Elliptic Curve Cryptography: Algorithms and Implementation Analysis over Coordinate Systems

[1]Iskandar Setiadi, [2]Achmad Imam Kistijantoro
School of Electrical Engineering and Informatics
Institut Teknologi Bandung
Bandung, Indonesia
[1]iskandarsetiadi@students.itb.ac.id,
[2]imam@informatika.org

Atsuko Miyaji
School of Informatics Science
Japan Advanced Institute of Science and Technology
Nomi, Japan
miyaji@jaist.ac.jp

Abstract— **Since the last decade, the growth of computing power and parallel computing has resulted in significant needs of efficient cryptosystem. Elliptic Curve Cryptography (ECC) offers faster computation and stronger security over other asymmetric cryptosystems such as RSA. ECC can be used for several cryptography activities: secret key sharing, message encryption, and digital signature. This paper gives step-by-step tutorial to transform ECC over prime field *GF(p)* from mathematical concept to the software implementation. This paper also gives several alternatives and tradeoffs between different coordinate systems in the computational process. The implementation result is quite interesting since several computational costs have been optimized in latest instruction sets. For the study case, we provides the implementation result in C language with GNU GMP library on Intel i3 CPU M350 2.27GHz (1 Core, 2 GB RAM, 32-bit architecture).**

Cryptography, Elliptic Curve, Coordinate System, ECC Algorithm

I. Introduction

Elliptic curve cryptography is a class of public-key cryptosystem which was proposed by N. Koblitz [1] and V. Miller [2]. ECC protocols assume that finding the elliptic curve discrete algorithm is infeasible. ECC provides strong security as RSA with smaller bits key, which implies faster performance and lower computational complexity. A 160-bit key in ECC has the same security level as 1024-bit key in RSA [3].

There are several parameters and algorithm choices which should be considered before implementing ECC system. Several curve domain parameters (field representation, curve type), algorithm for field arithmetic, elliptic curve arithmetic, and protocol arithmetic can be influenced by security factors, platform, constraints, and communications environment [4].

Algorithms and coordinate systems, which are given in this paper, are used to emphasize the benefit of elliptic curve in cryptosystems and give an insight for technical people to implement a simple elliptic curve cryptosystem. This paper also offers several performance comparisons to show the tradeoffs between coordinate systems. It should be noted that there are several other researches which have tried to optimize ECC algorithms in several different ways [4] [5] [6] [7] [8].

II. Basic Concepts

A. Finite Field

Finite field, which is also known as Galois field, is an algebraic system defined on a set F that contains a finite number of elements. Finite field needs to have well-defined binary operations ($+$, x) satisfying the abelian group. It should be noted that finite field only exists when the order is a prime power p^k (p: prime number, k: positive integer). The formal proof of finite field is omitted [9].

B. Elliptic Curves

The general *Weierstrass* equation defines a cubic curve over a field as the following:

$$E: y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \tag{1}$$

where $a_1, a_2, a_3, a_4, a_6 \in F$ and the discriminant of E is not equal zero ($\Delta \neq 0$). Alongside, there is a specified point at infinity which is denoted as O. From the general *Weierstrass* equation, any elliptic curve E in its standard form can be written as:

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

where the value of a, b are predefined and $\Delta = -16\,(4a^3 + 27b^2) \neq 0$. Figure 1 below represents $y^2 = x^3 - x + 1$ over real ($\mathbb{R}$) field:

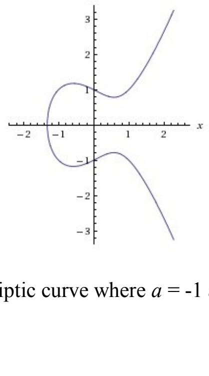

Figure 1. Elliptic curve where $a = -1$ and $b = 1$

Since it satisfies the abelian group, we need to define binary operations over elliptic curve. All operations in abelian group are commutative. These operations are defined as:

Let P and Q be two points in the curve.

- $P + Q$ where $P \neq Q \neq O$ will be resulted in a new point R ($P + Q = R$).
- If $P + Q$ doesn't intersect the curve, we can say that $P + Q$ is equal to infinity ($P + Q = O$). This case happens when $P = -Q \rightarrow P(x, y), Q(x, -y)$
- If $P = O$ or $Q = O$, this operation will be resulted in the other point. For example, if $P = O$, then $P + Q = O + Q = Q$ (vice versa).
- $P + Q$ where $P = Q \neq O$ can be denoted as $2P$. If the y-coordinate is equal to 0, $2P$ is equal to infinity ($2P = O$).
- If $P = Q = O$, we can infer that $P + Q = O + O = O$.

C. Elliptic Curves over GF(p)

For cryptographic purposes, we want to use integer points instead of real points across the curve. Let *GF(p)* be the finite field with p elements and E be an elliptic curve. To find all the points in the finite field *GF(p)*, we only need to consider $x = 0,1,\ldots,p-1$ and take the square root to find the value of y.

Since elliptic curve is symmetric over $y = 0$, it is guaranteed that every valid x-coordinate in the curve can represent y-coordinates in two different points: $a \pmod p$ and $[p-a] \pmod p$, where a is the square root value in modulo p.

The number of points (order) on elliptic curve over finite field can be computed using *Schoof*'s algorithm [10]. *Hasse*'s theorem [11] gives us an estimation of the number of points N by $|N - (p + 1)| \leq 2.\sqrt{p}$. Fortunately, there are several standard curves which can be used for implementing elliptic curve cryptosystems such as NIST [12], so that number of points on elliptic curve is already known.

D. Single Coordinate Systems

In the early implementation, an elliptic curve can be represented by several coordinate systems. The simplest one is affine coordinates. In order to improve the performance of elliptic curve computation, there are several coordinate systems such as projective coordinates, Jacobian coordinates, Chudnovsky Jacobian coordinates, and Modified Jacobian coordinates. The representation of each coordinate system can be written as below:

- **Affine** coordinates:

$$P(x, y)$$

- **Projective** coordinates:

$$P(X, Y, Z)$$

where $x = X / Z$ and $y = Y / Z$

- **Jacobian** coordinates:

$$P(X, Y, Z)$$

where $x = X / Z^2$ and $y = Y / Z^3$

- **Modified Jacobian** coordinates:

$$P(X, Y, Z, aZ^4)$$

where a is the value from $E : y^2 = x^3 + ax + b$, $x = X / Z^2$ and $y = Y / Z^3$

E. Mixed Coordinate Systems

It is possible to mix different coordinates for the computation in point arithmetic level. The mixing between Jacobian and affine ($\Im + Af$) for addition operation only uses ($8M + 3S$) computational time, compared to $\Im + \Im$ ($12M + 4S$) and $Af + Af$ ($2M + S + I$) [M: Multiplication, S: Squaring, I: Inversion]. As the number of bits gets longer, the computational cost of I (inversion) gets more expensive. Formula for Jacobian + affine operation can be written as:

Let $P + Q = R$ where $P, Q \neq O$ defined as:

$P(X_1, Y_1, Z_1)$ in Jacobian coordinate

$Q(X_2, Y_2)$ in affine coordinate

$R(X_3, Y_3, Z_3)$ in Jacobian coordinate

Let define A, B, C, D as the following:

$A = X_2.Z_1^2$, $B = Y_2.Z_1^3$, $C = A - X_1$, $D = B - Y_1$

From these variables, we can derive:

$$X_3 = D^2 - (C^3 + 2\, X_1.C^2)$$

$$Y_3 = D.\,(X_1.C^2 - X_3) - Y_1.\,C^3$$

$$Z_3 = Z_1.\,C \qquad (4)$$

In this paper, three coordinate systems will be used: affine coordinates (Af), Jacobian coordinates ($\Im$), and mixed coordinates between Jacobian and affine ($\Im + Af$). Affine and Jacobian coordinates are chosen because of their simplicity in representing ECC computations. We can also utilize those coordinate systems to show the differences between single and mixed coordinate systems.

III. Elliptic Curve Cryptosystems

Elliptic curve has been adapted for several cryptographic schemes, such as:

-	Key agreement scheme	: ECDH, ECDHE
-	Encryption scheme	: ECIES, PSEC
-	Digital signature scheme	: ECDSA

For better understanding, this part will elaborate the implementation of ECDH and ECIES.

A. Key Agreement Scheme

ECDH (Elliptic Curve Diffie-Hellman) is a scheme where two or more parties can agree on a key over insecure channel. The agreed key is usually used to derive another key which can be used for symmetric key cipher (i.e. AES). On the other occasion, we can use the shared key directly, but it still contains weak bits due to the Diffie-Hellman exchange; the solution is that we can hash the shared key in order to remove these weak bits. It should be noted that Diffie-Hellman scheme doesn't provide authentication; authenticity assurance must be obtained by other mechanism.

For this key agreement key, each party must agree upon the domain parameters (prime case): p as the finite field size, a, b as the curve parameters, G as the point generator, n as the order of G, and h as the cofactor.

Let P_1 be the private key of A, P_2 be the private key of B, Q_1 be the public key of A, and Q_2 be the public key of B. P_1, P_2 are random numbers in interval [1, n-1] where $Q_1 = [P_1].G$ and $Q_2 = [P_2].G$.

Algorithm 1. ECDH

```
INPUT:
Q where Q is the public key of other party
P_k where P_k is the private key of receiver
OUTPUT:
x where x is the shared key between two parties
STEP:
1. $R \leftarrow [P_k].Q$
2. Return($R_x$)
```

The first party will calculate $R = [P_1].([P_2].G)$ while the second party will calculate $R = [P_2].([P_1].G)$. Since the multiplication operation is commutative, both parties will calculate the same value of R. The shared secret is R_x (x is coordinate of the point). This shared secret, as stated above, can be used for symmetric key cipher or other cryptographic schemes.

B. Encryption Scheme

On the early age of elliptic curve cryptosystems, there are several choices of scheme integration such as Massey-Omura and ElGamal cryptosystems ([13]). The main problem is that there is no convenient method known in transforming plaintext to points on E.

ECIES (Elliptic Curve Integrated Encryption Scheme) incorporates a symmetric-key encryption and message authentication scheme. This section will provide a simplification of ECIES (see ([14]) for the original idea).

For this scheme, we will use same variable notations as in Section 3A.

Algorithm 2. Simplified ECIES (Encryption)

```
INPUT:
Q where Q is the public key of other party
m where m is the intended message
OUTPUT:
e where e is the encrypted message
C (x,y) where C is the chosen point
STEP:
1. Choose random value k [1,n-1]
2. $C \leftarrow [k].G$
3. $R \leftarrow [k].Q$
4. $e \leftarrow (R_x$*DECIMAL($m$)) modulo $p$
5. Return ($e$,$C$)
```

DECIMAL() function above is used to transform plaintext to the decimal value. For example, we can use ASCII convention to convert 'A' → 65, etc. It can be inferred from **Step 3** that $R = [k].Q = [k].([P_k].G)$. The cipher text consists of two components: encrypted message e and chosen point C.

Algorithm 3 below shows decryption process of the cipher text.

Algorithm 3. Simplified ECIES (Decryption)

```
INPUT:
e where e is the encrypted message
C where C is the chosen point
P_k where P_k is the private key of receiver
OUTPUT:
d where d is the decoded message
STEP:
1. $R \leftarrow [P_k].C$
2. $d \leftarrow (e*(R_x)^{-1})$ modulo $p$
3. Return(TEXT($d$))
```

TEXT() function above is used to transform decimal value back to the original message. This protocol assumes both parties use the same message encoding agreement.

Step 1 above computes $R = [P_k].C = [P_k].([k].G)$. The value of R from **Algorithm 3** is equal to the value of R from **Algorithm 2**. This scheme is similar to the key agreement scheme as in Section 3A. Finally, **Step 2** computes modular multiplication inverse operation to decrypt the intended message.

IV. Implementation

On the scalar arithmetic level, a lot of work has been published which focuses on designing the most efficient scalar multiplication algorithm. A faster algorithm usually requires bigger memory size while a full-fledged secure algorithm usually runs slower. For example, elliptic curve implementation in smart card must be robust against side-channel attacks; it also requires small memory usage in the computational process. On the other hand, elliptic curve implementation in application server is able to use bigger memory size to deliver faster performance. In this section, we will present several algorithms for solving each case above.

A simple, yet efficient algorithm is binary scalar multiplication algorithm. The basic idea is similar to base conversion, where multiplication is performed in each shift while addition is performed in non-zero value bit. There are two variations of this algorithm: left-to-right binary algorithm and right-to-left binary algorithm. See **Algorithm 4** and **Algorithm 5** for better understanding.

Algorithm 4. Left to Right Binary

```
INPUT:
P where P is a valid point in the curve ($P \in E(F_q)$)
k where k is represented in base-2 ($(k_{n-1},\ldots, k_1, k_0)_2$)
OUTPUT:
R where R is a valid point in the curve ($R = [k].P$)
STEP:
1. Initialize R with P ($R \leftarrow P$)
2. for i = n-2 downto 0 do
3.     $R \leftarrow 2R$
4.     if ($k_i = 1$) then $R \leftarrow R + P$
5. end for
6. Return(R)
```

Algorithm 5. Right to Left Binary

```
INPUT:
P where P is a valid point in the curve ($P \in E(F_q)$)
k where k is represented in base-2 ($(k_{n-1},\ldots, k_1, k_0)_2$)
OUTPUT:
R where R is a valid point in the curve ($R = [k].P$)
STEP:
1. Initialize R with O ($R \leftarrow O$)
   // O is point at Infinity
2. Initialize S with P ($S \leftarrow P$)
3. for i = 0 to n-1 do
4.     if ($k_i = 1$) then $R \leftarrow R + S$
5.     $S \leftarrow 2S$
6. end for
7. Return(R)
```

The disadvantage of these algorithms above is that they are insecure to side-channel attack. Attackers can analyze the scalar bit in each loop iterations [15]. These algorithms are not suitable for the implementation in smart card or other embedded devices. To prevent side-channel attack, Montgomery ladder algorithm provides efficient and strong security guarantee for performing scalar multiplication in embedded devices [16].

Algorithm 6. Montgomery Ladder

```
INPUT:
P where P is a valid point in the curve ($P \in E(F_q)$)
k where k is represented in base-2 ($(k_{n-1},\ldots, k_1, k_0)_2$)
OUTPUT:
R where R is a valid point in the curve ($R = [k].P$)
STEP:
1.  Initialize R with O ($R \leftarrow O$)
    // O is point at Infinity
2.  Initialize S with P ($S \leftarrow P$)
3.  for i = n-1 downto 0 do
4.     if ($k_i = 1$) then
5.           $R \leftarrow R + S$
6.           $S \leftarrow 2S$
7.     else
8.           $S \leftarrow R + S$
9.           $R \leftarrow 2R$
10.    end if
11. end for
12. Return(R)
```

Montgomery ladder performs addition and doubling operations in each loop iteration. For random value of *k*, it is expected that Montgomery ladder will give 30-40% slower performance. The main reason is that binary scalar multiplication algorithm involves *n* point doublings and *n*/2 point additions, while **Algorithm 6** involves the same number of point doublings and point additions.

If the memory is less constrained, we may use window techniques in our algorithm. The idea of this technique is pre-computation. For example, $k = 30 = (11110)_2$ requires 5 point doublings and 4 point additions in binary scalar multiplication algorithm. By pre-computing 5*P*, we can perform $((5P * 2) + 5P) * 2$ which requires (3+2) point doublings and (2+1) point additions. This technique is very efficient for computing large number of *k*.

In implementing the window technique, NAF (Non-Adjacent Form) representation will be used. NAF representation assures the minimum value of Hamming weight. **Algorithm 7** below shows the method to compute the width-*w* NAF from a positive integer *k*.

Algorithm 7. Computing the width-*w* NAF

```
INPUT:
w where w is the window size
k where k is a positive integer in base-10
OUTPUT:
$d_{n-1},\ldots,d_1,d_0$ where $-2^{w-1} \le d_i < 2^{w-1}$ , $d_i$ is in base-10, and n is the length of k in base-2
STEP
1.  $i \leftarrow 0$
2.  while $k \ge 1$ do
3.     if (k modulo 2 = 1) then
4.           $k_i \leftarrow k$ modulo $2^w$
5.           if ($k_i > 2^{w-1} - 1$) then
6.                $k_i \leftarrow k_i - 2^w$
7.           end if
8.           $k \leftarrow k - k_i$
9.     else
10.          $k_i \leftarrow 0$
11.    end if
12.    $k \leftarrow k / 2$
13.    $i \leftarrow i + 1$
14. end while
15. Return($d_{n-1},\ldots, d_1, d_0$)
```

For example, the representation of $k = 1122334455$ for $w = 4$ is $1\ 00001\ 00070\ 00050\ 00700\ 07000\ \bar{1}0007$. ($\bar{1} = -1$)

Algorithm 8 below shows scalar multiplication operations with window-NAF technique.

Algorithm 8. Window – NAF (Left to Right)

```
INPUT:
P where P is a valid point in the curve ($P \in E(F_q)$)
w where w is the window size
$d_{n-1},\ldots,d_1,d_0$ where $-2^{w-1} \le d_i < 2^{w-1}$ , $d_i$ is in base-10, and n is the length of k in base-2
OUTPUT:
R where R is a valid point in the curve ($R = [k].P$)
STEP:
1.  for all d in {1, 3, …, $2^{w-1}$-1} do
2.      $A_d$ ← [d].P // pre-compute
3.  end for
4.  Initialize R with O (R ← O)
     // O is point at Infinity
5.  for i = n-1 downto 0 do
6.      d ← |$d_i$|
7.      R ← 2R
8.      if ($d_i$ > 0) then
9.              R ← R + $A_d$
10.     else if ($d_i$ < 0) then
11.             R ← R - $A_d$
12.     end if
13. end for
14. Return(R)
```

Typically, smart cards only provide memory in kilobits size, which is quite small. **Algorithm 8** is not quite feasible to be implemented in such kind of devices. For example, if we set the windows size to 6, the curve size to 384 bits, and if we use Jacobian coordinate, we need to pre-compute d = {1,3,5,…,31}, which requires (16 * 384 * 3) = 18,432 bits. However, this size constraint is considered small if we need to implement ECC in an application server.

V. Evaluation and Analysis

In this evaluation, RDTSC and RDTSCP method are implemented to measure operation cycles. In this paper, we assume that addition and subtraction require the same amount of computation cost. From our initial experimentation, multiplication and squaring have a little running time difference in GMP library, so we can assume that both operations are equal.

The comparison of several basic operations is presented in **Table I**. We use Intel i3 CPU M350 2.27GHz (1 Core, 2 GB RAM, 32-bit architecture) in executing all tests. In addition, we choose 256-bit length for all evaluations since 256-bit ECC are often used in practice (OpenSSL, etc).

It is shown that addition and shifting operation require nearly the same amount of time. We will use A, M, I to denote the computation complexity of addition, multiplication, and inversion respectively. Therefore, we can conclude that A = 0.514 M and I = 8.22 M. These results are architecture dependent, where different instruction sets may lead to different running time. Nevertheless, we can approximate the expected running time for each algorithm based on these statistics.

TABLE I. Performance comparison for basic operations

Basic Operation	Average # of Cycles	Running Time
Addition	957	0.422 µs
Shifting (2 * k)	941	0.415 µs
Multiplication (k * k_2)	1,861	0.821 µs
Inversion	15,300	6.750 µs

- Average of 10 different k, k_2 over 10,000 running times for each k, k_2 (k, k_2 is chosen randomly between [1, n-1])
- The average number of cycles in 1 second is 2,266,723,093

This section will compare the performance for scalar multiplication over different algorithms. Three coordinate systems will be considered: affine coordinate, Jacobian coordinate, and Jacobian-affine (mixed) coordinate. We will also show the differences between each scalar multiplication algorithm. The result will be measured in millisecond (ms).

For the implementation verification, we provide elliptic curve P-256 test vector ($Q = [k].G$) as below [17]:

k: `112233445566778899`

x (base 16):

```
339150844EC15234807FE862A86BE77977DBFB3
AE3D96F4C22795513AEAAB82F
```

y (base 16):

```
B1C14DDFDC8EC1B2583F51E85A5EB3A155840F2
034730E9B5ADA38B674336A21
```

In **Table II** below, we present the expected computational cost for addition and doubling operations in several coordinate systems.

TABLE II. Computational cost for operations over several coordinate systems

Coordinate System	Addition	Doubling
Af	$I + 3M + 6A$	$I + 4M + 7A$
$\Im$	$16M + 7A$	$10M + 7A$
$\Im + Af$	$11M + 7A$	-

- M comprises of multiplication and squaring
- A comprises of addition, shifting, and subtraction

For analysis purpose, left-to-right binary algorithm will be used to compare the computational costs between several coordinate systems. In this experiment, we use 256-bit curve type, so left-to-right binary algorithm requires 128 point doublings and 256 point additions. Theoretically, we can use Jacobian coordinate over affine coordinate if it satisfies $I \geq 8.33\ M + 0.33\ A$. It's better to use Jacobian-affine coordinate over affine coordinate if it satisfies $I \geq 6.66\ M + 0.33\ A$. These results show that Jacobian coordinate is expected to be slower than affine coordinate in this implementation.

In order to reconfirm these expected computational costs (**Table II**), we also check the running time for each operation (**Table III**).

TABLE III. COMPARISON BETWEEN EXPECTED AND ACTUAL RUNNING TIME FOR ADDITION AND DOUBLING OPERATIONS

Coordinate System	Operation	Expected Running Time	Actual Running Time
Af	Addition	11.745 µs	11.610 µs
Af	Doubling	12.988 µs	13.179 µs
$\Im$	Addition	16.090 µs	15.792 µs
$\Im$	Doubling	11.164 µs	11.242 µs
$\Im$ + *Af*	Addition	11.985 µs	11.854 µs

- Expected running time is calculated from Table I and Table II
- Actual running time: average of 10 different k over 100 running times for each k ($Q = [k].G$, k is chosen randomly between $[1, n\text{-}1]$)

From **Table III**, it is expected that left-to-right binary algorithm (Alg. 4) in affine, Jacobian, Jacobian-affine coordinates will run in 4.860 ms, 4.9 ms, 4.412 ms respectively. The actual running time is shown below (**Table IV**).

TABLE IV. PERFORMANCE COMPARISON FOR OPERATIONS OVER SEVERAL COORDINATE SYSTEMS AND SCALAR ALGORITHMS

Coordinate System	Scalar Algorithm	Memory Constrained	Running Time
Af	Alg. 4	No	5.037 ms
Af	Alg. 5	No	4.925 ms
$\Im$	Alg. 4	No	5.124 ms
$\Im$	Alg. 6	No	7.762 ms
$\Im$ + *Af*	Alg. 4	No	4.771 ms
$\Im$ + *Af*	Alg. $8^{(1)}$	Yes	3.501 ms
$\Im$ + *Af*	Alg. $8^{(2)}$	Yes	3.709 ms

- Curve type P-256 NIST
- Average of 10 different k over 100 running times for each k ($Q = [k].G$, k is chosen randomly between $[1, n\text{-}1]$)
- (1) Window size = 4; (2) Window size = 5

The result above (**Table IV**) can be concluded in the following remarks:

- For P-256 NIST curve type in less memory constrained platforms, window-NAF Jacobian–affine coordinate with $w = 4$ is 46.37% faster and 43.87% faster than the left-to-right binary algorithm with Jacobian coordinate and affine coordinate respectively (including the final inversion for Jacobian coordinate).
- For P-256 NIST curve type, window-NAF with $w = 4$ is 5.94% faster than window-NAF with $w = 5$. An optimum window size varies depending on curve types and environments which are used to conduct the computation.
- Left-to-right algorithm in Jacobian coordinate is 51.48% faster than Montgomery ladder in the same coordinate system. It should be noted that Montgomery ladder gives better security guarantee for embedded systems. However, if there's a security guarantee against side-channel attack, left-to-right algorithm in Jacobian – affine coordinate is 5.58% faster, 3.23% faster, and 7.4% faster than the left-to-right algorithm in affine coordinate, right-to-left algorithm in affine coordinate, and left-to-right algorithm in Jacobian coordinate respectively.
- Addition operation and doubling operation may result in different performances on other architectures. In this implementation, Jacobian coordinate doesn't offer better performance than affine coordinate system. On several other researches, it is shown that Jacobian coordinate offers better performance in 8-bit processor. This result is quite interesting for future explorations.
- Finally, different mixed coordinate systems and curve types may yield to better performances. In other reference [2], there are several mixed coordinate systems which are theoretically faster than window–NAF in Jacobian–affine coordinate.

REFERENCES

[1] N. Koblitz, "Elliptic curve cryptosystems," Mathematics of computation 48.177, 1987, pp. 203-209.

[2] V. S. Miller, "Use of elliptic curves in cryptograph,." Advances in Cryptology—CRYPTO'85 Proceedings.

[3] G. V. S. Raju and Rehan Akbani, "Elliptic curve cryptosystem and its applications," IEEE Systems, Man and Cybernetics, vol. 2, 2003.

[4] M. Brown, et al., Software implementation of the NIST elliptic curves over prime fields. Springer Berlin Heidelberg, 2001.

[5] H. Cohen, A. Miyaji, and T. Ono, "Efficient elliptic curve exponentiation using mixed coordinates," Advances in Cryptology—ASIACRYPT'98. Springer Berlin Heidelberg, 1998.

[6] A. Miyaji, T. Ono, and H. Cohen, "Efficient elliptic curve exponentiation," Information and Communications Security, 1997, pp. 282-290.

[7] M. Rivain, "Fast and regular algorithms for scalar multiplication over elliptic curves," IACR Cryptology ePrint Archive, 2011, pp. 338.

[8] P. Longa and C. Gebotys, "Efficient techniques for high-speed elliptic curve cryptography," Cryptographic hardware and embedded systems, CHES, 2010, pp. 80-94.

[9] N. Jacobson, Basic algebra I. Courier Dover Publications, 2012.

[10] R. Schoof, and Par Ren E. Schoof, "Counting points on elliptic curves over finite fields," 1995.

[11] L. C. Washington, Elliptic curves: number theory and cryptography. CRC press, 2012.

[12] NIST, Recommended elliptic curves for federal government use, available at http://csrc.nist.gov/encryption, 1999.

[13] V. G. Martinez, L. Hernández Encinas, and A. Martín Muñoz, "A comparative analysis of hybrid encryption schemes based on elliptic curves," Open Mathematics Journal 6, 2013, pp. 1-8.

[14] D. G. Stinson, Cryptography: theory and practice. CRC press, 2005.

[15] Paul Kocher, Joshua Jaffe, and Benjamin Jun, "Differential power analysis," Advances in Cryptology—CRYPTO'99. Springer Berlin Heidelberg, 1999.

[16] Peter L. Montgomery, "Speeding the pollard and elliptic curve methods of factorization," Mathematics of computation 48.177, 1987, pp. 243-264.

[17] Point at Infinity. Test Vectors for the NIST curves, available at http://point-at-infinity.org/ecc/nisttv, 2005.