이학석사 학위논문

Efficient Inversion Algorithms and Its Applications to ECDLP

(효과적인 역원 알고리즘과 타원 곡선 이산 대수 문제에의 응용)

2013년 8월

서울대학교 대학원

수리과학부

정희원

Efficient Inversion Algorithms and Its Applications to ECDLP

(효과적인 역원 알고리즘과 타원 곡선 이산 대수 문제에의 응용)

지도교수 천 정 희

이 논문을 이학석사 학위논문으로 제출함

2013년 4월

서울대학교 대학원

수리과학부

정희원

정희원의 이학석사 학위논문을 인준함

2013년 6월

위 원 장 _____ (인)

부 위 원 장 _____ (인)

위 원 _____ (인)

Efficient Inversion Algorithms and Its Applications to ECDLP

A dissertation
submitted in partial fulfillment
of the requirements for the degree of
Master of Science
to the faculty of the Graduate School of
Seoul National University

by

Heewon Chung

Dissertation Director : Professor Jung Hee Cheon

Department of Mathematical Sciences
Seoul National University

August 2013

Abstract

In this paper, we apply tag tracing technique to elliptic curves. Tag tracing is the technique which can apply to the Pollard rho algorithm, first suggested by Cheon et al. They can solve DLP 10 times faster than before in the multiplicative groups of finite field. However, for elliptic curves, tag tracing does not effect so much because of an addition in elliptic curves. To add two points in elliptic curves, we actually need one inversion and three multiplications including two squaring. An inversion operation is the most time-consuming arithmetic, so an inversion algorithm needs to be improved. We investigate some useful inversion algorithms which can be used for a finite extension field. Until now, there are two widely used algorithms to get an inverse element over the finite extension field. One is extended Euclidean algorithm, and the other is Itoh-Tsujii inversion algorithm. Beside these algorithms, we can compute an inverse element by computing an inverse matrix. Generally, this method is slower than the other algorithms, but when the degree of the extension field is small, this is slightly faster. We compare the complexity of these algorithms and use the most efficient method suitably for our setting when applying tag tracing to elliptic curves, Also, to accelerate the Pollard rho algorithm, we calculate the pre-computation table. Using these skills, we expect to reduce the time complexity to solve ECDLP than before.

Key words: Inversion Algorithm, Extended Euclidean Algorithm, Tag Tracing, ECDLP
Student Number: 2010-23075

Contents

CONTENTS

Chapter 1

Introduction

Elliptic curve cryptography (ECC), first introduced by Koblitz [12] and Miller [15], is a public-key cryptography and have advantages that ECC provide security equivalent to classical systems while using fewer bits. Many cryptosystems are based on elliptic curves, especially on the DLP, such as digital signatures and Diffie-Hellman key exchange [5]. Thus, a lot of people have interested in solving DLP with less time-consuming. Shank proposed a method [23], called Baby Step, Giant Step. It requires approximately $\sqrt{p}$ steps and around $\sqrt{p}$ storage, where p is the order of a group. However, a disadvantage of the Baby Step, Giant Step method is that it requires a lot of memory. To resolve this problem, Pollard introduced the Pollard rho algorithm [17], based on the birthday paradox. The Pollard rho algorithm runs in approximately the same time as Baby Step, Giant Step, but it rarely use memory, so the Pollard rho method is more preferred to Baby Step, Giant Step.

Recently, Cheon et al. [4] propose tag tracing for the multiplicative groups of finite field. They also use the Pollard rho algorithm, generating a random walk with adding iteration function and detecting a collision with a distinguished point. The main idea of tag tracing is we do not need to compute full computation because the probability of meeting a distinguished point is very low. The full computation requires when the end of iteration function or finding the distinguished point. Hence, they can reduce a lot of time to

solve DLP.

However, applying tag tracing to elliptic curves does not have an effect so much because of arithmetic in elliptic curves. In elliptic curves, one addition requires one inversion and three multiplications including two squaring. Among the finite field arithmetic, an inversion is one of the expensive computation cost, and a multiplication is relatively cheaper than an inversion. This is why tag tracing does not have an effect in elliptic curve, so we need to improve the inversion algorithm. There are widely used inversion algorithms for the finite extension field. One is extended Euclidean algorithm, and the other is Itoh-Tsujii inversion algorithm. Also, there are three variants of extended Euclidean algorithm [14], called IE, IP and IM. Algorithm IE is the classical extended Euclidean algorithm, but this algorithm requires so much the subfield inversion. Algorithm IP is the parallel version of IE, so the number of the subfield inversion required is less than IE. Algorithm IM requires the only one subfiled inversion, but it requires more multiplication more than IE and IP. Itoh-Tsujii inversion algorithm also requires the only one subfield inversion, but it is related to the minimal length of addition chain, which is the hard problem. According to [14], algorithm IM is the most efficient in software implementation of finite extension field, however, in a hardware implementation, Itoh-Tsujii inversion algorithm is the best performance [13].

As you seen above, every inversion algorithm needs the subfield inversion operation. Modifying the extended Euclidean algorithm or Itoh-Tsujii algorithm is not easy, so we decide that we calculate the subfield inversion before running the algorithm to make more efficient inversion algorithm. Then, we can obtain the inverse element with taking less time, and an effect of applying tag tracing is bigger than before.

Organization

In Section 2, we briefly introduce some definitions related to an finite extension field and review the Pollard rho algorithm in elliptic curves and tag tracing. Section 3 provides well-known inversion algorithms for an finite extension field and complexity of these algorithms. We suggest the technique of tag tracing fit for elliptic curves in Section 4.

Chapter 2

Preliminaries

In this chapter, we review some definitions and algorithms which we use in the rest of paper, especially arithmetic of elliptic curves and Pollard rho algorithm. We are interested in solving ECDLP over the finite extension field, and we only consider the finite extension field with small extension degree and middle size prime.

2.1 Arithmetic of Elliptic Curves

Let $\mathbb{F}_{q^n}$ be the finite extension field and let E be an elliptic curves defined over $\mathbb{F}_{q^n}$. Since there are only finitely many pairs (x, y) with $x, y \in \mathbb{F}_{q^n}$, the group $E(\mathbb{F}_{q^n})$ is finite. In this section, we present arithmetic of elliptic curves over finite extension fields.

We now examine the group law in more details. Let E be an elliptic curve defined by $y^2 = x^3 + Ax + B$ and let ∞ be point at infinity. Suppose $P_1 = (x_1, y_1)$, and $P_2 = (x_2, y_2)$ are points on E with $P_1, P_2 \neq \infty$. Define $P_1 + P_2 = P_3 = (x_3, y_3)$ as follows.

1. If $x_1 \neq x_2$, then $x_3 = s^2 - x_1 - x_2$ and $y_3 = s(x_1 - x_3) - y_1$ with $s = \frac{y_2 - y_1}{x_2 - x_1}$.
2. If $x_1 = x_2, y_1 \neq y_2$, then $P_1 + P_2 = \infty$.

3. If $P_1 = P_2$ and $y_1 \neq 0$, then $x_3 = s^2 - 2x_1$ and $y_3 = s(x_1 - x_3) - y_1$ with $s = \frac{3x_1^2 + A}{2y_1}$.

4. If $P_1 = P_2$ and $y_1 = 0$, then $P_1 + P_2 = \infty$.

5. $P + \infty = \infty + P = P$ for all points P on E.

Now, we give how to compute a scalar multiplication of some point on E. The straightforward way of computing scalar multiplication is through repeated addition. However, this is a fully exponential approach, so we use another way to computing the multiplication. There is more efficient way using only doubling arithmetic, called Double-and-Add. The algorithm works as follow.

Algorithm 1 Double-and-Add

Input : $P \in E$

Output : $Q = kP$

1: Bit representation of k, say $k = k_0 + 2k_1 + 2^2k_2 + \cdots + 2^mk_m$.
2: $Q \leftarrow 0$.
3: **for** $i = m$ to 0 **do**
4: $\quad Q \leftarrow 2Q$
5: $\quad$ **if** $k_i = 1$ **then**
6: $\quad\quad Q \leftarrow Q + P$
7: $\quad$ **end if**
8: **end for**
9: **return** Q

2.2 Pollard Rho Algorithm

In this section, we recall the Pollard rho method for elliptic curve discrete logarithm problem (ECDLP). Pollard rho algorithm is introduced in [17] for solving discrete logarithm problem (DLP) and is based on the birthday paradox. This algorithm can compute DLP in arbitrary finite abelian groups, such as in groups of points of elliptic curves over finite fields. Before the Pollard

rho algorithm was introduced, Shank's method [23], also called baby-step giant-step, was the most efficient algorithm to solve DLP. Shank's method allows one to compute DL in a cyclic group G of order p in deterministic time $O(\sqrt{p})$ and space for $\sqrt{p}$ group elements. Pollard rho method also has time complexity $O(\sqrt{p})$, but it rarely uses the memory, so using Pollard rho algorithm is preferred to Shank's method.

Let G be any finite group of order p. Choose a function $f : G \to G$ that behaves rather randomly and $\{g_i\}_{i\in\mathcal{I}}$ is a sequence in G defined by $g_{i+1} = f(g_i)$ with random $g_0 \in G$. Since $\{g_i\}_{i\in\mathcal{I}}$ is a periodic sequence, there exist k and ℓ such that $g_0, \ldots, g_{k+\ell-1}$ are all distincts and $g_j = g_{j+k}$ for all $j \geq \ell$. ℓ is called the preperiod and k is called the period of the sequence $\{g_i\}_{i\in\mathcal{I}}$.

From now on, we show how the Pollard rho method for elliptic curve discrete logarithm works. Let E be an elliptic curve over finite field, P a point of prime order p on E, and G the subgroup of E generated by P. The Pollard rho method divides the group G into three subsets of roughly the same size, say G_1, G_2, and G_3. Define $f : G \to G$ by

$$g_{i+1} := f(g_i) = \begin{cases} g_i + P & \text{if } g_i \in G_1 \\ 2g_i & \text{if } g_i \in G_2 \\ g_i + Q & \text{if } g_i \in G_3 \end{cases}$$

where $Q \in G$ is the target element. For any point R of the elliptic curve E, we can express into $R = aP + bQ$, with $a, b \in [0, p-1]$. Let $g_i = a_iP + b_iQ$ and $a_i, b_i \in [0, p-1]$ for all i. By above, there are j and ℓ such that $g_j = g_\ell$, which implies $a_jP + b_jQ = a_\ell P + b_\ell Q$. Since $Q = kP$ for unknown k, $a_j + b_jk \equiv a_\ell + b_\ell k \mod p$. Hence, we can get $k = (a_j - a_i)(b_i - b_j)^{-1} \mod p$ if $b_i - b_j$ and p are relatively prime.

Using arbitrary multipliers, we can make the better iteration function. As before, partition the group G into r subsets of roughly the same size, say $G_1, G_2, \ldots, G_r$. For $j = 1, 2, \ldots, r$, we generate $2r$ random numbers $a_j, b_j \in [0, p-1]$. Let $M_j = m_jP + n_jQ$ and set r multipliers M_j for all j. Define an index function $\iota : G \to \{1, 2, \ldots, r\}$ and the r-adding iteration function $f_r : G \to G$ by $f_r(g_i) = g_i + M_{\iota(g_i)} =: g_{i+1}$ and start from $g_0 = a_0P + b_0Q$. Since g_i has the form $a_iP + b_iQ$, $a_{i+1} \equiv a_i + m_j \mod p$ and $b_{i+1} \equiv b_i + n_j$

mod p. Similarly, the sequence $\{g_i\}$ is periodic, there exists a collision, and then we can solve DLP.

This iteration function is first introduced in [20] and according to [19], when $r \geq 8$, a collision is expected after $O(\sqrt{p})$ iterations. Also, in [24], r-adding iteration function with $r \geq 20$ perform very close to a random function.

As mentioned above, the Pollard rho algorithm can solve DLP with $O(\sqrt{p})$ time complexity and negligible memory complexity. This algorithm uses memory only finding the collision, so we need an efficient method to find a collision. There are many methods to detect a collision suggested by Floyd [10], Brent [2], Sedgewick, Szymanski and Yao [22], Quisquater and Delescaille [18], and Nivasch [16]. Among these methods, using a distinguished point proposed by Quisquater and Delescaille [18] is known for the most efficient method. A point is called a distinguished point if the point satisfies a certain condition, which is easily checked.

The method using a distinguished point is as follows: After one process of iteration function, if the resulting point is a distinguished point, then the point stores in the table. The algorithm ends when a collision is detected among the stored distinguished points. Thus, we should define a condition of distinguished points properly so that not too much store the distinguished points but enough efficient.

2.3 Tag Tracing

For discrete logarithms in multiplicative groups of finite field, Cheon et al. [4] is introduced tag tracing. Likewise the Pollard rho algorithm, this algorithm generates a random walk with r-adding iteration function and detects a collision with a distinguished point method. The idea of tag tracing is that we do not need to compute full computation because the probability of meeting a distinguished point is very low. They show that for most iterations, it suffice to compute just the index and perform a collision detection with a distinguished point method. This algorithm is expected fast because computing an index value has less time complexity than full computation. Using this technique, we can solve DLP in finite field 10 times speedup for 1024-bit

prime fields.

Now, we recall the detail of tag tracing technique. Let G be a cyclic group of order p with a generator g and h a target element and $\mathcal{S}$ an index set and a multiplier set $\mathcal{M} = \{M_i\}_{i \in \mathcal{S}}$, suitable for an adding walk. For small ℓ, define $\mathcal{M}_\ell = (\mathcal{M} \cup \{1\})^\ell$ and precompute all elements of $\mathcal{M}_\ell$ before running the algorithm. Also, suppose there exists a function $\bar{\iota} : G \times \mathcal{M}_\ell \to \mathcal{S}$ such that $\bar{\iota}(g, M) = \iota(gM)$. This function contains some information of g such as the most significant bit (MSB) of g.

Given $g_i \in G$, we can obtain the index value $s_i = \iota(g_i)$ and set $g_{i+1} = g_i M_{s_i}$, which means do not compute $g_i M_{s_i}$. Since $\iota(g_{i+1}) = \bar{\iota}(g_i, M_{s_i}) = s_{i+1}$, we can also get s_{i+1} without full computing g_{i+1}. Similarly, set $g_{i+2} = g_{i+1} M_{s_{i+1}} = g_i M_{s_i} M_{s_{i+1}}$ without computing. Since $M_{s_i} M_{s_{i+1}}$ is pre-computed, we can get $s_{i+2} = \iota(g_{i+2}) = \bar{\iota}(g_i, M_{s_i} M_{s_{i+1}})$. Repeat the same process until arriving at ℓ iterations or finding the same distinguished point in the table. If we arrive at ℓ iterations, then do the full computation $g_k = g_i \cdot (M_{s_i} M_{s_{i+1}} \cdots M_{s_{i+\ell-1}})$. Note that this full computation costs the only one product because all elements of $\mathcal{M}_\ell$ are pre-computed.

Chapter 3

Multiplicative Inversion Algorithms

Since computing multiplicative inverse element is the most expensive among the finite field arithmetic, this operation should be designed efficiently. The most efficient algorithm for a binary field $GF(2^n)$ is the algorithm known as the Almost Inverse Algorithm (AIA) [21] and its modified version [8]. For a prime field $GF(p)$, the binary extended Euclidean algorithm is the efficient algorithm [3]. However, these algorithms are not efficient for the finite extension field. For the finite extension field, two inversion algorithms are widely used, one is extended Euclidean Algorithm (EEA) and Itoh-Tsujii Inversion Algorithm (ITI) [9, 1]. There are three variants of EEA, one is a polynomial version of EEA (IE), another is a parallel version of EEA (IP), and the other is a field inversion using multiplication (IM) [14]. According to [14], algorithm IM is the most efficient in software implementation of the finite extension field, however, in a hardware implementation of the finite extension field, Itoh-Tsujii inversion algorithm is the best performance [13]. In addition, there is another way to compute inverse by using inverse matrix [11].

From now on, we describe these algorithms and the complexity of these algorithms.

3.1 Extended Euclidean Algorithm

Suppose two polynomials $f(x), p(x) \in GF(p^m)$ are given. Using the extended Euclidean algorithm, we can obtain the polynomials $s(x), r(x) \in GF(p^m)$ such that $s(x)f(x)+r(x)p(x) = \gcd(f(x), p(x))$. If $p(x)$ is an irreducible polynomial of degree m, then $\gcd(f(x), p(x)) = 1$ and $s(x)f(x) \equiv 1 \mod p(x)$, which implies $s(x)$ is a multiplicative inverse of $f(x)$ modulo $p(x)$.

As we saw above, there are three variants EEA, called IE, IP, and IM. In this chapter, we briefly look into how each algorithm works. There are some notations which appear in the algorithms. The capital letters denote the polynomial corresponding to elements of the finite extension field, and the lowercase letters denote the subfield elements or integers. The subscript on the capital letters means the coefficient of polynomial whose degree is equal to the subscript. For example, F_0 means the constant term of $F(x)$ and $F_{\deg F(x)}$ means the leading coefficient of $F(x)$.

3.1.1 Algorithm IE

At first, we describe the extended Euclidean algorithm for polynomial version, called algorithm IE. This is the traditional extended Euclidean algorithm and well-known algorithm. After one iteration, the degree of larger polynomial is reduced at least one and the repeated process is ended when $\deg P(x) = 0$. Originally, this algorithm produces $s(x)$ and $r(x)$, however, the multiplicative inverse does not have any relation with $r(x)$, so we do not store and calculate $r(x)$.

3.1.2 Algorithm IP

In line 7 of algorithm IE, one inversion is required at each iteration. Although this operation is the subfield inversion, algorithm IE requires $2n + 2$ iterations with n is the degree of the larger polynomial, so it is burden to compute $2n + 2$ subfield inversions. The author of [14] try to reduce the number of the subfield inversion by using some parallelism.

The different thing compared to algorithm IE is that after one iteration, the degree of larger polynomial is reduced at least two. Algorithm IE only

Algorithm 2 Algorithm IE

Input : $f(x), p(x) \in \mathbb{F}_q[x]$ and $\deg f(x) < \deg p(x) = n$

Output : $s(x) \in \mathbb{F}_q[x]$ such that $f(x)s(x) \equiv 1 \mod p(x)$

1: $s(x) \leftarrow 0,\ r(x) \leftarrow 1,\ P(x) \leftarrow p(x),\ F(x) \leftarrow f(x)$
2: **while** $\deg P(x) \neq 0$ **do**
3: **if** $\deg P(x) < \deg F(x)$ **then**
4: exchange $P(x), s(x)$ with $F(x), C(x)$, respectively.
5: **end if**
6: $j \leftarrow \deg P(x) - \deg F(x)$
7: $\alpha \leftarrow P_{\deg P(x)} F^{-1}_{\deg F(x)}$
8: $P(x) \leftarrow P(x) - \alpha x^j F(x)$
9: $s(x) \leftarrow s(x) - \alpha x^j r(x)$
10: **end while**
11: $s(x) \leftarrow P_0^{-1} s(x)$
12: **return** $s(x)$

removes the leading coefficient of $P(x)$, but algorithm IP removes the leading coefficient and the next leading coefficient of $P(x)$.

3.1.3 Algorithm IM

The number of subfield inversions we required in algorithm IP is reduced by half than algorithm IE. However, it is burden to compute subfield inversion when p increases, so the algorithm which computes the minimize the number of subfield inversions is needed. The author of [14] can find a variant of EEA which is needed the only one subfield inversion. Like algorithm IP, the degree of larger polynomial is reduced at least two after one iteration. In algorithm IP, the inverse of the leading coefficient of $F(x)$ is required when we subtract $F(x)$ from $P(x)$, however, in algorithm IM, we need not to compute the inverse of the leading coefficient of $F(x)$ and multiply the leading coefficients and the next leading coefficients of $F(x)$ and $P(x)$ by $P(x)$ and $F(x)$, respectively. Thus, algorithm IM does not needed any inversion algorithm at iteration and the only subfield inversion is needed in line 13. But, it looks like more complicated, which means the number of multiplications

Algorithm 3 Algorithm IP

Input : $f(x), p(x) \in \mathbb{F}_q[x]$ and $\deg f(x) < \deg p(x) = n$

Output : $s(x) \in \mathbb{F}_q[x]$ such that $f(x)s(x) \equiv 1 \mod p(x)$

1: $s(x) \leftarrow 0,\ r(x) \leftarrow 1,\ P(x) \leftarrow p(x),\ F(x) \leftarrow f(x)$
2: **while** $\deg P(x) \neq 0$ **do**
3: **if** $\deg P(x) < \deg F(x)$ **then**
4: exchange $P(x), s(x)$ with $F(x), r(x)$, respectively.
5: **end if**
6: $j \leftarrow \deg P(x) - \deg F(x)$
7: $\alpha \leftarrow P_{\deg P(x)} F^{-1}_{\deg F(x)}$
8: $\beta \leftarrow (P_{\deg P(x)-1} - \alpha F_{\deg F(x)-1}) F^{-1}_{\deg F(x)}$
9: $P(x) \leftarrow P(x) - (\alpha x^j + \beta x^{j-1}) F(x)$
10: $s(x) \leftarrow s(x) - (\alpha x^j + \beta x^{j-1}) r(x)$
11: **if** $\deg P(x) = \deg F(x)$ **then**
12: $\alpha \leftarrow P_{\deg P(x)} F^{-1}_{\deg F(x)}$
13: $P(x) \leftarrow P(x) - \alpha F(x),\ s(x) \leftarrow s(x) - \alpha r(x)$
14: **end if**
15: **end while**
16: $s(x) \leftarrow P_0^{-1} s(x)$
17: **return** $s(x)$

required in algorithm IM is larger than before. The worst case of algorithm IM is $\deg f(x) = n-1$ and $n-1$ iterations are required, the same as algorithm IP.

Algorithm 4 Algorithm IM

Input : $f(x), p(x) \in \mathbb{F}_q[x]$ and $\deg f(x) < \deg p(x) = n$
Output : $s(x) \in \mathbb{F}_q[x]$ such that $f(x)s(x) \equiv 1 \mod p(x)$

1: $s(x) \leftarrow 0,\ r(x) \leftarrow 1,\ P(x) \leftarrow p(x),\ F(x) \leftarrow f(x)$
2: **while** $\deg P(x) \neq 0$ **do**
3: **if** $\deg P(x) < \deg F(x)$ **then**
4: exchange $P(x), s(x)$ with $F(x), r(x)$, respectively.
5: **end if**
6: $j \leftarrow \deg P(x) - \deg F(x)$
7: $\alpha \leftarrow F^2_{\deg F(x)}$
8: $\beta \leftarrow P_{\deg P(x)} F_{\deg F(x)}$
9: $\gamma \leftarrow F_{\deg F(x)} P_{\deg P(x)-1} - P_{\deg P(x)} F_{\deg F(x)-1}$
10: $P(x) \leftarrow \alpha P(x) - (\beta x^j + \gamma x^{j-1}) F(x)$
11: $s(x) \leftarrow \alpha s(x) - (\beta x^j + \gamma x^{j-1}) r(x)$
12: **if** $\deg P(x) = \deg F(x)$ **then**
13: $P(x) \leftarrow F_{\deg P(x)} P(x) - P_{\deg P(x)} F(x)$
14: $s(x) \leftarrow P_{\deg P(x)} s(x) - P_{\deg P(x)} r(x)$
15: **end if**
16: **end while**
17: $s(x) \leftarrow P_0^{-1} s(x)$
18: **return** $s(x)$

3.2 Itoh-Tsujii Inversion Algorithm

The Itoh-Tsujii inversion algorithm is used to obtain an inverse of element in a finite field. Actually, this algorithm is first proposed over a binary field $GF(2^n)$ using normal basis [9]. However, this can be applied to polynomial basis, and in any finite field [1].

Itoh-Tsujii inversion algorithm looks very simple.

Algorithm 5 Itoh-Tsujii Inversion Algorithm

Input : $A \in GF(p^m)$

Output : $B \in GF(p^m)$ such that $AB = 1$ in $GF(p^m)$.

1: $r \leftarrow \frac{p^m-1}{p-1}$
2: $C \leftarrow A^{r-1}$ using an addition chain
3: $D \leftarrow (C \cdot A)^{-1}$
4: $B \leftarrow D \cdot C$
5: **return** B

Since $r = \frac{q^m-1}{q-1}$ and $(A^r)^{q-1} = A^{q^m-1} = 1$ over $\mathbb{F}_{q^m}$, A^r is an element in $\mathbb{F}_q$. This means the line 3 involves an inversion operation in the subfield $\mathbb{F}_q$, and line 5 corresponds to a scalar multiplication to a polynomial in $GF(p^m)$. The most time-consuming in algorithm 5 is in line 2. This is why the algorithm 5 is well-suited for the normal basis. However, the multiplication two elements represented by normal basis over $GF(p^m)$ is expensive. Without using an addition chain, we can obtain A^{r-1} by computing $A^q, A^{q^2}, \ldots, A^{q^{m-1}}$ and then, multiply all. But, in this way, the number of q-th power and multiplication we need to compute A^{r-1} are $m-1$ and $m-2$, respectively. Using an addition chain, the number of these operations can be minimized . From now on, we give a definition of an addition chain and an example of computing A^{r-1} when $m = 5$.

Definition 3.2.1 (Addition Chain)**.** An addition chain for a positive integer n is a set of integers $a_0 = 1 < a_1 < a_2 < \cdots < a_r = n$ such that every element a_k can be written as sum $a_i + a_j$ with $1 \leq i, j < k \leq r$. r is the length of the addition chain.

Given a positive integer m, we can compute A^{r-1} with the minimum number of q-th power and multiplications if we can find the minimum length of an addition chain. Unfortunately, this is not easy and according to [7], finding the minimum length of additional-sequence problem, generalization of addition chain problem, is NP-complete.

Now, we give an example of computing A^{r-1} when $m = 5$. Using Algorithm 6, by computing 3 p-th power and 2 multiplications, we can obtain A^{r-1}.

Algorithm 6 Addition Chain for A^{r-1} in $GF(p^5)$

Input : $A \in GF(p^5)$
Output : $B = A^{r-1}$ with $r = \frac{p^5-1}{p-1}$

1: $B_0 \leftarrow A^p$
2: $B_0 \leftarrow B_0 A$ ▷ $B_0 = A^{p+1}$
3: $B_0 \leftarrow B_0^p$ ▷ $B_0 = A^{p^2+p}$
4: $B_1 \leftarrow B_0^{p^2}$ ▷ $B_1 = A^{p^4+p^3}$
5: $B \leftarrow B_0 B_1$ ▷ $B = A^{p^4+p^3+p^2+p}$
6: **return** B

3.3 Inverse Matrix

According to [11], we can obtain an inverse element of an finite extension field using inverse matrix. Assume $GF(q^m)$ is isomorphic to $\mathbb{F}_q[x]/(p(x))$, where $p(x)$ is the irreducible polynomial of the form $x^n+\beta$. Then, when we multiply two polynomials $a(x)$ and $b(x)$ in $GF(q^m)$, we can derive the matrix which is corresponding to polynomial $a(x)$, and this matrix has a special form. If $b(x)$ is the inverse element of $a(x)$, which means $a(x)b(x) = 1$ in $\mathbb{F}_q[x]/(p(x))$, then the inverse of the corresponding matrix of $a(x)$ contains the coefficient of $b(x)$. Actually, the first column consists of the coefficients of $b(x)$. Also, this matrix has a special form, so we expect to compute inverse matrix of this form faster than other matrix. As a result, when the degree of the extension filed is low, this method is slightly faster than extended Euclidean algorithm and Itoh-Tsujii inversion algorithm. Details are follows. In [11], they show an algorithm to compute inverse element when $m = 3$, but we give how to compute inverse element generally.

Let $GF(p^n) = \mathbb{F}_q[x]/(p(x))$, where $p(x) = x^n + \beta$ is an irreducible polynomial of degree n. Then, we can consider an element of $\mathbb{F}_{q^n}$ as a polynomial $f(x)$ of degree less than n. Suppose $a(x) = \sum_{i=0}^{n-1} a_i x^i$, where $a_i \in \mathbb{F}_q$. Let $b(x) = \sum_{i=0}^{n-1} b_i x^i$ be an inverse of $a(x)$, which means $a(x)b(x) = 1$ in $\mathbb{F}_q[x]/(p(x))$. Then, $1 = a(x)b(x) \mod p(x) = c_{n-1}x^{n-1} + \sum_{i=0}^{n-2}(c_i - \beta c_{i+n})x^i$,

where $c_k = \sum_{i+j=k} a_i b_j$ for $0 \le k \le n-1$. Hence,

$$\begin{aligned}
1 &= a_0 b_0 - \beta(a_{n-1}b_1 + \cdots + a_1 b_{n-1}) \\
0 &= a_1 b_0 + a_0 b_1 - \beta(a_{n-1}b_2 + \cdots + a_2 b_{n-1}) \\
&\vdots \\
0 &= a_{n-2} b_0 + \cdots + a_0 b_{n-2} - \beta a_{n-1} b_{n-1} \\
0 &= a_{n-1} b_0 + a_{n-2} b_1 + \cdots + a_0 b_{n-1}.
\end{aligned}$$

With these n equations, we express into a matrix representation.

$$\begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \\ 0 \end{pmatrix} = \underbrace{\begin{pmatrix} a_0 & -\beta a_{n-1} & -\beta a_{n-2} & \cdots & -\beta a_1 \\ a_1 & a_0 & -\beta a_{n-1} & \cdots & -\beta a_2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n-2} & a_{n-3} & a_{n-4} & \cdots & -\beta a_{n-1} \\ a_{n-1} & a_{n-2} & a_{n-3} & \cdots & a_0 \end{pmatrix}}_{M} \begin{pmatrix} b_0 \\ b_1 \\ \vdots \\ b_{n-2} \\ b_{n-1} \end{pmatrix}$$

and

$$\begin{pmatrix} b_0 \\ b_1 \\ \vdots \\ b_{n-2} \\ b_{n-1} \end{pmatrix} = \begin{pmatrix} a_0 & -\beta a_{n-1} & -\beta a_{n-2} & \cdots & -\beta a_1 \\ a_1 & a_0 & -\beta a_{n-1} & \cdots & -\beta a_2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n-2} & a_{n-3} & a_{n-4} & \cdots & -\beta a_{n-1} \\ a_{n-1} & a_{n-2} & a_{n-3} & \cdots & a_0 \end{pmatrix}^{-1} \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \\ 0 \end{pmatrix}.$$

Let M be the corresponding matrix of $a(x)$. Note that the right hand side of second matrix equation only survives the first column of M. So, by only computing the first column of M^{-1}, we can get b_i for all i, which is the coefficient of the inverse polynomial $b(x)$.

In this way, the most time-consuming operation is the finding the inverse matrix. Now, we focus on the method of calculating inverse matrix efficiently. Mostly, the methods finding inverse matrix are using Gaussian elimination and Cramer's rule. However, when we use Gaussian elimination, we cannot

find special algorithm for M. Just do Gaussian elimination to

$$\begin{pmatrix} a_0 & -\beta a_{n-1} & -\beta a_{n-2} & \cdots & -\beta a_1 & 1 \\ a_1 & a_0 & -\beta a_{n-1} & \cdots & -\beta a_2 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n-2} & a_{n-3} & a_{n-4} & \cdots & -\beta a_{n-1} & 0 \\ a_{n-1} & a_{n-2} & a_{n-3} & \cdots & a_0 & 0 \end{pmatrix}$$

and then, we can obtain the first column of the inverse matrix. Since we want to compute the first column of inverse matrix, the first column of the identity matrix is attached to M. This method is nothing special compared to other matrix.

Let $B = (b_{i,j})_{1 \le i,j \le n}$ be an inverse matrix of M and $M_{i,j}$ a (i,j)-minor matrix of M. Then, $b_i = b_{i,1} = \det B_{1,i} / \det B$ for all $1 \le i \le n$. Thus, we need to compute a determinant of $n \times n$ matrix and n determinants of $(n-1) \times (n-1)$ matrix. Note that the same sub-matrix is contained in M, such as

$$\begin{pmatrix} a_0 & -\beta a_{n-1} \\ a_1 & a_0 \end{pmatrix}, \begin{pmatrix} a_2 & a_1 & a_0 \\ a_3 & a_2 & a_1 \\ a_4 & a_3 & a_2 \end{pmatrix} \cdots$$

During computing the determinant of M and $M_{i,j}$, store determinants if we do not meet a matrix before and load the value when we meet the same matrix. Thus, we can get determinants more fast than usual matrix. Although there is another way to find the determinant such as Dodgson's condensation [6], required the number of multiplications is larger than Cramer's rule.

Theses methods are quite simple, but high complexity when n increases. Even if we use Cramer's rule as stated above, the complexity of Gaussian elimination and Cramer's rule is $O(n^3)$, so we can conclude this method is faster only when n is small, which means the degree of extension field is small.

3.4 Complexity

In this chapter, we analyze the complexity of inversion algorithms over the finite extension field. Among the finite field arithmetic, inversion and mul-

tiplication have the high complexity. We expect the algorithm is fast when these operations are required less. Therefore, we focus on the number of theses operations for each algorithm.

Let M and I denote the average cost of multiplication and inversion, respectively. As you know, when we analyze the complexity, we consider the worst case.

In Algorithm IE, $\deg f(x) = n - 1$ is the worst case. After one iteration, the degree of larger polynomial is reduced at least one, and the repeated process is ended when $\deg P(x) = 0$, which implies $P(x)$ is an element of the subfield. $2(n-1)$ iteration are required and thus, $2(n-1)$ inversion are required.

In Algorithm IP, $\deg f(x) = n - 1$ is the worst case and the repeated process is ended when $\deg P(x) = 0$, too. However, after one iteration, the degree of larger polynomial is reduced at least two, so $n + 1$ iterations are required and thus, $n + 1$ inversion are required.

Algorithm IM is similar to Algorithm IP. This algorithm also reduce the degree of larger polynomial at least two after one iteration and $\deg f(x) = n-1$ is the worst case, so $n-1$ iterations are required. However, this algorithm is not needed any inversion algorithm at iteration and the only subfield inversion is needed in line 13. But, it looks like more complicated, which means the number of multiplications required in Algorithm 4 is larger than before.

Algorithms	#M	#I
IE	$2n^2 + n - 4$	$2n - 1$
IP	$2n^2 + n - 4$	$n + 1$
IM	$3n^2 - n - 7$	1

Table 3.1: Efficiency Comparison between Inversion Algorithms

Chapter 4

Elliptic Curve Discrete Logarithm Problem

Our goal is solving discrete logarithm problem in elliptic curves faster. As you seen before in Section 2.2, using the Pollard rho algorithm is the most efficient algorithm solving DLP in a finite field with $O(\sqrt{p})$ time complexity and negligible memory complexity [17]. Also, applying tag tracing, we can reduce the time about $\frac{1}{10}$ for 1024-bit prime field [4]. Likewise, when we solve ECDLP, we use the Pollard rho algorithm and to be more efficient, we want to apply the technique of tag tracing. To do that, we need some preparation and pre-computation to use tag tracing in elliptic curves. Basically, the process is almost the same used in [4], just modify fit for elliptic curves. Now, we give the details in this section.

4.1 Preparation

To accelerate the Pollard rho algorithm with tag tracing, there are something to do before running the algorithm.

Suppose an elliptic curve is defined over the finite extension field $GF(q^m)$. Since one addition in elliptic curve actually requires one inversion and three multiplications in $GF(q^m)$. Among these operations, inversion is the most time-consuming operation, and we need to improve the inversion algorithm

to speedup Pollard rho algorithm in elliptic curves. Refer to Section 3, every inversion algorithm needs the subfield inversion operation. We may assume we can find the inverse element in $GF(q^m)$ fast than before that if we know the inverse element in the prime field before running the inversion algorithm. Therefore, we decide that we compute the inverse element in $\mathbb{F}_q$ during the preparation phase.

For small r and ℓ, let $\mathcal{S} = \{0, 1, \ldots, r-1\}$ be an index set and $\mathcal{M}_\ell$ a set of addition of a multiplier.

In other words, we have the two pre-computation tables. One has an element of $\mathbb{F}_q$ and its inverse element and the other has all elements of $\mathcal{M}_\ell$ with their coefficients. Thus, we can find the inverse element of $\mathbb{F}_q$ and all values of $\mathcal{M}_\ell$ by searching the tables whenever we want.

4.2 Tag Tracing

To run the Pollard rho method with tag tracing on the elliptic curves, the index function ι, which determines the index value of $P \in E(\mathbb{F}_{q^n})$, the r-adding walk f_r, and the collision detection method are required. Now, we review the definition of tag set, the tag function and the index function [4].

Let a tag set $\mathcal{T} = \{h(x) \in \mathbb{F}_q[x] : \deg h(x) < t \text{ for } t \leq n\}$. The tag function $\tau : \mathbb{F}_q[x]/(p(x)) \to \mathcal{T}$ is defined to be $\tau(h(x)) = h(x) \mod x^t$, with $\deg p(x) = n$. Given two polynomials $a(x)$ and $b(x) \in \mathbb{F}_q[x]$, with $a(x) \mod p(x) = \sum_{i=0}^{n-1} a_i x^i$ and $a_i \in \mathbb{F}_q$,

$$a(x)b(x) \mod p(x) = \sum_{i=0}^{n-1} a_i \{x^i b(x) \mod p(x)\} \mod p(x).$$

So, $\tau(a(x)b(x)) = \sum_{i=0}^{n-1} a_i \tau(x^i b(x))$. Choose a positive integer r such that $r \leq q$. Define $\sigma : \mathcal{T} \to \mathcal{S} = \{0, 1, \cdots, r-1\}$ by $\sigma(h_0) = h_0 \mod r$ and the

index function $\iota : G \to \mathcal{S}$ is defined to be $s = \sigma \circ \tau$. Then,

$$\begin{aligned}\iota(a(x)b(x)) &= \sigma \circ \tau(a(x)b(x)) \\ &= \sigma\Big(\sum_{i=0}^{n-1} a_i \tau(x^i b(x))\Big)\end{aligned}$$

Now, we define the tag function and index function suitably for our setting. In our case, $t = 1$ is enough, which implies the tag set means the constant term of polynomial and the tag function maps to a constant term in $\mathbb{F}_q[x]/(p(x))$. In other words, if we get any coefficient of the polynomial, not even constant term, then we use that coefficient as the tag.

Given two distinct points $P = (x_1, y_1), Q = (x_2, y_2) \in E(\mathbb{F}_{q^n})$, the x-coordinate of $P+Q$ is $(\frac{y_1-y_2}{x_1-x_2})^2 - x_1 - x_2$. Since $GF(q^n) = \mathbb{F}_q[x]/(p(x))$, where $p(x) = x^n + \beta$ is an irreducible polynomial of degree n, we consider an element of $GF(q^n)$ as a polynomial over a finite field $\mathbb{F}_q$, so there are polynomials $a(x), b(x)$ and $c(x)$ corresponding to $(x_1 - x_2)^2, (y_1 - y_2)^2$, and $-x_1 - x_2$, respectively. Then, we can write x-coordinate of $P + Q$ is $a(x)^{-1}b(x) + c(x)$ mod $p(x)$. We want to get an index of $P + Q$ without full computing.

Note that given two polynomials $f(x)$ and $g(x)$ in $\mathbb{F}_q[x]/(p(x))$, where $p(x)$ is the irreducible polynomial $x^n + \beta$, we show that a multiplication of two polynomial $f(x)g(x)$ can be expressed into a matrix multiplication in Section 3.3. Suppose $a(x)d(x) = b(x)$ for unknown polynomial $d(x)$ and we want to find any one of coefficient of $d(x)$. Consider the $n \times (n+1)$ matrix

$$M_1 := \begin{pmatrix} a_0 & -\beta a_{n-1} & -\beta a_{n-2} & \cdots & -\beta a_1 & b_0 \\ a_1 & a_0 & -\beta a_{n-1} & \cdots & -\beta a_2 & b_1 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n-2} & a_{n-3} & a_{n-4} & \cdots & -\beta a_{n-1} & b_{n-2} \\ a_{n-1} & a_{n-2} & a_{n-3} & \cdots & a_0 & b_{n-1} \end{pmatrix} = \left(\begin{array}{c|c} & b_0 \\ & b_1 \\ M & \vdots \\ & b_{n-2} \\ & b_{n-1} \end{array}\right)$$

where M is the corresponding matrix of $a(x)$. When we use the Gaussian elimination to M_1 and then left square matrix becomes identity matrix, $b_0, b_1, \ldots, b_{n-1}$ are changed into $d_0, d_1, \ldots, d_{n-1}$. However, we use just one coefficient as the tag, so we do not need to compute all coefficients of $d(x)$.

Recall the Gaussian elimination process. At first, we make the matrix M to the upper triangular matrix whose diagonal entries are 1, which called the forward step. And make the upper triangular matrix to identity matrix by backward row operations. Then, the last column of M_1 is the solution $d(x)$ satisfying $a(x)d(x) = b(x)$. Since we do not need all coefficients of $d(x)$, we need not to do the backward row operation because we obtain d_{n-1} after forward step. From this fact, we can define the tag set $\mathcal{T}$ as the leading coefficient of the polynomial in $GF(p^m)$, and get an index from the tag.

Above paragraph means we compute the index of some element of $P \in E(\mathbb{F}_{q^n})$ without calculating inversion. Since we do not compute not only full computation of an addition in elliptic curves, but also computing inversion, we expect to reduce the time and then, we solve DLP with the less time. In fact, required the number of multiplications of this method is $\sum_{k=1}^{n} k^2$, which is asymptotic $O(n^3)$. Although we do not continue every inversion process and compute an addition in elliptic curves, this is natural because the complexity of the forward step in Gaussian elimination is also $O(n^3)$.

Chapter 5

Implementation Results

In this chapter, we present our experimental results and compare the time to solve ECDLP using Pollard rho algorithm and tag tracing algorithm for elliptic curves as we proposed in Chapter 4. Our experimental environment is 64-bit quad-core Intel i7-2600 processor, running at 3.4G Hz and 16GB of RAM. We uses Shoup's NTL library version 6.0.0 and GNU's GMP library version 5.1.1 and the code was compiled using gcc complier version 4.2.1.

Since elliptic curves cryptosystems are secure at least 160 bits, we choose $\lfloor \frac{160}{n} \rfloor$-bit prime where n is the degree of the extension field. For example, if the extension degree is 4, we choose a 40-bit prime and when the extension degree is 5, then we choose a 32-bit prime. Since solving ECDLP in 160-bit group takes too much time, we solve ECDLP in the subgroup of $E(\mathbb{F}_{q^n})$ and the order of the subgroup is about 36 bits.

The main results of our experimental are summarized in Table 5.

	Prime Order	Subgroup Order	Pollard Time	Tag Time	Tag / Pollard	Arithmetic	
						Addition	Tag
deg 4	40 bit	36 bit	7.58s	3.94s	0.52	21.2μs	6.0μs
deg 5	32 bit	35 bit	11.92s	7.27s	0.61	26.4μs	11.6μs
deg 6	27 bit	37 bit	16.33s	14.78s	0.90	29.6μs	13.2μs

Table 5.1: Time Comparison with $r = 4$ and $\ell = 6$

Chapter 6

Conclusion

To apply tag tracing, we investigate some inversion algorithms for a finite extension field. Since we obtain the subfield inversion in preparation phase, Algorithm 3 is the most effective inversion algorithm because this algorithm requires the least number of p-bit multiplications and modulo reductions. Also, we give a tag function and an index function fit for elliptic curves to accelerate the Pollard rho algorithm.

As a further work, we would like to find the index function ι such that

$$\iota\Big(f(x)g(x)^{-1} \mod p(x)\Big) = \iota\Big(f(x)\Big) * \iota\Big(g(x)\Big)^{-1}$$

If we can find an index function having this property, then we need not to use the inversion algorithm.

Bibliography

[1] D. V. Bailey and C. Paar. Efficient arithmetic in finite field extensions with application in elliptic curve cryptography. *J. Cryptology*, 14(3), 2001.

[2] R. Brent. An improved monte carlo factorization algorithm. *BIT Numerical Mathematics*, 20(2):176–184, 1980.

[3] M. Brown, D. Hankerson, J. López, and A. Menezes. Software implementation of the NIST elliptic curves over prime fields. In D. Naccache, editor, *Topics in Cryptology - CT-RSA 2001*, volume 2020 of *Lecture Notes in Computer Science*, pages 250–265. Springer, 2001.

[4] J. H. Cheon, J. Hong, and M. Kim. Speeding up the pollard rho method on prime fields. In J. Pieprzyk, editor, *Advances in Cryptology - ASIACRYPT 2008*, volume 5350 of *Lecture Notes in Computer Science*, pages 471–488. Springer, 2008.

[5] W. Diffie and M. Hellman. New directions in cryptography. *Information Theory, IEEE Transactions on*, 22(6):644–654, 1976.

[6] C. L. Dodgson. Condensation of determinants, being a new and brief method for computing their arithmetical values. *Proceedings of the Royal Society of London*, 15:pp. 150–155, 1866.

[7] P. Downey, B. Leong, and R. Sethi. Computing sequences with addition chains. *SIAM Journal on Computing*, 10(3):638–646, 1981.

[8] D. Hankerson, J. L. Hernandez, and A. Menezes. Software implementation of elliptic curve cryptography over binary fields. In Çetin Kaya Koç

and C. Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2000*, volume 1965 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 2000.

[9] T. Itoh and S. Tsujii. A fast algorithm for computing multiplicative inverses in $GF(2^m)$ using normal bases. *Inf. Comput.*, 78(3):171–177, 1988.

[10] D. Knuth. *The Art of Computer Programming: Seminumerical algorithms*. Number V. 2 in Addison-Wesley series in computer science and information processing. Addison-Wesley, 1969.

[11] T. Kobayashi, H. Morita, K. Kobayashi, and F. Hoshino. Fast elliptic curve algorithm combining frobenius map and table reference to adapt to higher characteristic. In J. Stern, editor, *Advances in Cryptology - EUROCRYPT '99*, volume 1592 of *Lecture Notes in Computer Science*, pages 176–189. Springer, 1999.

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

[13] M.-K. Lee, K. T. Kim, H. Kim, and D. K. Kim. Efficient hardware implementation of elliptic curve cryptography over $GF(p^m)$. In J. Song, T. Kwon, and M. Yung, editors, *Information Security Applications, WISA*, volume 3786 of *Lecture Notes in Computer Science*, pages 207–217. Springer, 2005.

[14] C. H. Lim and H. S. Hwang. Fast implementation of elliptic curve arithmetic in $GF(p^n)$. In H. Imai and Y. Zheng, editors, *Public Key Cryptography, PKC 2000*, volume 1751 of *Lecture Notes in Computer Science*, pages 405–421. Springer, 2000.

[15] V. S. Miller. Use of elliptic curves in cryptography. In H. C. Williams, editor, *Advances in Cryptology - CRYPTO '85*, volume 218 of *Lecture Notes in Computer Science*, pages 417–426. Springer, 1985.

[16] G. Nivasch. Cycle detection using a stack. *Information Processing Letters*, 90:135–140, 2004.

[17] J. Pollard. Monte Carlo methods for index computation (mod p). *Mathematics of Computation*, 32:918–924, 1978.

[18] J.-J. Quisquater and J.-P. Delescaille. How easy is collision search. new results and applications to des. In G. Brassard, editor, *Advances in Cryptology - CRYPTO '89*, volume 435 of *Lecture Notes in Computer Science*, pages 408–413. Springer, 1989.

[19] J. Sattler and C. Schnorr. Generating random walks in groups. *Ann. Univ. Sci. Budapest. Sect. Compu.*, 6:65–79, 1985.

[20] C. P. Schnorr and H. W. Lenstra. A Monte Carlo factoring algorithm with linear storage. *Mathematics of Computation*, 43(167):289–311, 1984.

[21] R. Schroeppel, H. K. Orman, S. W. O'Malley, and O. Spatscheck. Fast key exchange with elliptic curve systems. In D. Coppersmith, editor, *Advances in Cryptology - CRYPTO '95*, volume 963 of *Lecture Notes in Computer Science*, pages 43–56. Springer, 1995.

[22] R. Sedgewick, T. Szymanski, and A. Yao. The complexity of finding cycles in periodic functions. *SIAM Journal on Computing*, 11(2):376–390, 1982.

[23] D. Shanks. Class number, a theory of factorization, and genera. In *1969 Number Theory Institute (Proc. Sympos. Pure Math., Vol. XX, State Univ. New York, Stony Brook, N.Y.)*, pages 415–440. Amer. Math. Soc., 1969.

[24] E. Teske. Speeding up pollard's rho method for computing discrete logarithms. In J. Buhler, editor, *Algorithmic Number Theory, ANTS*, volume 1423 of *Lecture Notes in Computer Science*, pages 541–554. Springer, 1998.

국문초록

이 논문에서 타원 곡선에서 태그 트레이싱 기법을 적용시켰다. 처음에 천정희 등이 제안한 태크 트레이싱은 폴라드로 알고리즘을 더욱 빠르게 돌도록 해준다. 이 방법을 이용하면 유한체에서 이산 대수 문제를 10배 더 빠르게 풀 수 있다. 그러나, 타원 곡선에서는 덧셈 연산 때문에 이정도의 효과를 보지 못한다. 덧셈을 하기 위해서는 역원 연산과 제곱 두 번을 포함한 곱셈 연산을 세 번을 해야한다. 역원 연산은 시간이 가장 많이 걸리는 연산이기 때문에 좀 더 빠르게 할 수 있도록 개선을 할 필요가 있다. 그래서 우리는 확장체에서 유용하게 사용되는 역원 알고리즘에 대해서 살펴보았다. 가장 널리 쓰이는 두 개의 알고리즘이 있는데, 하나는 확장된 유클리디안 알고리즘이고, 다른 하나는 이토-쯔지 알고리즘이다. 이 외에도, 역행렬을 이용하여 역원을 구할 수 있다. 우리는 이 알고리즘들의 계산 복잡도를 비교해보았고, 우리에 환경에 맞췄을 때 가장 빠른 역원 알고리즘을 비교 대상으로 삼았다. 그리고 복잡도를 계산해본 결과, 확장체의 차수가 낮은 경우에 역행렬을 이용하여 역원을 구하는 방법이 더 빠르다는 것을 알 수 있었다. 또한, 역행렬을 이용할 경우 역원의 전체를 다 구하지 않고 역원의 계수 하나를 빠르게 구할 수 있었다. 이 방법을 이용하여 우리는 확장체의 차수가 낮은 경우에 타원 곡선 이산 대수 문제를 이 전에 알려진 방법보다 더 빠르게 풀 수 있었다.

주요어휘: 역원 알고리즘, 확장 유클리디안 알고리즘, 태그 트레이싱, 타원 곡선 이산 대수 문제
학번: 2010-23075

감사의 글

우선 논문이 나올 수 있도록 도움을 주신 분들께 감사드립니다. 여러모로 부족하고 모자란 저를 지도해주시고 이끌어주신 천정희 선생님께 감사드립니다. 현재에 안주하지 않고, 항상 멀리 내다볼 수 있도록 많은 가르침을 주셔서 저에게는 큰 도움이 되었습니다. 아직까지 많이 부족한 점이 많지만, 앞으로는 더욱 열심히 해서 선생님께서 원하시는 수준이 되도록 열심히 하겠습니다. 그리고 바쁜 시간을 쪼개서 논문심사를 맡아주신 오병권 선생님, 변동호 선생님께도 이 자리를 빌어 감사의 말씀 올리겠습니다.

석사과정으로 입학하여 새로운 환경에서 잘 적응할 수 있도록 많은 도움을 준 여러 형, 친구, 동생들에게도 감사의 말을 전합니다. 특히, 같이 KAIST를 졸업하고 대학원 생활까지 같이 생활하며 힘을 줬던 의찬이, 적응할 수 있도록 여러가지 도움을 준 병일이형, 동현이형, 공부와 농구를 같이 하며 친해진 현수형과 상연이, 힘들 때나 고민이 많을 때 저에게 많은 이야기를 해주고 옆에서 항상 용기를 줬던 웅배와 웅석이형, 경민이형 등 다른 석사 동기들 및 형들, 동생들에게도 감사하다고 전해주고 싶습니다. 그리고 어떠한 질문을 해도 잘 받아주는 형태형, 항상 옆에서 저를 챙겨주는 진수를 비롯해 모든 연구실 사람들에게 감사합니다.

마지막으로 지금까지 저를 길러주시고 항상 제 편이 되어주시는 아버지, 어머니 항상 감사합니다. 그리고 제 동생 지인이에게도 고맙다는 말을 하고 싶습니다. 앞으로 모든 일에 열심히 하여 더욱 발전하는 모습을 보이도록 하겠습니다.