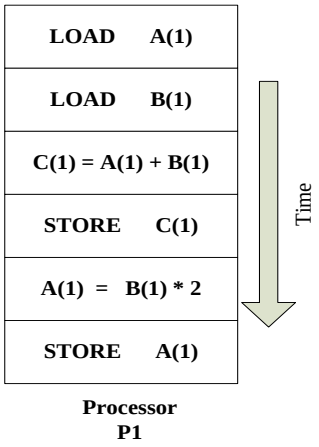


Figure 15: Flow of SISD

As we see in Figure 15, it can execute one instruction at a particular time. After loading A, this system can load B, and then can compute C, and so on.

3.3.2. Single instruction stream, multiple data stream (SIMD)

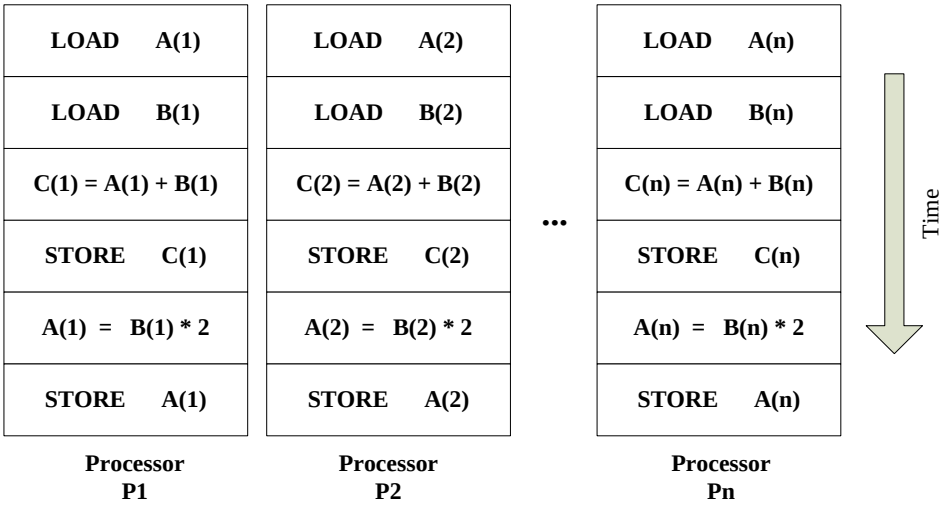


Figure 16: Flow of SIMD

In SIMD, the same instruction is executed by multiple processors using different data streams. Each processor has its own data memory. Each processing unit can operate on a different data element. It is best suited for specialized problems characterized by a high degree of regularity, such as image processing. SIMD type architectures perform synchronous and deterministic execution [22]. Figure 16 shows an example of SIMD execution. As we see Figure 16, n processors execute the same instruction with different data. Processor P1, P2, and Pn perform data 1, 2, and n respectively.

3.3.3. Multiple instruction streams, single data stream (MISD)

A single data stream is fed into multiple processing units. Each processing unit operates on the data independently via independent instruction streams. Few actual examples of this class of parallel computer have ever existed. One is the experimental Carnegie-Mellon C.mmp computer (1971). Figure 17 shows an example of MISD execution [22].

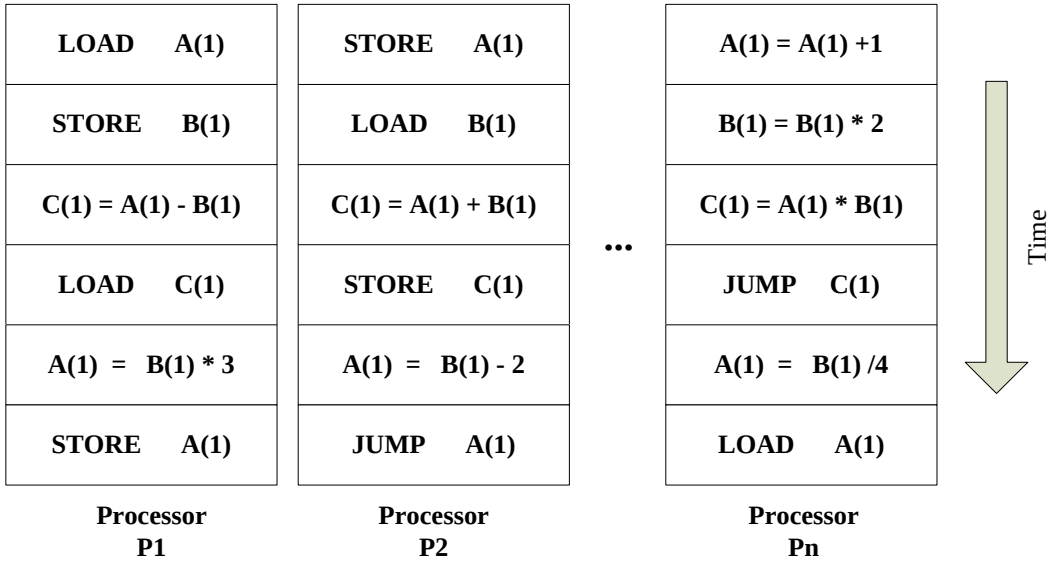


Figure 17: Flow of MISD

MISD has a contrary concept to SIMD. As we see in Figure 17, a processor P1 performs a load operation with data 1, P2 performs a store operation with same data, and Pn performs an add operation. MISD is hard to implement due to data dependencies.

3.3.4. Multiple instruction streams, multiple data stream (MIMD)

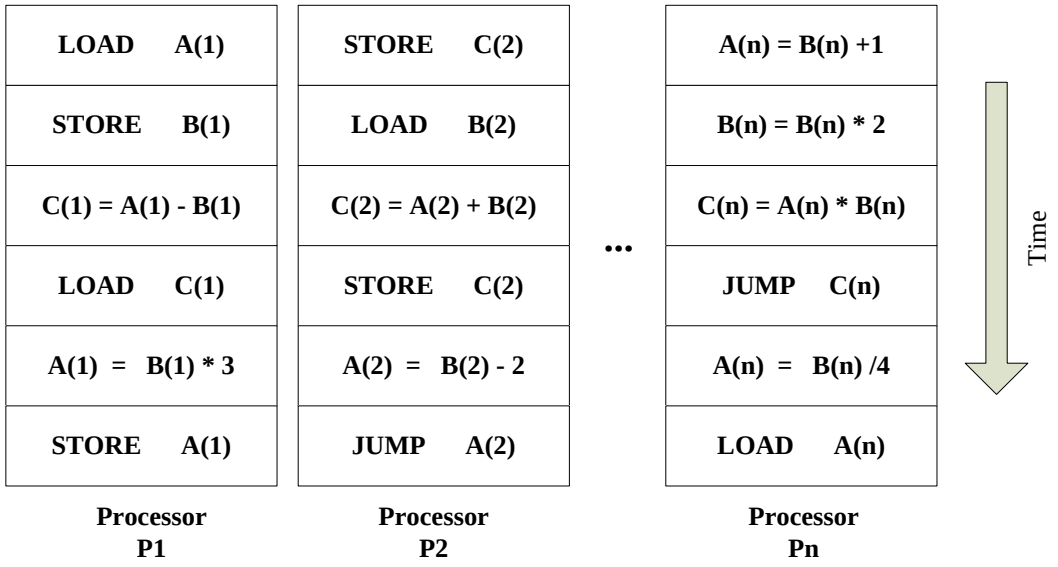


Figure 18: Flow of MIMD

Currently, the most common type of multi core computer is MIMD. Most modern computers fall into this category. Every processor may be executing a different instruction stream while working with a different data stream. Execution can be synchronous or asynchronous, and deterministic or non-deterministic. Figure 18 shows an example of MIMD execution [22].

As we see in Figure 18, each processor performs different operations with different data. While P1 performs the loading of data A(1), Pn performs an add operation with A(n) and B(n).

4. Side Channel Attack (SCA)

As we mentioned before, Cryptography is the science of hiding information. However, there are also people who try to uncover hidden information without having access to a secret or private key. This process of cracking hidden information is usually called *Cryptanalysis* or *Attack*. Conventional cryptanalysis has been focused on finding weaknesses in crypto-algorithms. Recently, people have been trying to break cryptosystems using side channel information, which is referred to as *Side-channel Attack* (SCA). One general method of SCA is power and timing analysis because different operations of a cryptosystem result in different power consumption and execution time characteristics.

SCA has become the focus for cryptanalysis ever since Kocher discovered side channels, such as voltages, currents, power consumption and electromagnetic emissions of a cryptosystem [14, 15], can be used to perform attacks. Obviously, SCA is not new. One well-known example is a safecracker who uses feelings from his hands and sounds he hears to open a cash box. This is also similar to a submarine that can find out adversary's ship size, class, and speed using sonar.

The following subsections discuss general SCAs and countermeasures for SCA.

4.1. SCA Methods

There are various techniques utilized in cryptanalysis. We will briefly introduce three types of general methods; *timing analysis*, *power analysis*, and *CPU component* (Cache) *analysis*.

4.1.1. Timing Analysis

A processor results in different execution times depending on the types and the number of instructions executed and the amount of data processed. Before the discovery of side channels, computer engineers thought that execution time for encryption/decryption was the benchmark for comparing different crypto-algorithms. That is, the shortest execution time means the most efficient and thus the most marketable.

However, the execution time depends not only on the crypto-algorithm but also on the input data, i.e., plaintext/cipher text and key. Kocher showed a relationship between timing characteristic and input data in 1995 [14]. For example, as shown in Figure 12, the addition and doubling (AD) algorithm is the main function of ECC scalar multiplication. When d_i is 1, more time is required to compute an elliptic point. As discussed in Section 2, n-bit Addition Chain needs n-1 doublings operations and the number of additions equal to the number of 1's in the bi-

nary expansion of d, without the MSB. For example, if d is $(11111)_2$, the cryptosystem computes 4 doubling and 4 addition operations. If d is $(10000)_2$, only 4 doubling operations are needed. Therefore, an attacker can figure out how many bits are set to 1 since the execution time depends on numbers of operations.

If an attacker can observe and compare the execution times of several loop iterations in the addition chain, the attacker may be able to deduce the value of the corresponding bits in d. Thus, the private key in ECC can be obtained. Cryptanalysis for RSA is similar to ECC; however, in both cases an attacker may not be able to observe the timing characteristics of individual loop iterations.

For example, suppose that an attacker, Eve, sends a series of plain points (i.e., plain texts) $P_1, P_2, \ldots, P_k$ to a cryptosystem that implements ECC using the AD algorithm. Eve records the times $t_1, t_2, \ldots, t_k$ the cryptosystem take to return the encrypted points $S_1, S_2, \ldots, S_k$ on each of the known points $P_1, P_2, \ldots, P_k$. Then, the timing analysis of $t_1, t_2, \ldots, t_k$ allows Eve to recover the bits in d.

4.1.2. Power analysis

Power consumption occurs when the output of a gate transitions from 0 to 1 and vice versa. Power analysis consists of two phases - data collection and data analysis. In the *data collection* phase, an adversary monitors power consumption of a cryptosystem to obtain a pattern depending on the processed data and executed instructions. In the *data analysis* phase, the collected data is analyzed and compared with randomly chosen data to guess on the value of d.

We can find a correlation between time analysis and power analysis. However, we can find more useful information from power analysis compared to time analysis. For example, if the chosen data results in the same power consumption as with the collected data, then the chosen value would most likely be the secret key inside the cryptosystem.

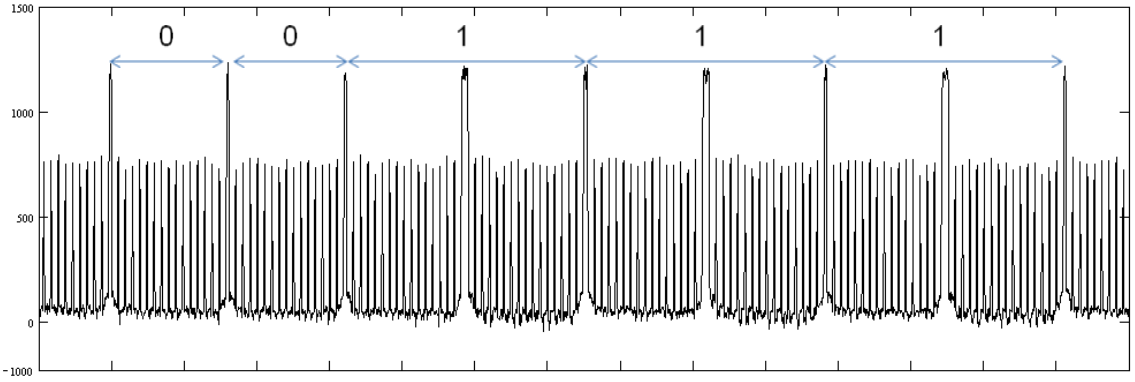


Figure 19: Power Analysis in RSA

Figure 19 shows an actual power analysis output of an RSA addition chain [42]. The power trace shows nine spikes during an addition chain operation. In each iteration, the value *S* to be squared or multiplied is initially loaded from memory to a register, which causes the power to spike. If the multiplication requires the value *P* to be loaded, then another, slightly wider spike occurs. As a result, the square operation $d_i = 0$ (i.e., a narrow spike) can be distinguished from the square-and-multiply operation $d_i = 1$ (i.e., a narrow spike followed by a wider spike). Thus, five key bits (i.e., $d = 00111$) can be determined from the above figure.

4.1.3. Micro-architecture Analysis

Microarchitecture analysis is a kind of time or power analysis on a specific component of the microarcitecture. For example, cache architecture can be analyzed to determine what data was processed. A processor needs to retrieve data at high speeds in order to improve performance. However, the latency of main memory makes it difficult to deliver data at high speeds. Moreover, the gap between the latency of main memory and processor speed continues to grow causing performance degradation. Therefore, cache is used to reduce this gap.

A cache attack exploits the cache hits and misses that occur during the encryption/ decryption process of a cryptosystem. Usually, cryptosystems have data-dependent memory access patterns and leak information about the cache hit/miss statistics of ciphers through side channels, e.g., execution time and power consumption.

For example, S-box in DES will be used 16 rounds during the encryption/decryption process. In each round, there is an access to the eight substitute tables as shown in Figure 6. The indices, which are 48-bit input to S-box, used in the first two rounds are given below:

$I_0 = K_0$ XOR E(R_0)

$I_1 = K_1$ XOR E(R_1) = K_1 XOR E(L_0 XOR P(S(K_0 XOR E (R_0))),

where E is E-table, P is the permutation function, and K_0 and K_1 are subkeys. If an attacker can capture the profile of the cache activity during the second round, i.e., the outcomes of the S-box using index I_1, then I_0 and I_1 can be correlated. From this correlation, the attacker can partially find the bits of K_0 and K_1. Note that an attacker knows the internal structure of DES and the input value (i.e., plaintext L_0 and R_0).

The aforementioned technique is a hypothetical method where the author's assumed that it was possible to capture the cache profile without explaining how this can be achieved [45]. A real example of this type of attack was discussed in [48], where a spy process is run simultaneously within the cryptosystem. The spy process continuously reads each cache set in the same order and measures the access times during the operation of the cipher. If a read access time of a cache set takes longer than usual, the attacker can conclude that this set was accessed during the time interval between the last and the current read access period. Using this method, an attacker was able to identify the value of *d* in an Addition chain operation as shown Figures 12 and 13. For example, suppose a crypto algorithm uses the point *P* and *S* data structure in memory as shown in Figure 20. The attacker uses spy data which has the same index in memory and continuously accesses it. Then the cache updates *P* and *S* when d is one, or *S* is updated when *d* is zero. By this characteristic of the algorithm, the attacker can figure out the secret value *d*.

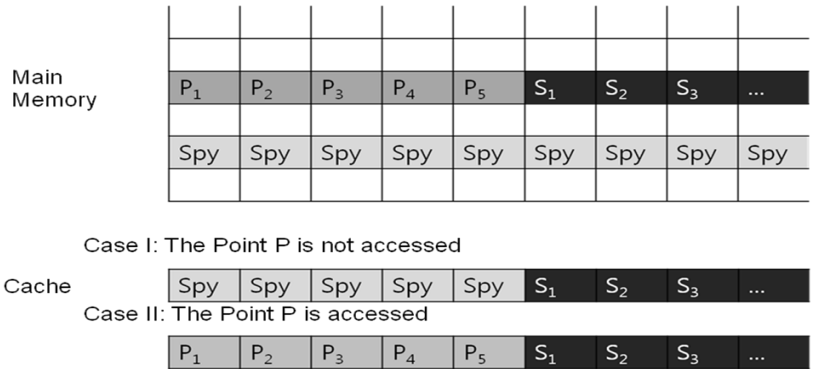


Figure 20: Cache Analysis

4.2. SCA Countermeasures

There are also several techniques to prevent SCA. These methods include gate-level masking, inserting dummy instructions, and a regularly behaving algorithm. Countermeasures (CM) are not limited to certain types of SCAs; e.g., Countermeasure for timing analysis could be that for power analysis or CPU component analysis.

4.2.1. CM for Timing Analysis

There are two ways to make timing analysis difficult for attackers. The first method is to guarantee constant execution time regardless of input data using dummy instructions; however this method makes the system inefficient.

```
if d_{n-1} = 1 then S = P else S = 0
for i from n-2 to 0{
    S = S + S (mod p)
    if d_i =1 then S = S + P (mod p)
    else dummy instructions
    }
return S
```

Figure 21: Dummy instruction in AD Algorithm

Another option is to introduce some noise using randomly inserted instructions, which require more execution time and make them difficult to conduct timing analysis.

Figure 21 shows an example of using a dummy instruction that has similar weight, i.e. execution time, as $S + P \pmod{p}$. Therefore, the execution time will be similar regardless of input d. By using this method, an attacker will have trouble analyzing the execution time of a cryptosystem. However, this method results in constant execution time.

In the second method, the instructions can be chosen and inserted randomly, which will result in a different execution time. The difference with dummy instructions is independent of the crypto-algorithm. Dummy instructions are used to make crypto-algorithms behave consistently. The purpose of Random instructions is different from Dummy instructions. Random instructions are used to make noise.

```
if d_{n-1} = 1 then S = P else S = 0
for i from n-2 to 0{
    S = S + S (mod p)
    Random instructions (1)
    if d_i =1 then S = S + P (mod p)
    Random instructions (2)
    }
return S
```

Figure 22: Random instruction in AD Algorithm

Random instructions randomly generated in cryptosystem are shown Figure 22. Consider Random instructions (1) and (2) are inserted randomly. Random instructions (1) is related to arithmetic operations and Random instructions (2) is related to branch operations. Random instruction (1) could create additional spike similar to what was shown in Figure 19 with extra power consumption and execution time. Random instructions (2) create confusion to figure out the value d_i as well as generating power and time confusion. Randomly chosen instructions [44] might be more efficient than the first method by appropriate use. A recent paper shows 29.8% in runtime and 27.1% in energy consumption overhead when using random instructions [54].

4.2.2. CM for Power analysis

There are several techniques to prevent power analysis. These methods include gate-level masking, inserting dummy/random instructions, and regularly behaving algorithms that force consistent behavior regardless of the data processed [16, 19]. Masking at the gate-level is algorithm independent and in principle it can even be done completely automatically, i.e., a program can be used to control when gates are masked.

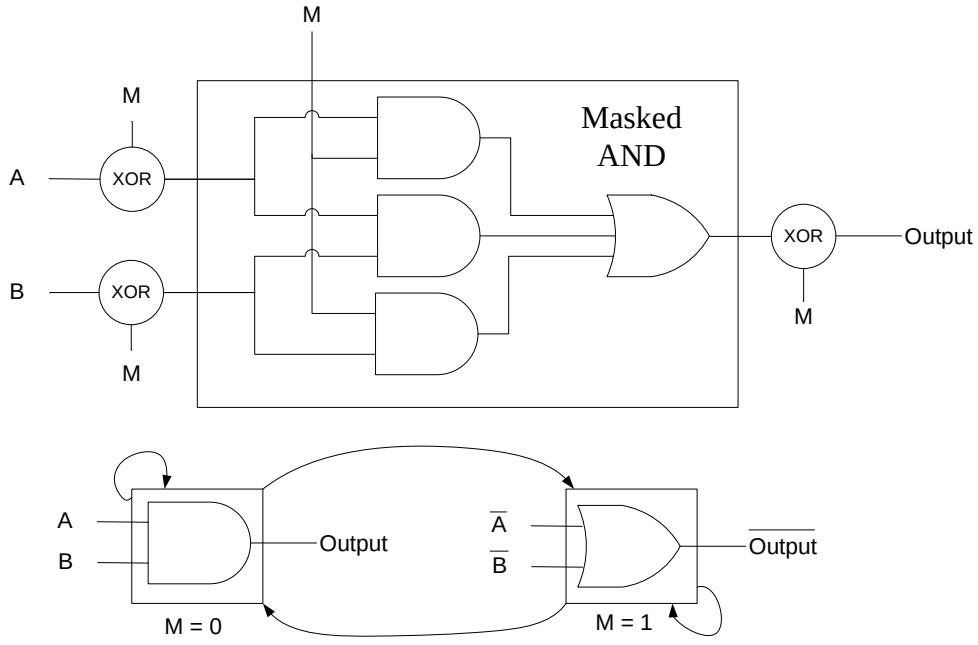


Figure 23: Masked Gates

Gate-level masking makes power monitoring difficult. In a digital circuit, a gate generates the output q based on the inputs a and b, i.e., $q = f(a, b)$. In a masked circuit, the inputs as well as the output are masked. This means that $a_m = f(a, m)$, $b_m = (b, m)$ and $q_m = f(q, m)$, where

m is a randomly generated mask to make SCA difficult [37]. Figure 23 shows an example of masked gates. The normal circuit consists of two AND gates and one OR gate. Using masked gates, each input signal has been randomized and processed by predefined logic, e.g., XOR logic. Then the output is masked. The output signal is recovered with one more XOR gate. The masked Gate acts as an AND gate or OR gate depending on M. The input and output are also masked with M. The masked gates in Figure 23 have both inputs and outputs masked and behave as AND gates regardless of the M signal. Since M is randomly generated, cryptosystem analysts have difficulty in getting information.

The second method involves inserting dummy instructions. This technique gives an effect of eliminating *if-then* statements. Dummy instructions have the same workloads as normal instructions, e.g., $S = S + P \pmod{p}$ in Figure 21. This results in similar power consumption whether a branch is taken or not taken.

The last method is to use a regularly behaving algorithm, which involves replacing the algorithm of a major arithmetic operation with another algorithm that behaves consistently regardless of data. Usually, an algorithm that does not have branch instructions due to *if-then* statements is considered a regularly behaving algorithm. In fact, the dummy instruction and regularly behaving algorithm share the same idea: it consists in ultimately having an algorithm that behaves consistently and regularly.

In RSA/ECC, the MM algorithm is used for this purpose [19]. As discussed in Section 3, it operates regularly except for the last subtraction. Furthermore, MM in $GF(2^n)$ does not have the last subtraction part [33] because no carries are generated when it performs additions and subtractions. As shown in Figure 24, the intermediate value m is always less than r, i.e., $0 \leq m < r$. Thus, the inequality $0 \leq mn < rn$ (1) is satisfied by multiplying the inequality by n. Another value $t\,r^{-1}$ is also less than n. Therefore the inequality $0 \leq t < rn$ (2) is satisfied. From these inequalities (1) and (2), i.e., $0 \leq t + nm < 2rn$, the value $u = (t + nm)/r$ might be bigger than the value n. Therefore, the final subtraction is required in $GF(p)$.

The equation $(t + nm)$ does not causes a carry during the addition in $GF(2^n)$. The value u is always less than the value rn, i.e., $0 \leq t + nm < rn$.

Input : a, b

Output : $a \cdot b \cdot r \bmod n$ (i.e., $A \cdot B \cdot r^{-1} \bmod n$)

n^{-1} & r^{-1} are pre computed.

Function: MMmult(A,B)

$t = A \cdot B$

$m = t \cdot n^{-1} \bmod r$ m is always less then r

➔ $0 \le m \cdot n < r \cdot n$

$u = (t + m \cdot n)/r$ $0 \le t \cdot r^{-1} < n$ since $t = A \cdot B$

➔ $0 \le t < r \cdot n$

Both t and mn are less than n, the addition in $GF(2^n)$ is also less than n.

Figure 24: MM in GF(2^n)

4.2.3. CM for Micro-Architecture Analysis

There are several techniques to make cache analysis difficult. General methods are *using a data-oblivious memory access pattern, disabling cache sharing,* and *Static or disabled cache.*

Oblivious memory access pattern is that the pattern of accesses to the memory is completely oblivious to the data passing through the algorithm. Using a naïve approach, to implement a memory access one can

read *all* entries of the relevant table, in fixed order, and use just the one needed. This induces significant slowdown. Goldreich and Ostrovsky [49] showed a generic program transformation for hiding memory accesses, which is quite satisfactory from a theoretical perspective. However, its concrete overheads in time and memory size appear too high for most applications.

The second method is disabling cache sharing. To protect against software-based attacks, it would suffice to prevent cache state effects from spanning process boundaries. However, practically this is very expensive to achieve. On a single-threaded processor, it would require flushing all caches during every context switch. On a processor with simultaneous multithreading, it would also require the logical processors to use separate logical caches, statically allocated within the physical cache [50].

The last and brutal countermeasure against the cache-based attacks is to completely disable the CPU's caching mechanism; the effect on performance would be devastating. An alterna-

tive is to activate a “no-fill” mode where the cache is used but not updated (i.e., eviction is disabled). We are not aware of any processor that provides the necessary facilities [50].

5. Related Work

Although there has been a lot of work on effective crypto-algorithms, SCA is a relatively new subject. Various countermeasures for SCA have been introduced since it first emerged. Using a SCA-resistant crypto-algorithm might be an easy choice for protecting cryptosystems. However, these research efforts have mostly been performed on a single-core processor. The multiplication operation is the most important part of a cryptosystem. With the advent of multi-cores, people have made some multiplication algorithms for multi-core processors. In RSA/ECC, one useful algorithm is the Montgomery modular multiplication or simply Montgomery Multiplication (MM). A lot of work has appeared in the literature related to efficient implementations of MM in both software and hardware.

In this section, J. Fan's Montgomery Multiplication modular method [24], which has been implemented on multi-cores will be discussed. To the best of our knowledge, Fan's method is the only implementation on multi-cores as of this writing. Following Fan's method, two different types of projective coordinates will be discussed. Projective coordinates are used to reduce the burden of a division operation in ECC. The projective coordinates give the improvement of performance as described in background work. However, the projective coordinates need a higher multiplication to division ratio. To reduce the multiplications, some mixed coordinates techniques are used [25, 46].

5.1. Montgomery Multiplication on Multi-cores

MM is a fundamental operation in many PKC algorithms, such as RSA and ECC. Since the division operation in modular reduction is time-consuming, Montgomery [24] proposed a new algorithm that avoids division as discussed in the previous section. However, it is still computationally intensive, which makes it difficult for software implementation.

With the advent of multi-cores in commercial computers including embedded systems, MM can be accelerated using parallel processing. Fan *et al.* [26] proposed partitioning the MM algorithm into tasks and then mapping those tasks to specific cores in order to achieve a high performance. Figure 25 [27] shows a more detailed Montgomery algorithm compared to Figure 14. This is a version for long word Montgomery multiplication. As discussed in Section 2, RSA/ECC uses longer bit vectors (e.g., 131 bits over the $GF(2^{131})$) than the word size of general-purpose computer systems, i.e., 32 bits or 64 bits. In a typical implementation, operations on large numbers are performed by breaking the number into multiple words. Thus, MM has the

long bit vectors divided into several words. Suppose we want to compute $X \cdot Y \cdot R^{-1} \bmod M$. First, X, Y, and M are broken into words, and then MM is performed word by word. The difference between Figure 19 and Figure 5 is that the Montgomery algorithm in Figure 25 generates a carry Z. Otherwise, the rest of the operation is the same as the original algorithm.

The main data dependencies in MM using multi-cores are due to the carries generated by additions. As can be seen in Figure 25, the operation is $(z_j + (X \cdot y_i)_j + (M \cdot T)_j + carry\ z_{j-1})$, where carry z_{j-1} is the carry generated when computing z_{j-1}. Obviously, $x_j \cdot y_i$, for any $0 \le i, j \le s\text{-}1$, is dependent only on the operands X and Y. We can also calculate $M \cdot T$ immediately after the generation of T.

Function: Radix-2^w(n-bit) Montgomery modular multiplication [27]

Input: integers $M = (m_{s-1}, \ldots, m_0)_r$, $X=(x_{s-1}, \ldots, x_0)_r$, $Y=(y_{s-1}, \ldots, y_0)_r$,
where $0 \le X, Y < M$, $r=2^w$, $s=\lceil n/w \rceil$, $R=r^s$ with $\gcd(M,r)=1$
and $M' = -M^{-1} \bmod r$

Output: $Z = X \cdot Y \cdot R^{-1} \bmod M$

$Z = (z_{s-1}, \ldots, z_0)_r \leftarrow 0$
for i= to s-1 do
$T \leftarrow (z_0 + x_0 \cdot y_i) \cdot M' \bmod r$
$Z_i \leftarrow (Z_i + (X \cdot y_i)_i + (M \cdot T)_i + carry\ from\ z_{i-1})/r$
end for
if $Z > M$ then
$Z \leftarrow Z - M$
end if
return Z

Figure 25: Radix-2^w (n-bit) MM

It is therefore desirable to partition the Montgomery algorithm so that carries are localized within a core. T is generated and distributed to the other cores as shown in Figure 26. The other cores compute the distributed T. Using this method, Fan *et al.* increased the performance of multiplication. Compared to the implementations on a single-core system, the performance can be improved by a factor of 1.87 and 3.68 when 256-bit modular multiplication is being performed on a 2-core and 4-core system, respectively [26].

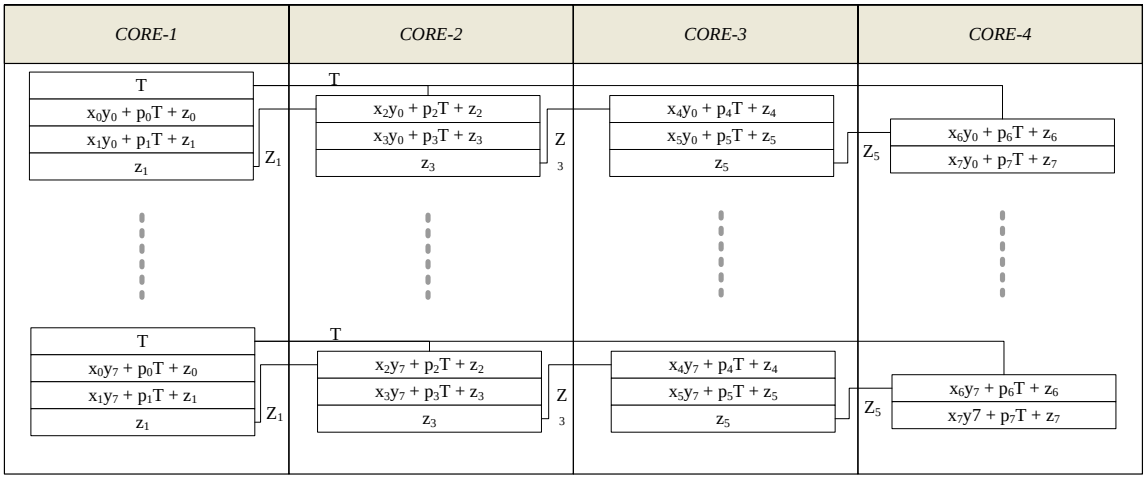


Figure 26: Fan's MM on Multi-cores.

5.2. Montgomery multiplication in $GF(2^k)$

Dhem *et al.* proposed the first timing attack on RSA using MM. They focused on the final subtraction performed in MM. They experimentally showed a timing attack by analyzing the appearance rate of final subtraction correlated with the multiplication of two random inputs. The experiment showed 17% of multiplication generates the final subtraction. They expected a 512-bit RSA key to be cracked within a few minutes once 350,000 timing measurements are collected [36].

Koç and Acar proposed MM in $GF(2^k)$ [33]. They removed the final subtraction from MM by considering the unique characteristics of $GF(2^k)$, which does not generate a carry. The algorithm is shown as Figure 24.

5.3. Projective Coordinate in $GF(2^k)$

As described in the background work, division is the most expensive operation over *GF*s. Jebril et al. showed the effectiveness of projective coordinate systems compared to affine coordinate systems. According to their experiment, one division has similar workloads with 10 multiplications in terms of execution times [52]. Therefore, researchers have proposed the various projective coordinates in order to replace the inversion operations with multiplications [25, 30, 46]. In this section, we will show two types of projective coordinates; projective coordinates [47] and Jacobian coordinates [30]. The differences with two coordinate systems come from mapping with affine coordinates, i.e., projective coordinates have the form $(x, y) = (X/Z, Y/Z)$, while Jacobian coordinates have the form $(x, y) = (X/Z^2, Y/Z^3)$. Figure 27 and 28 show an elliptic point addition and an elliptic point doubling.

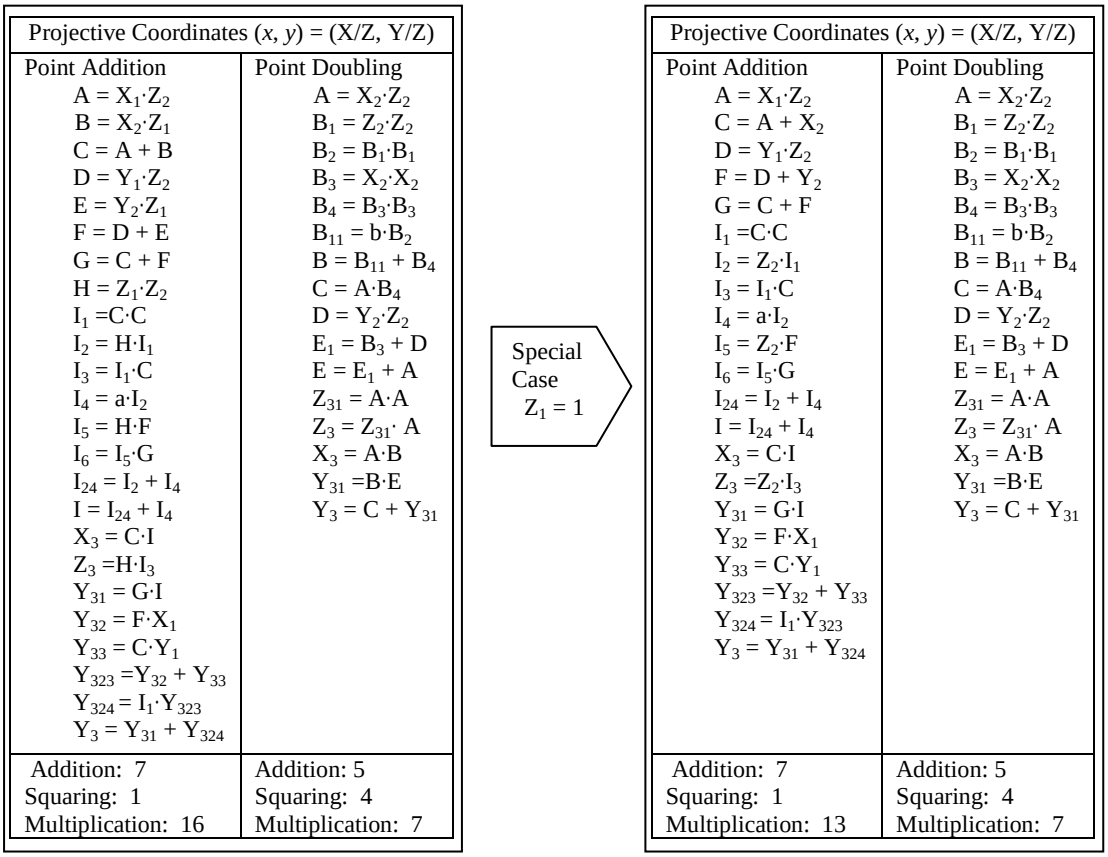


Figure 27: Projective Coordinates and Mixed Coordinates with Affine systems

As shown in Figure 27, 7 additions, 1 squaring and 16 multiplications are required during an elliptic point additions, while 5 additions, 4 squaring and 7 multiplications are needed in an elliptic point doublings. One of the points has 1 as a z-coordinate, then the required field arithmetic operation is reduced [25]. A scalar multiplication of an elliptic curve point is performed by the Addition chain as shown in Figure 12, i.e., the intermediate point S is always updated, while the point P is fixed. Therefore, the point P could be set as (X/1, Y/1). By this method, the required operations can be reduced from 16 multiplications to 13 multiplications as shown in figure 27. The Jocobian coordinate systems have similar results.

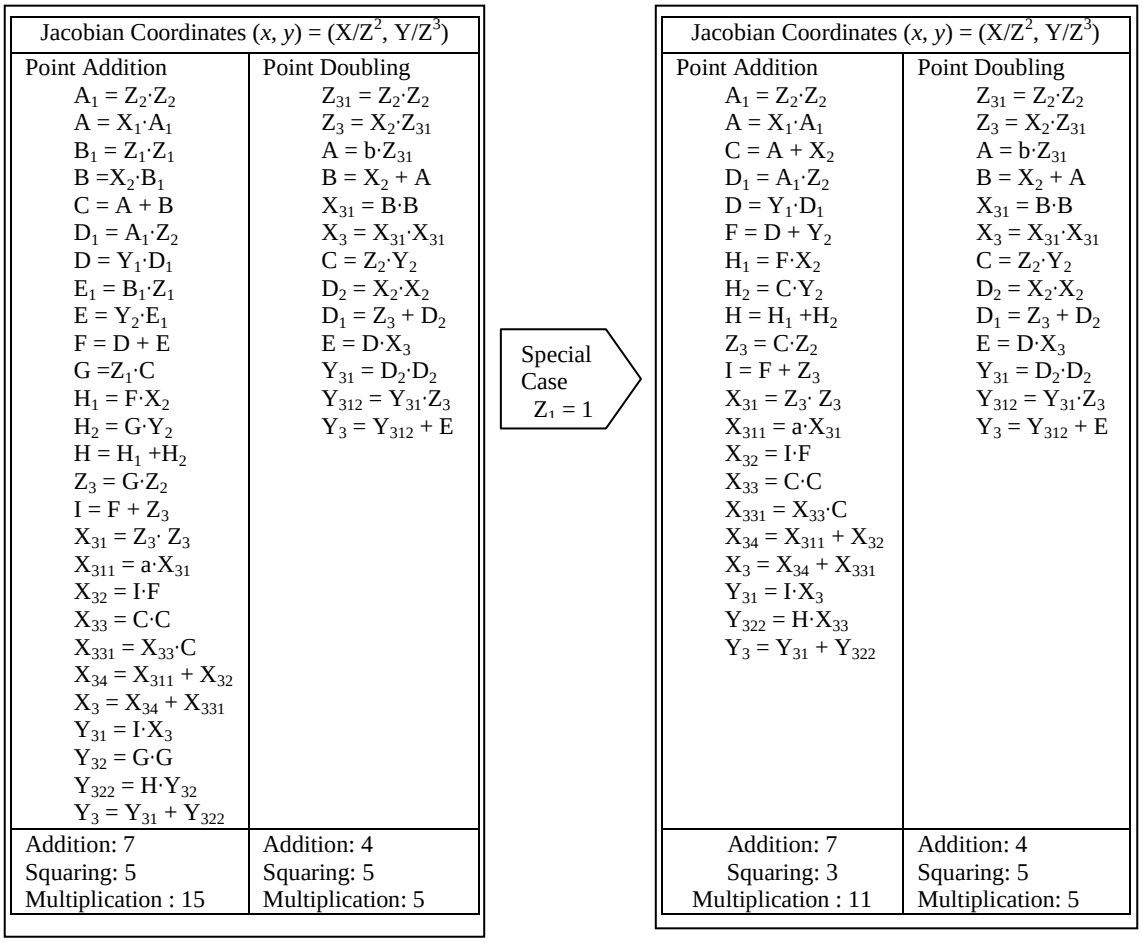


Figure 28: Jacobian Coordinates and Mixed Coordinates with Affine systems

Compared to Projective coordinate systems, Jacobian coordinate systems need one less multiplication. Similar to Projective coordinate systems, if one point has 1 as a z-coordinate, then the required multiplications are reduced from 15 to 11 and squaring is also reduced from 5 to 3.

6. The Proposed Method

Basically, RSA/ECC uses addition chains to reduce the number of multiplications or exponentiations. However, these algorithms have strong data dependencies as discussed in Section 3, which make multiplications more difficult to parallelize. Fan *et al.* [25] proposed a parallel Montgomery Algorithm to improve performance. Although their method improves performance, their study was done using idealistic field sizes that result in the same workload for each core. However, ECC in real cryptographic systems uses a prime number of bits for the field size, such as $GF(2^{131})$, $GF(2^{163})$, $GF(2^{193})$ and $GF(2^{233})$, which are recommended by the U.S. Government [39]. In the RSA case, from the RSA factoring challenge [51], some field sizes, i.e., RSA-576 and RSA-704, result in idle cores. A cryptosystem that uses these field sizes results in uneven workloads and in turn causes idle cores, making it vulnerable to power or timing analysis. Therefore, a counter measure is proposed for SCA. The proposed method is to rearrange point operations with dummy instructions and parallelize the multiplier which involves distributing the loads equally across cores to make power analysis difficult.

In order to illustrate the problem of unbalanced load, Figure 30 shows load distribution of Fan's multiplication over $GF(2^{131})$ on four cores with a word size of 32 bits. In the 2^{nd} iteration, one core has more work than the others because it performs an additional 3-bit multiplication causing the other three cores to be idle. Even if a cryptosystem uses eight cores, it will have a similar problem because three of the cores will be idle.

As can be seen in Figure 30, these idle cores could be used in SCA. By finding the occurrence of idle cores we can figure out the secret value of d_i in the addition chain in some RSA or ECC cases, i.e., RSA-704[51], ECC over $GF(2^{131})$, $GF(2^{193})$, etc. In RSA, we can find the number of multiplications by observing the occurrence of idle cores and timing. The occurrence of idle cores can be checked by similar methods with cache analysis, particularly I-cache, as we mentioned in the section on SCA. We can also figure out the secret value d_i in ECC by observing the occurrence of idle cores. Since elliptic curve addition and doubling have been shown for a different number of field operations, the occurrence pattern is shown differently as described in Figure 28. This figure showed a modified doubling process in a Jacobian coordinate systems in Figure 27 after checking for RAW(Read After Write) dependencies.

As shown in Figure 28, the different number of fields operations between Point Addition and Point Doubling might cause vulnerability to SCA. For example, when we assume each point operation processed in-order, we can figure out the number of multiplications between additions

by checking the I-cache, i.e., the addition might access different cache blocks instead of previous cache access patterns. Therefore, a 2-2-2-1-5-2 multiplication pattern will show in Point Addition, while a 2-6-2 pattern will show in Point Doubling. The difference of multiplication occurrences in Points Operation can also be seen in the projective coordinates system as shown Figure in 27. Therefore, the attacker can deduce the secret value of d_i from patterns; If the attacker observed the doubling pattern twice, then he could deduce the value of d_i as 1. If he got addition pattern followed by doubling, then he could guess the value of d_i as 0. Thus, the elliptic point operations need to be shown similarly from the attacker. For this purpose, dummy instructions are used. As we mentioned in section 4, dummy instructions must have same weight. Figure 29(a) shows the difference between two point operations. As shown in Figure (a), Point doubling has 3 less additions, 2 more squaring operations, and 6 less multiplications. Thus, 2 squarings in Point doubling need to be changed into multiplication, i.e., $D_2 = X_2 \cdot X_2$ is substituted with $D_{21}=X_2 \cdot Z_1$ and $D_2=X_2 \cdot D_{21}$, $X_{31}=B \cdot B$ is also substituted with $X_{311} = B \cdot Z_1$ and $X_{31} = B \cdot X_{31}$. Then, Point doubling arranges field operations as shown Figure 29(b). After arrangement, the appropriate dummies (multiplication denoted as a Dummy(mul) or addition as a Dummy(+)) are inserted as described in Figure 29(c).

Jacobian Coordinates $(x, y) = (X/Z^2, Y/Z^3)$	
Point Addition	Point Doubling
$A_1 = Z_2 \cdot Z_2$	$Z_{31} = Z_2 \cdot Z_2$
$A = X_1 \cdot A_1$	$A = b \cdot Z_{31}$
$C = A + X_2$	$B = X_2 + A$
$D_1 = A_1 \cdot Z_2$	**$D_2 = X_2 \cdot X_2$**
$D = Y_1 \cdot D_1$	$C = Z_2 \cdot Y_2$
$F = D + Y_2$	**$X_{31} = B \cdot B$**
$H_1 = F \cdot X_2$	$Y_{31} = D_2 \cdot D_2$
$H_2 = G \cdot Y2$	$X_3 = X_{31} \cdot X_{31}$
$H = H_1 + H_2$	$Z_3 = X_2 \cdot Z_{31}$
$Z_3 = C \cdot Z_2$	$D_1 = Z_3 + D_2$
$I = F + Z_3$	$D = D_1 + C$
$X_{31} = Z_3 \cdot Z_3$	$E = D \cdot X_3$
$X_{311} = a \cdot X_{31}$	$Y_{312} = Y_{31} \cdot Z_3$
$X_{32} = I \cdot F$	$Y_3 = Y_{312} + E$
$X_{33} = C \cdot C$	
$X_{331} = X_{33} \cdot C$	
$X_{34} = X_{311} + X_{32}$	
$X_3 = X_{34} + X_{331}$	
$Y_{31} = I \cdot X_3$	
$Y_{322} = H \cdot Y_{32}$	
$Y_3 = Y_{31} + Y_{322}$	
Addition: 7 Squaring: 3 Multiplication : 11	Addition: 4 Squaring: 5 Multiplication: 5

(a)

Jacobian Coordinates
Point Doubling
$Z_{31} = Z_2 \cdot Z_2$
$A = b \cdot Z_{31}$
$B = X_2 + A$
$D_{21} = X_2 \cdot 1$
$D_2 = X_2 \cdot D_{21}$
$C = Z_2 \cdot Y_2$
$X_{311} = B \cdot 1$
$Y_{31} = D_2 \cdot D_2$
$X_{31} = B \cdot X_{311}$
$Z_3 = X_2 \cdot Z_{31}$
$X_3 = X_{31} \cdot X_{31}$
$D_1 = Z_3 + D_2$
$D = D_1 + C$
$E = D \cdot X_3$
$Y_{312} = Y_{31} \cdot Z_3$
$Y_3 = Y_{312} + E$
Addition: 4 Squaring: 3 Multiplication: 9

(b)

Jacobian Coordinates
Point Doubling
$Z_{31} = Z_2 \cdot Z_2$
Dummy(mul)
$A = b \cdot Z_{31}$
$B = X_2 + A$
$D_{21} = X_2 \cdot 1$
Dummy (+)
$D_2 = X_2 \cdot D_{21}$
$C = Z_2 \cdot Y_2$
Dummy (+)
$X_{311} = B \cdot 1$
Dummy (+)
$Y_{31} = D_2 \cdot D_2$
$X_{31} = B \cdot X_{311}$
$Z_3 = X_2 \cdot Z_{31}$
$X_3 = X_{31} \cdot X_{31}$
Dummy(mul)
$D_1 = Z_3 + D_2$
$D = D_1 + C$
$E = D \cdot X_3$
$Y_{312} = Y_{31} \cdot Z_3$
$Y_3 = Y_{312} + E$
Addition: 4 + **3** Squaring: 3 Multiplication: 9 + **2**

(c)

Figure 29: The Difference of Elliptic curve PA and PD

Furthermore, each operation performs a long-bit number operation, thus it can also implement parallel processing like Fan's MM. Then, each one may also produce idle cores as shown in Figure 30. These idle cores may give useful more information than the cache analysis does. By observing occurrence patterns of idle cores, we can distinguish each field operation since idle cores will not access I-cache or D-cache for certain periods. By using cache analysis combined with idle core analysis, we can figure out the secret value *d* more easily.

Moreover, the last word of each *GF* is less than one word size. A smaller-sized multiplier can be used instead of a full-sized multiplier to reduce power consumption.

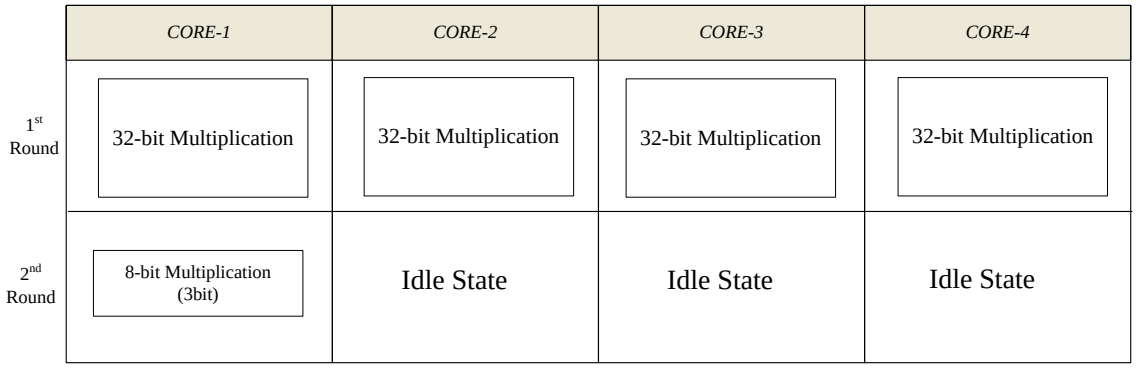


Figure 30: Multiplication over GF(2^{131}) for Word Size 32 bits.

The characteristics of cryptosystems, such as the number of cores and GF size are known as described in Figure 31, therefore, the *multiplier adjuster* can be defined for power efficiency and dummies in Figure 29(c) make analysis more difficult. We propose the idea of a *multiplier adjuster* on multi-cores that uses a power-performance improving technique described in Figure 30. Using this technique, we can expect a more power efficient multiplier.

Types	GF131	GF163	GF193	GF233	GF239	GF283	GF409	GF571
Required Iterations	5	6	7	8	8	9	1	1
Required Round	5/3/2/1	6/3/2/1	7/4/2/1	8/4/2/1	8/4/2/1	9/5/3/2	13/7/4/2	18/9/5/3
Idle core	0/1/3/3	0/0/2/2	0/1/1/1	0/0/0/0	0/0/0/0	0/1/3/7	0/1/3/3	0/0/2/6

* Notation : A/B/C/D A : Single core B:Dual cores C: Quad cores D: Eight cores

Figure 31: Characteristics of GFs

As shown in Figure 31, *GF*(2^{131}) requires 5 iterations with each iteration performing a 32-bit operation. In case of a quad-core, the 5^{th} iteration requires execution time to perform a 3-bit operation causing the rest of the cores to become idle. Similarly, *GF*(2^{163}) requires 6 iterations and causes two cores to become idle. The last word size is also a 3-bit. Thus, a smaller-sized multiplier can be used. In general, the relationship between field size and the number of idle cores is given as

$$\text{Number of idle cores} = p \cdot \left\lceil \frac{f}{p \cdot s} \right\rceil - \left\lceil \frac{f}{s} \right\rceil,$$

where f is the field size, s the system word size, and p the number of cores. The last word size can also be defined as

$$\text{Size of Last word} = f - \left\lfloor \frac{f}{s} \right\rfloor \cdot s$$

Section 6.1 describes the parallelization of the original source code described in [38] using Pthreads to run on multi-cores. Then, Section 6.2 discusses how and what dummy instructions are assigned during Point operation. Finally, Section 6.3 presents the multiplier adjuster technique.

6.1. Parallel Implementation of Modular Multiplication

Rosing provided simple code for ECC [50]. He uses a conventional multiplication algorithm. Rosing 's code is actually not implemented in parallel. As shown in Figure 32, the function *poly_mul*( ) consists of the function *poly_mul_partial*( ) and *poly_div*( ). The function *poly_mul*( ) gets two inputs: *a* and *b*. Then it outputs *a* * *b* mod irreducible polynomial. The function *poly_div*( ) returns the *quotient* and *remainder* with two inputs: *top* and *bottom.*

```
/*  Polynomial multiplication modulo poly_prime.  */

void poly_mul(a, b, c)                          /* c = a*b mod irreducible polynomial */
FIELD2N *a, *b, *c;
{
        DBLFIELD temp;
        FIELD2N  dummy;

        poly_mul_partial(a, b, &temp);
        poly_div(&temp, &poly_prime, &dummy, c);
}

void poly_div(top, bottom, quotient, remainder)          /* quotient = top/bottom */
DBLFIELD *top;                                            /* remainder = top%bottom */
FIELD2N  *bottom, *quotient, *remainder;
```

Figure 32: Original Code – poly_mul( ) & poly_div( )

```
void poly_mul_partial(a, b, c)                    /* c= a*b   */
FIELD2N *a, *b;
DBLFIELD *c;
{
        INDEX i, bit_count, word;
        ELEMENT mask;
        DBLFIELD B;

/*  clear all bits in result  */
        dblnull(c);

/*  initialize local copy of b so we can shift it  */
        sngltodbl(b, &B);

/*  for every bit in 'a' that is set, add present B to result  */

        mask = 1;
        for (bit_count=0; bit_count<NUMBITS; bit_count++)
        {
                word = NUMWORD - bit_count/WORDSIZE;
                if (mask & a->e[word])
                {
                        DBLLOOP(i) c->e[i] ^= B.e[i];
                }
                /* multiply copy of b by x  */
                   mul_shift( &B);
                   /* shift mask bit up  */
                mask <<= 1;
                   /* when it goes to zero, reset to 1  */
                if (!mask) mask = 1;                }
}
```

Figure 33: Original Code – poly_mul_partial( )

The function *poly_mul_partial*(), as shown Figure 33, performs general multiplication with two inputs: *a* and *b*. It returns the value *c* which is a long-bit number. The overall process is shown below in Figure 34. *poly_mul_partial*() gives the result t9 ~ t1, and *poly_div*( ) computes the c4~c0.

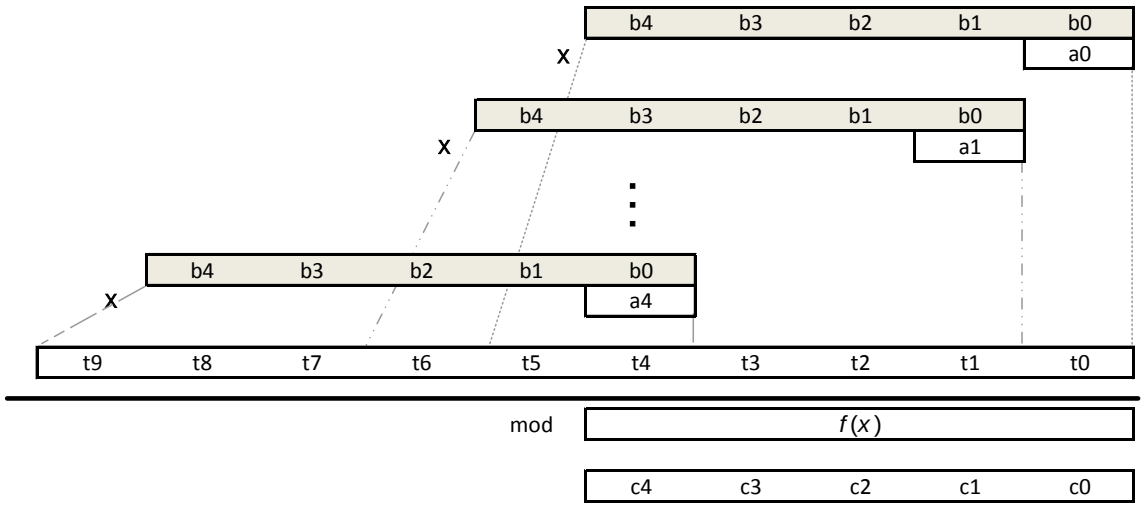


Figure 34: Conventional MM operation

We have used conventional modular multiplication instead of Montgomery Multiplication, since we want to see the occurrence of the idle core and the effect of dummy instructions. Suppose modular multiplication is performed on A and B in GF(2^{131}), where A = (a_4, a_3, a_2, a_1, a_0) and B = (b_4, b_3, b_2, b_1, b_0).

The result of multiplication T = (t_9, t_8, t_7, t_6, t_5, t_4, t_3, t_2, t_1, t_0) consists of the sum of partial products $a_i \times B$ ($0 \le i \le 5$) as described in figure 26. The partial product only affects some part of T; i.e. a0 × B affects the (t5, t4, t3, t2, t1, t0) in T and a1 × B affects the (t6, t5, t4, t3, t2, t1).

Since the size of result T is larger than the size of $f(x)$, we can get the result C of A × B (mod $f(x)$) by using modular operation.

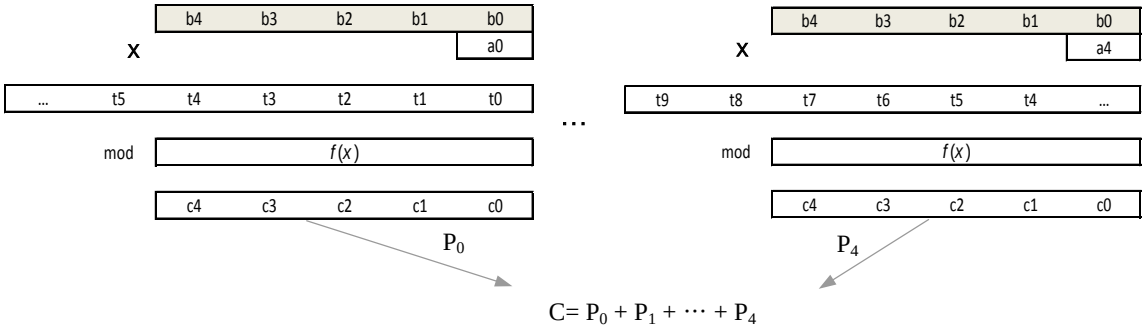


Figure 35: Parallelizing MM operation

Like Figure 35, the conventional multiplications can be parallelized since the computation of partial products needs only a_i ($0 \le i \le 4$) and B. Each intermediate result $P_0 \sim P_4$ is obtained independently. Therefore, we can execute sub multiplication operations in parallel. Using this property, the original source code has been modified to fit parallel processing.

The modified source code from Figure 33 using pThread is shown in Figure 37. The modified function *poly_mul*( ) sets the number of threads. This thread calls the function *mul_sub*( ). The function *mul_sub*( ) is combined with *poly_mul_partial*( ) and *poly_div*( ) as shown in Figure 38. The function *mul_sub*( ) uses the structure SUB as shown Figure 36 since pThread cannot accept multiple variables.

```
typedef struct {
        INDEX    i;
        FIELD2N  *a, *b, *c;
}  SUB;
```

Figure 36: Data Structure – SUB

```
void poly_mul(a, b, c)
FIELD2N *a, *b, *c;
{
        SUB *input[NUMWORD]
        FIELD2N total,temp[NUMWORD];
        pthread_t p_thread[NUMWORD];
        int t, q, status;
        INDEX i;

        for(t=0;t<NUMWORD;t++){
                input[t] = (SUB *) malloc(4 * sizeof(FIELD2N *));
                input[t]->i=t;
                input[t]->a=a;
                input[t]->b=b;
                input[t]->c=&temp[t];
                pthread_create(&p_thread[t], NULL, mul_sub, (void *)input);
        }

        for(q=0;q<NUMWORD;q++){
                pthread_join(p_thread[q], (void *)&status);
        }

        for(t=0;t<NUMWORD;t++){
                SUMLOOP(i) total.e[i] ^= temp[t].e[i];
        }
        copy(&total,c);
}
```

Figure 37: Modified Code – poly_mul( )

```
void mul_sub(para)
SUB *para;
{
        INDEX i, t, bit_count;
        ELEMENT mask, ta;
        FIELD2N  dummy;
        DBLFIELD B, C;
        t=para->i;
        ta = para->a.e[NUMWORD-t];
        sngltodbl(para->b, &B);

        mask = 1;
        for (bit_count=0; bit_count<WORDSIZE; bit_count++)
        {
                if ( (ta) & mask )
                {
                        DBLLOOP(i) C.e[i-t] ^= B.e[i];
                }
                mul_shift( &B);      /* multiply copy of b by x  */
                mask <<= 1;          /* shift mask bit up  */
                if (!mask) mask = 1;            /* when it goes to zero, reset to 1  */
        }

        poly_div(&C, &poly_prime, &dummy, &para->c);
        dblnull(&C);
}
```

Figure 38: Modified Code – mul_sub( )

6.2. Inserting Dummy Instruction

Inserting dummy instructions involves simply inserting some dummy functions. However, as we saw before, the field operation causes idle cores depending on *GF*s. Full dummy insertion, called *low-level dummy inserting*, for every idle core is quite expensive. Low-level dummy inserting might also be vulnerable to cache analysis where the attacker could figure out the position of point *P* in the memory since the point *P* is not accessed during the Point doubling operations. In the case of PD-PD, the point *P* is not accessed during the 32 field operations in Projective coordinate systems, 28 field operations in Jacobian coordinate systems. The low-level dummy inserting is shown in Figure 39.

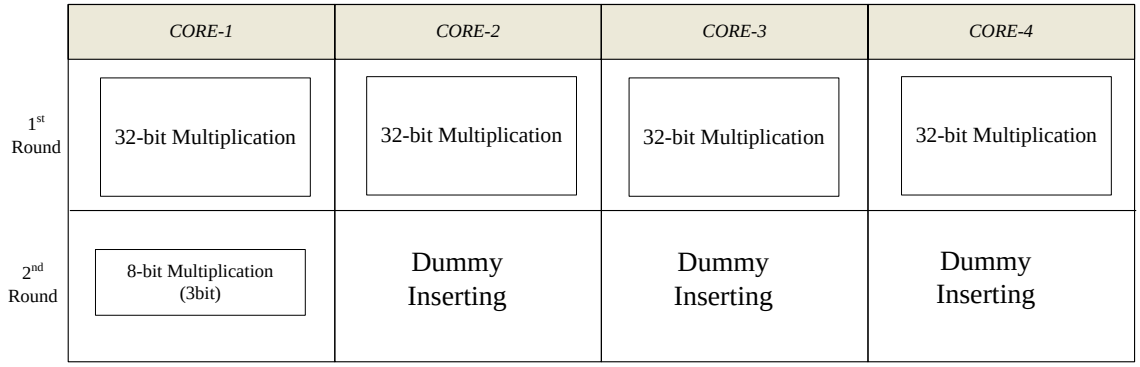


Figure 39:Low-level Dummy inserting over $GF(2^{131})$ for word size 32 bits.

Compared to low-level inserting, we propose *high-level dummy inserting*. Point arithmetic operations are quite different as shown in Figure 28. Thus, as mentioned before, the different characteristics can be used in SCA by the attacker. Figure 40 shows the high-level dummy inserting techniques.

Point Addition	Point Doubling
$A_1 = Z_2 \cdot Z_2$	$Z_{31} = Z_2 \cdot Z_2$
$A = X_1 \cdot A_1$	**Dummy1(mul)**
$C = A + X_2$	$A = b \cdot Z_{31}$
$D_1 = A_1 \cdot Z_2$	$B = X_2 + A$
$D = Y_1 \cdot D_1$	$D_{21} = X_2 \cdot 1$
$F = D + Y_2$	**Dummy 2(+)**
$H_1 = F \cdot X_2$	$D_2 = X_2 \cdot D_{21}$
$H_2 = G \cdot Y2$	$C = Z_2 \cdot Y_2$
$H = H_1 + H_2$	**Dummy 3(+)**
$Z_3 = C \cdot Z_2$	$X_{311} = B \cdot 1$
$I = F + Z_3$	**Dummy4 (+)**
$X_{31} = Z_3 \cdot Z_3$	$Y_{31} = D_2 \cdot D_2$
$X_{311} = a \cdot X_{31}$	$X_{31} = B \cdot X_{311}$
$X_{32} = I \cdot F$	$Z_3 = X_2 \cdot Z_{31}$
$X_{33} = C \cdot C$	$X_3 = X_{31} \cdot X_{31}$
$X_{331} = X_{33} \cdot C$	**Dummy5(mul)**
$X_{34} = X_{311} + X_{32}$	$D_1 = Z_3 + D_2$
$X_3 = X_{34} + X_{331}$	$D = D_1 + C$
$Y_{31} = I \cdot X_3$	$E = D \cdot X_3$
$Y_{322} = H \cdot Y_{32}$	$Y_{312} = Y_{31} \cdot Z_3$
$Y_3 = Y_{31} + Y_{322}$	$Y_3 = Y_{312} + E$

Figure 40: High-level Dummy inserting

As described in Figure 40, 5 dummy instruction sets (2 field multiplication dummy sets and 3 field addition sets) are required in high-level dummy inserting. The point *P* will be accessed in dummy instructions. Therefore, the attacker cannot find the difference between PA and PD. To make same weight with PA operations, we inserted two types of dummy instructions into PD, i.e., Dummy1 and 5 have multiplication whereas dummy 2 to 4 have addition chain. The modified is source code shown in Figure 41.

```
void Jpoly_edbl (dummy, p1, p3, curv)
JPOINT *dummy, *p1, *p3;
CURVE *curv;
{
   INDEX   i;
// for Jacobian coordinate system//
   FIELD2N  A,B,X31,C,D,D2,,D21,D1,E,Y31,Y312,Z31,X311;
   FIELD2N  Dummy1, Dummy2, Dummy3, Dummy4, Dummy5;
   int q, thid;

   poly_mul( &p1->z, &p1->z, &Z31);
   //dummy1
   poly_mul( &dummy->x, &p1->z, &Dummy1);
   poly_mul( &curv->a6, &Z31, &A);
   SUMLOOP (i) B.e[i] = p1->x.e[i] ^ A.e[i];
   poly_mul( &p1->x, &dummy->z, &D21);
   //dummy2
   SUMLOOP (i) Dummy2.e[i] = p1->x.e[i] ^ Dummy1.e[i];
   poly_mul( &D21, &dummy->z, &D2);
   poly_mul( &p1->z, &p1->y, &C);
   //dummy3
   SUMLOOP (i) Dummy3.e[i] = Dummy1.e[i] ^ dummy->x.e[i];
   poly_mul( &B, &dummy->z, &X311);
   //dummy4
   SUMLOOP (i) Dummy4.e[i] = Dummy3.e[i] ^ Dummy2.e[i];
   poly_mul( &D2, &D2, &Y31);
   poly_mul( &B, &X311, &X31);
   poly_mul( &p1->x, &Z31, &p3->z);
   poly_mul( &X31, &X31, &p3->x);
   // Dummy5
   poly_mul( &X31, &p3->x, &Dummy5);
   SUMLOOP (i) D1.e[i] = D2.e[i] ^ p3->z.e[i];
   SUMLOOP (i) D.e[i] = D1.e[i] ^ C.e[i];
   poly_mul( &D, &p3->x, &E);
   poly_mul( &Y31, &p3->z, &Y312);
   SUMLOOP (i) p3->y.e[i] = Y312.e[i] ^ E.e[i];
}
```

Figure 41: Modified Code – Jpoly_edbl( )

6.3. Multiplier Adjuster

Inserting random instructions causes extra power consumption. In order to achieve a power efficient implementation, we propose the Adjusted Multiplier for the last iteration in GF. The motivation behind the Adjusted Multiplier is that the last iteration is always less than the register word size. Thus, multiplication may be performed using only a half- or quarter-sized multiplier.

For example, if we use $GF(2^{131})$ in a 32-bit quad-core system, dividing the field size by the register word size results in the last word to be 3 bits. Thus, a quarter-size a multiplier can be used to reduce power and increase performance since the 32-bit multiplier scans all operands even though the leading zeros only exist. Figure 42 shows Modified Source code from previous

source code in Figure 38. The *bit_count* is modified for the last word multiplication. As described, the modified source code determines the number of iterations by *bit_count*. It reduces the computation workloads.

```
void mul_sub(para)
SUB *para;
{
        INDEX i, t, bit_count, word_count;
        int q;
        ELEMENT mask, ta;
        FIELD2N a,b, dummy, c;
        DBLFIELD B, C;
        copy(para->a, &a);
        t=para->i;
        ta = a.e[NUMWORD-t];
        copy(para->b, &b);
        sngltodbl(&b, &B);
         // multiplier adjuster
        word_count = WORDSIZE
        if (t == NUMWORD)  word_count= NUMBITS - WORDSIZE*(t-1);

        mask = 1;
        for (bit_count=0; bit_count<word_count; bit_count++)
        {

                if ( (ta) & mask )
                {
                        DBLLOOP(i) C.e[i-t] ^= B.e[i];
                }
                mul_shift( &B);     /* multiply copy of b by x  */
                mask <<= 1;         /* shift mask bit up  */
                if (!mask) mask = 1;            /* when it goes to zero, reset to 1  */
        }

        poly_div(&C, &poly_prime, &dummy, &c);
        SUMLOOP(i) para->c->e[i] = c.e[i];
        dblnull(&C);

        sesc_exit(0);
}
```

Figure 42: Multiplier Adjuster

7. Simulation Study

This section discusses the simulator, experimental method, and environment parameters. A simulator is an important component of an architectural study. There are two types of simulators; full-system simulators and application-only simulators. Although full-system simulators are preferable, server class workloads are difficult to run on the simulators because they require simulations of all aspects of a computer system, including OS, networking, storage, I/O, etc. On the other hand, application-only simulators can easily verify application performance on a single computing environment.

Many simulators have been used in computer architecture research. These simulators include Wattch [21], SimpleScalar [22], and Simics [23]. Each of these simulators has its own limitations. Wattch is an architectural-level power analysis tool that runs on top of other microarchitecture simulators. SimpleScalar does not support multi-cores, but it does have support for multithreading. Although Simics allows users to simulate multi-core and multithreading, it is proprietary and has limitations for evaluating power. Therefore, we used SuperESCalar (SESC) simulator [20] for evaluating performance and power of our target system. SESC provides a cycle-accurate timing model using a MIPS Interpreter (MINT) for functional simulation. It also allows power evaluation of multi-cores and multithreading. SESC is an event-driven simulator and the actual instructions are executed by an emulation module using MIPS Instruction Set Architecture (ISA).

The purpose of the simulation study is to analyze power consumption characteristics of different methods of preventing side-channel attacks on multi-core based cryptographic systems.

7.1. Simulation Environment

SESC models MIPS R10K microprocessor [53] with a variety of options including the number of CPUs with either in-order or out-of-order execution, operating frequencies and memory configurations. The MIPS 10K microprocessor has separate 32 KB I-Cache and D-Cache. It also has L2 cache ranging in size from 512 KB to 8 MB. A multiprocessor system can be created with up to four processors by using a cluster bus configuration as shown in Figure 43. A Cluster bus is created by attaching the system interfaces of up to four R10K processors with an external agent (called the *clustering coordinator*). The clustering coordinator is responsible for managing the flow of data within the cluster. For embedded systems, the most popular processor is the ARM

processor. Since SESC does not support ARM processors, the simulation parameters had to be modified for an embedded ARM processor.

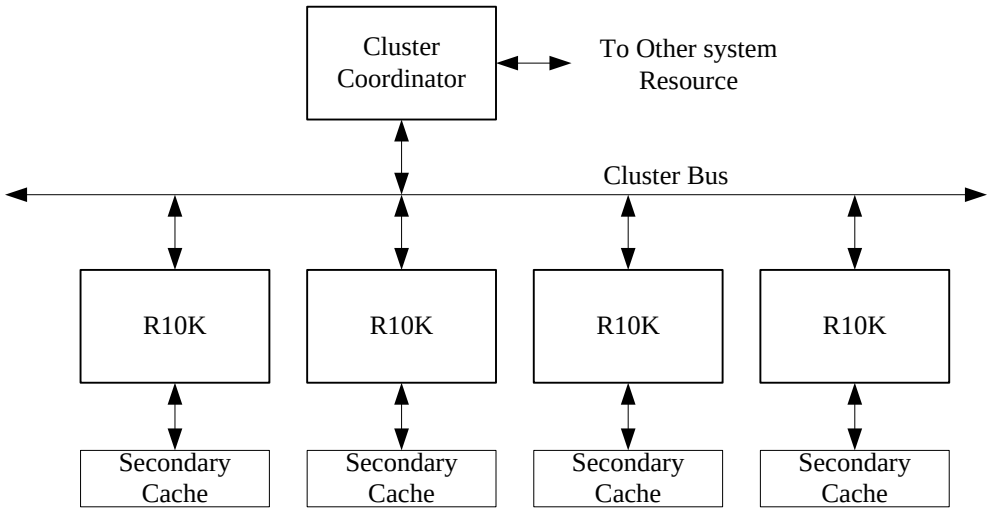


Figure 43: MIPS R10K multiprocessor configuration using the cluster bus

Figure 43 shows the architectural layout used in the simulation study, which are similar to the ARM11MP system [34]. As shown in Figure 44, the ARM 11MP system also supports up to four cores. The *Snoop Control Unit* (SCU) has a similar function to Cluster Coordinator. I-Cache and D-Cache are separated. Cache size is configured from 16 KB to 64 KB. However, ARM 11MP accesses L2 cache through the *Advanced eXtensible Interface* (AXI), while R10K core has its own L2 cache interface. Therefore, SESC can be similarly configured to model ARM 11MP.

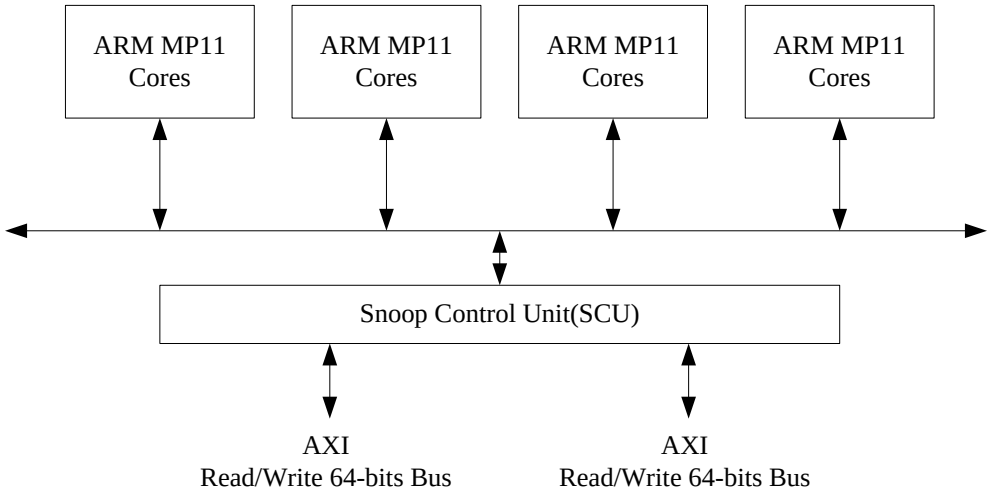


Figure 44: ARM 11MP processor Configuration

```
procsPerNode  = 1    /* number of cores */
cacheLineSize = 32
                        ⋮
################################
# clock-panalyzer input      #
################################
[techParam]
clockTreeStyle = 1              # 1 for Htree or 2 for balHtree
tech       = 70                 # nm
frequency  = 620e6              # Hz
skewBudget = 20                 # in ps
                        ⋮
################################
# MEMORY SUBSYSTEM          #
################################

# instruction source
[IMemory]
deviceType    = 'icache'
size          = 32*1024
assoc         = 2
                        ⋮
# data source
[DMemory]
deviceType    = 'cache'
size          = 32*1024
                        ⋮
```

Figure 45: Configuration file in SESC(*sesc.conf*)

Although ARM11MP supports only up to four cores, the number of cores in our simulation study goes up to eight. SESC uses *.conf* file to configure its simulation environment as shown in Figure 45. In our experiment, we modify the number of cores, and cache size. The other parameters remain are unchanged from the default values. In our simulation, we have set the number of cores as a single-core, clock frequency of 620 MHz and L1 I/D-Cache as 32 KB as described in Figure 45. In order to test a dual-core environment, the number of cores is changed (**procsPer-Node**) to 2 in *sesc.conf*.

7.2. SESC API

The source code for our experiment was obtained from [38], and parallelized to run on multi-cores as well as modified to work with the SESC simulator. We have also modified the multiplication operation for the last-word in $GF(2^k)$.

Figure 46 shows a simple parallel program using Pthreads. The program creates 5 threads with each thread having a unique *thread id* and generates "*Hello Beaver!*" as output.

```
#include  <pthread.h>
#include  <stdio.h>
#define NUM_THREADS 5

void *print_hello_beaver(void *threadid)
{
      printf("\n %d: Hello Beaver!\n", threaded);
      pthread_exit(NULL);
}


int main()
{
      pthread_t threads(NUM_THREADS);
      int i;
      for (i = 0; i < NUM_THREADS; i ++) {
            printf("Creating thread %d\n",i);
            pthread_create(&threads[i], NULL, print_hello_beaver, (void *) i);
      }
      pthread_exit(NULL);
}
```

Figure 46: A Pthreads program

In order to run this program in SESC, some modifications are needed. First, the program needs to use <sescapi.h> instead of <pthread.h>. Then, all thread related functions should be replaced with the functions provided by SESC API. Figure 47 shows SESC API and its relation to Pthreads and the SESC version of the program *Hello Beaver!*. The sesc_init( ) function initializes the threads library and prepares the thread creation. sesc_init( ) needs to be executed before threads are created. The SESC uses sesc_spawn( ) function instead of pthread_create( ) to create threads. The other functions are described in Figure 47.

SESC API	Pthread function
sesc_init( ); : initialize the library and prepare thread creation	none
sesc_spawn(); : create threads, allocates each thread to a particular processor	pthread_create( );
sesc_wait(); : block processing until threads activation is completed	pthread_join();
sesc_exit(); : terminates the current thread activated.	pthread_exit();

```
#include  "sescapi.h"
#include  <stdio.h>
#define NUM_THREADS 5

void *print_hello_beaver(void *threadid)
{
      printf("\n %d: Hello Beaver!\n", threaded);
      sesc_exit(0);
}
int main()
{
      int i;
      sesc_init();
      for (i = 0; i < NUM_THREADS; i ++) {
            printf("Creating thread %d\n",i);
            sesc_spawn((void *) *print_hello_beaver, (void *) i, 0);
      }
      pthread_exit(NULL);
      sesc_exit(0);
}
```

Figure 47: SESC version of threading program

7.3. Simulation Methods

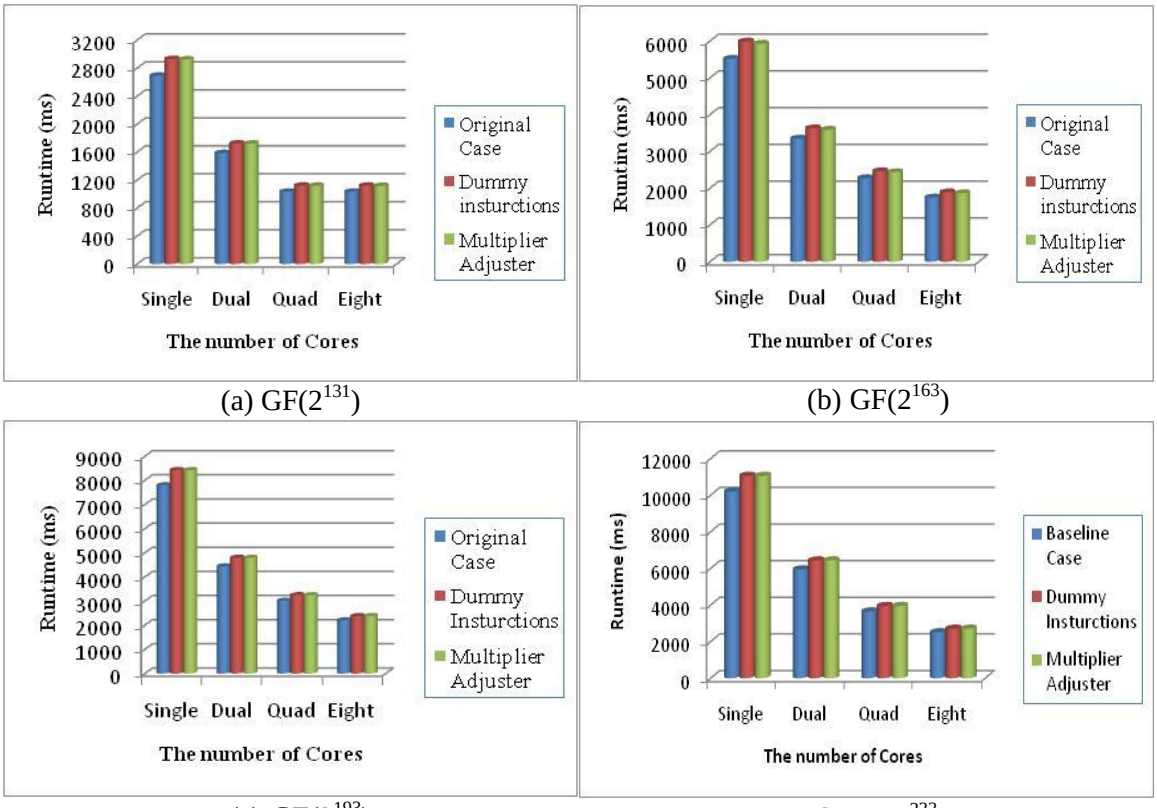
Each ECC operation using Addition Chain requires one point-doubling operation, and either one or no point-addition operation depending on the secret key values. We can assume that two point-doubling and one point-addition operations are used for every two bits. Furthermore, each point operation needs different field operations depending on the coordinate system. Table 4 shows the required number of field operations for each of the following coordinates: *Affine*, *Classical projective* and *Jacobian projective coordinate* system [52]. Therefore, each *k*-bit operation needs *k* point-doubling and *k*/2 point-addition for a simple elliptic scalar multiplication. Therefore, *3k* (*k* * 2 multplication per point-doubling, *k*/2 * 2 multiplication per point-addition) multiplication are required in Affine coordinates and 19*k* in Mixed Jacobian coordinates. Thus, the number of multiplications is different depending on a coordinates system. In our experiment, we have simulated scalar multiplications on Mixed Jacobian coordinates, which require less multiplications than the others. For the purpose of comparing GFs, we simulate the secret values *d* from 1 to 99 for all cases. To simulate effectiveness of the Multiplier adjuster scheme, we have chosen $GF(2^{131})$, $GF(2^{163})$, $GF(2^{193})$ and $GF(2^{233})$, which causes different numbers of cores to be idle in an eight-core system. All GFs used in the simulation study are recommended in ECC by the U.S. Government [39]. The results are analyzed in terms of performance, power, energy, and energy-delay product based on the following three cases: (1) fixed size word multiplier without dummy instructions, i.e., the baseline case; (2) a fixed size word multiplier with dummy instructions; and (3) a multiplier adjuster with dummy instructions.

Table 4: The number of field operations on each coordinates systems

	Multiplication(PA/PD) (include Squaring)	Addition(PA/PD)	Division(PA/PD)
Affine Coordinates (x, y)	2/2	9/6	1/1
Projective Coordinates $(X/Z, Y/Z)$	17/11	7/5	0/0
Mixed Projective Coordinates $(X/Z, Y/Z)$	14/11	7/5	0/0
Jacobian Coordinates $(X/Z^2, Y/Z^3)$	20/10	7/4	0/0
Mixed Jacobian Coordinates $(X/Z^2, Y/Z^3)$	14/10	7/4	0/0

*PA: Point Addition PD: Point Doubling

7.4. Simulation Results



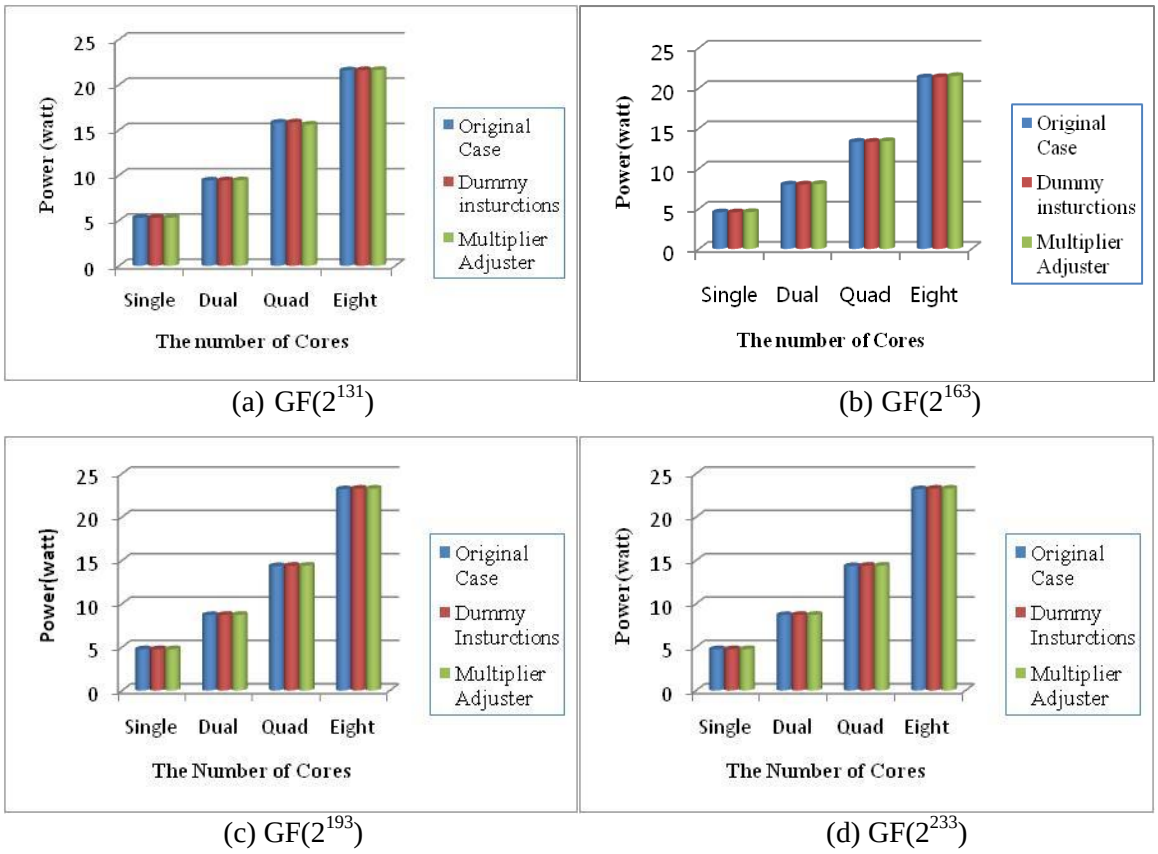
(a) GF(2^{131}) (b) GF(2^{163})

(c) GF(2^{193}) (d) GF(2^{233})

Figure 48: Performance of GFs

Figure 48 shows the performance of GFs on a single-core and different multi-core configurations. Execution times compared to a single-core for all GFs improve by a factor of 1.64 ~ 1.75, 2.42 ~ 2.79, and 2.6 ~ 4.06 when scalar multiplication is performed on dual-cores, quad-cores, and eight-cores, respectively. As shown Figure 48, overhead for dummy instructions causes 6.6 ~ 8.6 % and 7.6 ~ 8.8% increase in execution time when dummy instructions are inserted with and without the multiplier adjuster, respectively.

For *GF* (2^{131}), improvement from quad-cores to eight cores is smaller than other GFs. This is because of the characteristics of this *GF*, i.e., the 5^{th} iteration of *GF*(2^{131}) requires just a 3-bit multiplication in the 2^{nd} round and the other three cores remain idle. Thus, the time required to complete a field operation depends on the last iteration.

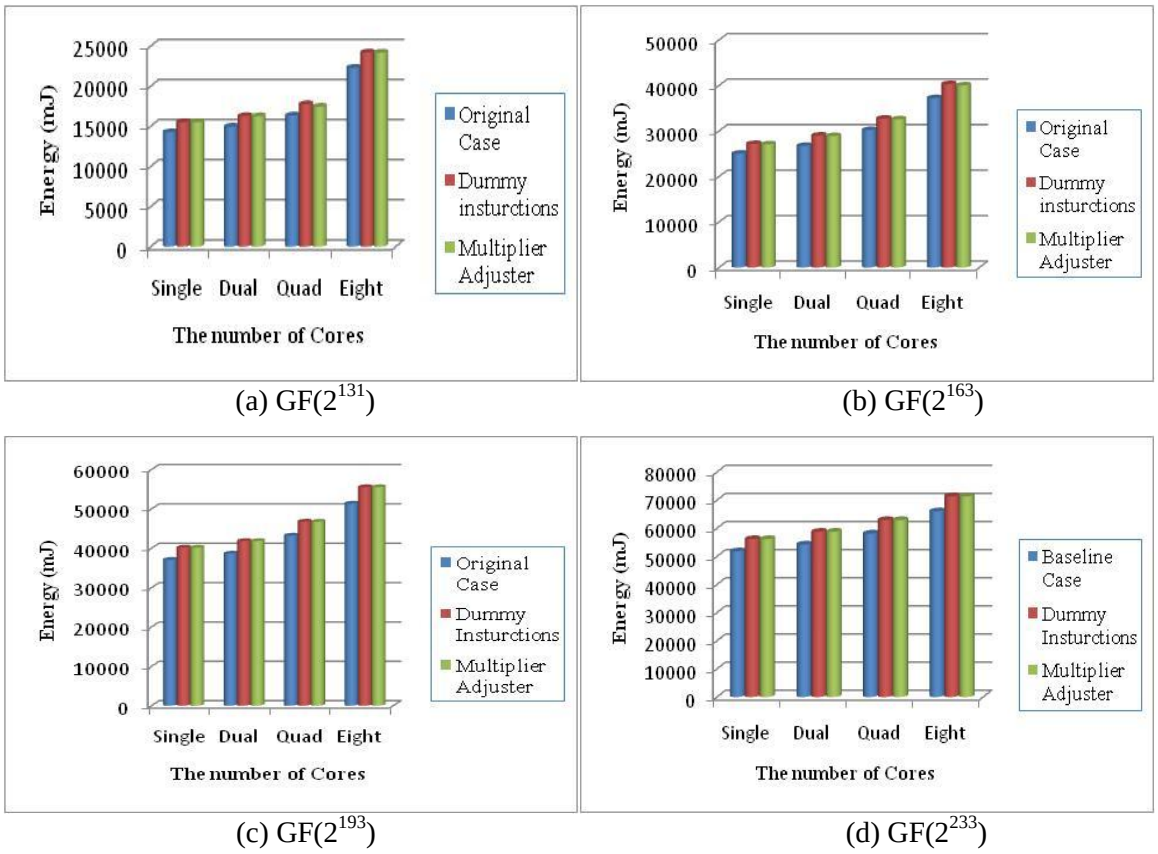


(a) GF(2^{131}) (b) GF(2^{163})

(c) GF(2^{193}) (d) GF(2^{233})

Figure 49: Instantaneous Power of GFs

Figure 49 show that instantaneous power of each *GF* is larger on multi-cores when compared to single cores. The power of all *GF*s increased by a factor of 1.76 ~1.83, 2.92 ~ 3.02 and 4.07 ~5.2 on dual-cores, quad-cores, and eight-cores, respectively. However, the difference among all three cases was only 0.001 ~ 0.03 watts depending on the *GF*s. Thus, the effect of the dummy instructions or the multiplier adjuster is negligible.



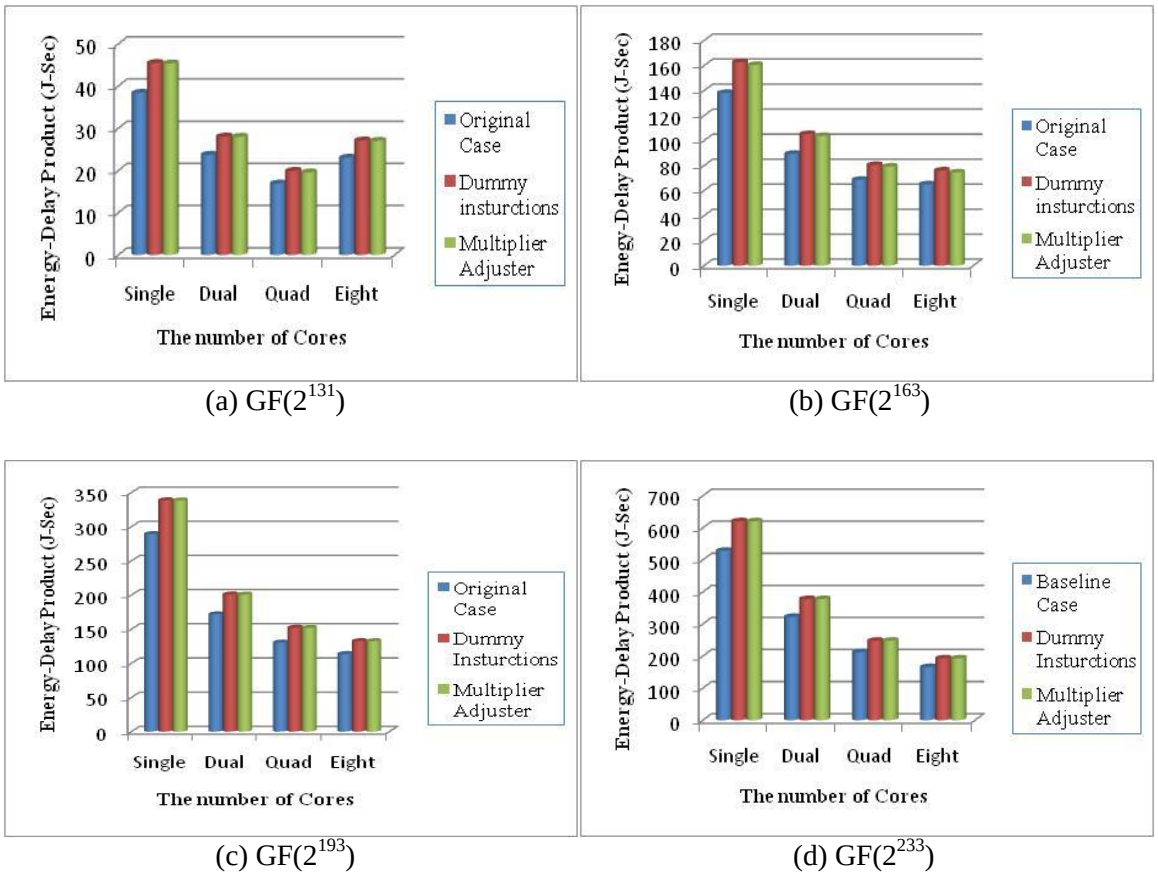
(a) GF(2^{131}) (b) GF(2^{163})

(c) GF(2^{193}) (d) GF(2^{233})

Figure 50: Energy of GFs

Figure 50 shows the energy of GFs, which increased by a factor of 1.04 ~ 1.06, 1.12 ~ 1.16 and 1.27 ~ 1.56 for dual-cores, quad-cores, and eight-cores respectively. Furthermore, as shown in Figure 50, the overhead of dummy instruction causes a 6.7~8.7% and 8.0~8.7% increase in energy with and without the multiplier adjuster, respectively.

These results may lead one to believe that multi-cores are not as power-efficient as a single-core. As discussed in the background, two different systems can have equivalent energy consumption even though one system performs better than the other. Therefore, the system evaluation should be based on both performance and energy. Thus, we use the energy-delay product (EDP) for system evaluation.



(a) GF(2^{131}) (b) GF(2^{163})

(c) GF(2^{193}) (d) GF(2^{233})

Figure 51: Energy-Delay Product of GFs

EDP is also called power-performance. As shown in Figure 51, EDP for all *GF*s except *GF*(2^{131}) is reduced by a factor of 0.59 ~ 0.64, 0.39 ~ 0.49 and 0.31 ~ 0.46 on dual-cores, quad-cores, and eight-cores, respectively. *GF*(2^{131}) causes three idle cores for each iteration in each round. Note that only *GF*(2^{131}) have rounds with three idle iterations. All other GFs do not have this situation because they have less than (*p*-1) idle core for every round. As shown Figure 50, the overhead of dummy instructions causes 14.7~18.0% and 16.5 ~17.8% increase in execution time with and without the multiplier adjuster, respectively.

8. Future work & Conclusion

Our experiment shows that increasing the number of cores does guarantee the increase of performance even though there are some idle cores causes by data dependencies. Furthermore, rearrangement of PD operation with dummy instructions can provide the appropriate protection from SCA, i.e., Timing Analysis, Power Analysis and Cache Attack. Overheads which are cause by rearrangement of this technique can be reduced by a multiplier adjuster. In this thesis, we covered only one possibility for SCA on multi-cores. Therefore, more research is needed to find out other weakness of SCA on multi cores as well as to find a way to reduce data dependencies which cause vulnerabilities for SCA and Power-performance degradation. Also, we implemented multiplier adjuster at the software level. As with adaptive cache, it needs to be implemented in hardware or at the architectural level as we described in the background work.

BIBLIOGRAPHY

[1] Data Encryption Standard (DES). http://en.wikipedia.org/wiki/Data_Encryption_Standard.

[2] W. Stalling. *Cryptography and Network Security, Principles and Practice, 2nd Ed.* Prentice Hall, 1998.

[3] J. F. Kurose, and K. W. Ross. *Computer Networking, A Top-Down Approach Featuring the Internet, 3rd Ed*. Addison Wesley, 2005.

[4] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining Digital Signatures and public-key Cryptosystems. *Communications of the ACM*, volume 21, issue 2, pages 120-126, 1978.

[5] Riverst-Shamir and Adleman Scheme (RSA). http://en.wikipedia.org/wiki/RSA

[6] N. Koblitz, A. Menezes, and S. Vanstone. The State of Elliptic Curve Cryptography. *Designs, Codes and Cryptography*, volume 19, issue 2, pages 173 -193, March 2000.

[7] SEC1. Elliptic Cruve Cryptography, Standards for Efficient Cryptography Group. Available at http://www.secg.org

[8] Hasse's theorem on elliptic curves. Wikipedia. Available at http://en.wikipedia.org/wiki/Hasse%27s_theorem_on_elliptic_curves

[9] M. Morales-Sandoval and C. Feregrino-Uribe. On the Hardware Design of an Elliptic Curve Cryptosystem. *Fifth Mexican International Conference in Computer Science* (ENC'04), pages 64-70, 2004.

[10] B. Schneier. Applied Cryptography: Protocols, Algorithms, and Source Code in C. 2nd Ed.
Willey, 1996.

[11] K. Randall and Nichols. *ICSA Guide to Cryptography*. McGraw Hill, 1999.

[12] W. Diffie and M. E. Hellman. New Directions in Cryptography. *IEEE Transactions on Informa-tionTheory*, volume IT-22, November 1976

[13] Ç. K. Koç. ECE 575 Data Security & Cryptography. 2004 Class Hand out available at http:/islab.oregonstate.edu/koc/ece575/

[14] P. C. Kocher. Timing Attacks on Implementations of Diffie–Hellman, RSA, DSS, and Other Systems. *Advances in Cryptology - CRYPTO '96*. pages 104-113, Lecture Notes in Computer Science, Springer-Verlag, 1996.

[15] P. C. Kocher, J. Jaffe, B. Jun. Differential Power Analysis. Advances in Cryptology –

CRYPTO '99. pages 388-397, Lecture Notes in Computer Science, Springer-Verlag,1999.

[16] K. Tiri and P. Schaumont. Changing the odds against Masked Logic. *Selected Areas of Cryptography (SAC)*, Lecture Notes Computer Science, Springer-Verlag, 2006.

[17] T. Mudge. Power: a first class design constraint. *IEEE computer*, volume 34, issue 4, pages 52-57, April, 2001.

[18] R. E. Grant and A. Afsahi. Power-performance efficiency of asymmetric multiprocessors for multi-threaded scientific applications. *Parallel and Distributed Processing Symposium, IPDPS 2006. 20th International*, pages 8, April, 2006.

[19] El-Badawy and El-Sayed A. Proposed Elliptic Curve for Counter-Measuring both Sign Change

Fault Attacks and Side Channel Attacks. *Radio Science Conference, NRSC 2006. Proceedings of the Twenty Third National*, Pages1-7, March 2006.

[20] J. Renau *et al.*, SESC simulator. available at http://sesc.sourceforge.net/, Jan, 2005.

[21] D. Brooks, V. Tiwari, and M. Martonosi. Watthch: A Framework for architectural-level power analysis and Optimizations. *International Symposium on Computer Architecture (ISCA)*, 2000

[22] T. M. Austin. SimpleScalar. available at http://www.simplescalar.com/

[23] Simics. available at http://www.virtutech.com/

[24] P. Montgomery,"Modular multiplication without trial division"., Mathematicis of Computation, 44(170) p519-512, 1985

[25] J. Fan, K. Sakiyama and I. Verbauwhede, "Elliptic curve cryptography on Embedded Multicore Systems.",In *Workshop on Embedded Systems Security - WESS 2007*, pp. 17-22, 2007

[26] J. Fan, K. Sakiyama and I. Verbauwhede, "Montgomery modular multiplication Algorithm for Multi core Systems", In *proceedings of the IEEE Workshop on Signal Processing Systems: Design and Implementation (SIPS 2007)*, IEEE, 2007.

[27] Ç. K. Koç, T. Acar and B. S. Kaliski. Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro*, volume 16, pages 26-33, 1996

[28] J. L. Hennessy and D. A. Patterson. Computer Architecture: A quantitative Approach 3rd Ed. Mogarn Kaufman, 2003

[29] Introduction to Parallel Computing. Available at

https://computing.llnl.gov/tutorials/parallel_comp

[30] G. Blake, Seroussi and N. Smart. Elliptic curves in Cryptography, Cambridge University Press,
1997

[31] Venkatachalam. Power Reduction Techniques For Microprocessor Systems. *ACM Computing Surveys*, September 2005

[32] M. Powell, S-H. Yang, B. Falsafi, K. Roy, and T. N. Vijaykumar. Reducing leakage in a high-performance deep-submicron instruction cache. *IEEE Trans. VLSI Syst.* volume *9*, issue 1, pages 77–90. 2001.

[33] Ç. K. Koç and T. Acar. Montgomery Multiplication in GF(2^k), *Designs, Codes and Cryptography***,** Volume 14 , Issue 1, April 1998.

[34] ARM11MP. Available at http://www.arm.com

[35] S. Hisayoshi, S. Daniel, and T. Tsuyoshi. Exact Analysis of Montgomery Multiplication. *Progress in Cryptology - INDOCRYPT 2004*. 2004.

[36] J. Dhem, F. koeune, P. Leroux, P. Mestre, J. Quisquater and J. Willems. A practical Implementation of the timing attack. *CARDIS 1998, LNCS 1820*, pages 167-182, 2002.

[37] M. Stefan, P. Thomas and G. M. Berndt. Side-Channel Leakage of Masked CMOS Gates. *Topics in Cryptology – CT-RSA 2005*, 2005.

[38] M. Rosing. Implementing Elliptic curve cryptography. Manning, 1999.

[39] FIPS 186-2, Digital signature Standard. *Federal Information Processing Standards Publication 186-2*, available at http://csrc.nist.gov/, 2000.

[40] P. Mishra, N. Dutt and A. Nicolau. Technical Report #01-06: A Study of Out-of-Order Completion for the MIPS R10K Superscalar Processor. *University of Irvine*, 2001.

[41] Ç. K. Koç. Timing Attacks on Implementations of Diffi-Hellman, RSS, DSS and Other systems. *Advances in Cryptology –CRYPTO '96*, Lecture Notes Computer Science, Springer-Verlag, 1996.

[42] J. A. Muir. Techniques of Side Channel Cryptanalysis, Master of Math Thesis. *University of Waterloo*, 2001.

[43] IEEE P1363/ Draft 13, IEEE Standard Specification for Public-Key Cryptography, Annex A, *IEEE*, 1999.

[44] J. A. Ambrose, R. G. Ragel, and S. Parameswaran. RIJID: Random Code Injection to Mask Power Analysis based Side Channel Attacks. *Design Automation Conference, DAC '07. 44th ACM/IEEE,* 2007.

[45] D. Page. Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel. Technical Report CSTR-02-003, Department of Computer Science, University of Bristol, June 2002.

[46] T. F. Al-Somani, M. K. Ibrahim and A. Gutub. Highly Efficient Elliptic Curve Crypto-processor with Parallel GF(2^m) Field Multipliers. *Journal of Computer Science 2*(*5*), pages 395-400, 2006.

[47] O. Aciçmez. Advances in Side channel Cryptosystem Cryptanalysis: Micro Architectural Attacks. Ph.D Thesis of EECS, Oregon State University, 2006

[48] C. Percival. Cache missing for fun and profit. *BSDCan 2005*, Ottawa, 2005.

[49] O. Goldreich, R. Ostrovsky, Software protection and simulation on oblivious RAMs, *Journal of the ACM, vol. 43 no. 3*, pages 431–473, 1996

[50] D. A. Osvik, A. Shamir and E. Tromer. Cache Attacks and Countermeasures: The Case of AES. *Topics in Cryptology – CT-RSA 2006*, Lecture Notes Computer Science, Springer-Verlag, 2006.

[51] RSA Challenge Numbers. available at http:// http://web.archive.org/web/20061209135708/http://www.rsasecurity.com/rsalabs/node.asp?id=2093

[52] I. H. Jebril, R. Salleh, and Al-Shawakbeh. Efficient Algorithm in Projective Coordinates for ECC over *GF*(2^n). *International Journal of The computer, the Internet and Management Vol. 15#1*, pages 43-50, 2007.

[53] MIPS R10000 Microprocessor User's Manual, V.2. Available at http://techpubs.sgi.com/library/manuals/2000/007-2490-001/pdf/007-2490-001.pdf

[54] J. A. Ambrose, R. G. Ragel, and Sri Parameswaran. RIJID: Random Code Injection to Mask Power Analysis Based Side Channel Attacks. *DAC 2007*, 2007.