A High-Speed Square Root Computation in Finite Fields with Application to Elliptic Curve Cryptosystem

Feng Wang, Yasuyuki Nogami, and Yoshitaka Morikawa†

†Department of Communication Network Engineering
Okayama University
3-1-1, Tsushima-naka, Okayama, 700-8530, Japan

(Received November 25, 2004)

In this paper, we focus on developing a high-speed square root (SQRT) algorithm required for an elliptic curve cryptosystem. Examining Smart algorithm, the previously well-known SQRT algorithm, we can see that there is a lot of computation overlap in Smart algorithm and the quadratic residue (QR) test, which must be implemented prior to a SQRT computation. It makes Smart algorithm inefficient. The essence of our proposition is thus to present a new QR test and an efficient SQRT algorithm to avoid all the overlapping computations. The authors devised a SQRT algorithm for which most of the data required have been computed in the proposed QR test. Not only there is no computation overlap in the proposed algorithm and the proposed QR test, but also in the proposed algorithm over $GF(p^2)$ $(4 \mid p-1)$ some computations can be executed in $GF(p)$; whereas in Smart algorithm over $GF(p^2)$ all the computations must be executed in $GF(p^2)$. These yield many reductions in the computational time and complexity. We implemented the two QR tests and the two SQRT algorithms over $GF(p^m)$ $(m=1,2)$ in C++ language with NTL (Number Theory Library) on Pentium4 (2.6GHz), where the size of p is around 160 bits. The computer simulations showed that the proposed QR test and the proposed algorithm over $GF(p^m)$ were about 2 times faster than the conventional QR test and Smart algorithm over $GF(p^m)$.

1 Introduction

The importance of keeping communication secure is increasing due to the prevalence of the Internet and other forms of electronic communication. A public-key cryptosystem is well known as a technology to provide a secure environment where communication can be conducted without fear. As public-key cryptosystems, both Rivest-Shamir-Adleman (RSA) cryptosystem and elliptic curve cryptosystem (ECC) can provide secure communication. However, ECC offers equivalent security with smaller key sizes. For example, a 160-bit ECC guarantees as secure as a 1024-bit RSA cryptosystem, and it follows that ECC is more suitable for small devices such as smart cards and cellular telephones. Therefore ECC has tremendous potential to keep communication secure and much attention [1]-[3] has been attracted to the realization of ECC.

In realizing ECC, not only the fundamental operations such as multiplications and inversions, but also the square root (SQRT) computations must be implemented. For an around 160-bits ECC, implemented with aid of NTL (Number Theory library), a multiplication in prime fields $GF(p)$ needs to take 0.2 mil-

†E-mail: {wangfeng/nogami/morikawa}@trans.cne.okayama-u.ac.jp

lisecond; whereas a square root computation in $GF(p)$ needs to take 50 millisecond by using Smart algorithm which is the previously well-known SQRT algorithm. This is a big motivation to develop a high-speed SQRT algorithm over finite fields $GF(p^m)$.

ECC provides the user a great deal of flexibility in the choice of system parameters. The authors have considered the case of $m = rn$ [4], where $r = 1$ or 2 and n is an odd prime number. In this paper, we let p be a prime number greater than 3 and let m have the form of $m = 2^c$, where c is a nonnegative integer. However, ECC over $GF(p^{2^c})$ can provide secure communications if and only if $c = 0, 1$ and 2 [5]. In fact, its security is weak when $c = 2$ [5]. So, we mainly consider the two cases of $c = 0$ and 1. Without any additional explanations, $m = 1$ and 2 in what follows.

It is well known that if an arbitrary element x in $GF(p^m)$ is not a quadratic residue, i.e. the element does not have its SQRT in the same field, then it is nonsensical to compute its SQRT. Prior to the SQRT computation, we should thus identify whether or not x is a quadratic residue, which is called the quadratic residue test. Therefore, if most of data required for a SQRT algorithm have been computed in the quadratic residue test, then it will provide a means for a high-speed SQRT computation.

Smart algorithm [6], the previously well-known SQRT algorithm over $GF(p^m)$, has the idea how to compute the SQRT from the quadratic residue test. However there is a lot of computation overlap in the quadratic residue test and Smart algorithm, which makes the SQRT computation inefficient. The essence of our proposition is thus to present a new quadratic residue test and an efficient SQRT algorithm to avoid all the overlapping computations. Based on the main idea of Smart algorithm, the authors devised an algorithm, called the moving window-sign testing (MW-ST) algorithm, for which most of the data required have been computed in the proposed quadratic residue test. Not only there is no computation overlap in the proposed algorithm and the proposed quadratic residue test, but also in the proposed algorithm over $GF(p^2)$ $(4 \mid p-1)$ some computations can be executed in $GF(p)$. However, in Smart algorithm over $GF(p^2)$ all the computations must be executed in $GF(p^2)$. These yield many reductions in the computational time and complexity. In this paper, the MW-ST algorithm for SQRT computation over $GF(p^m)$ has been proved to be much faster than Smart algorithm.

The remainder of the paper is organized as follows. In Sec.2, we first present how to implement the quadratic residue test over $GF(p^m)$, and then estimate its computational complexity. In Sec.3, we first review Smart algorithm over $GF(p^m)$, and then estimate its computational complexity. In Sec.4, the MW-ST algorithm is first presented in $GF(p^m)$, and then we estimate its computational complexity in $GF(p)$ and $GF(p^2)$ respectively. In Sec.5, we implement the two QR tests and the two SQRT algorithms over $GF(p^m)$ in C++ language with NTL on Pentium4 (2.6GHz), where the size of p is around 160 bits. The computer simulations show that the proposed QR test and the proposed algorithm over $GF(p^m)$ are about 2 times faster than the conventional QR test and Smart algorithm over $GF(p^m)$.

Throughout the paper, A_m and M_m denote additions and multiplications in finite fields $GF(p^m)$, and S denotes shift computations. $\#A_m$, $\#M_m$ and $\#S$ respectively denote the numbers of these operations. Especially, $\#\overline{M}_m$ denotes the average number of the multiplications required for an arbitrary input element.

2 Quadratic residue test

In this section, we first present how to implement quadratic residue test, and then estimate its computational complexity.

2.1 How to implement the quadratic residue test

We usually use the Euler's criterion to identify whether or not a nonzero element x is a quadratic residue:

$$C(x) = x^{(p^m-1)/2} = \begin{cases} 1 & \text{QR} \\ -1 & \text{QNR} \end{cases}, \qquad (1)$$

where QR and QNR are the abbreviations of *quadratic residue* and *quadratic non-residue*, respectively. Conventionally, we directly compute $(p^m - 1)/2$-th power of x to implement the QR test as shown in Fig.1.

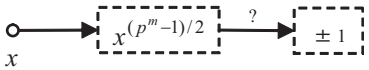

Figure 1: Conventional QR test for $x \in GF(p^m)$

The exponent in Eq.(1) i.e. $(p^m - 1)/2$ can be factorized as

$$(p^m - 1)/2 = 2^T s, \qquad (2)$$

where T is a nonnegative integer and s is an odd number. It follows that

$$C(x) = x^{(p^m-1)/2} = (x^s)^{2^T} = x_0^{2^T} \text{ for } x_0 = x^s. \quad (3)$$

In fact, it is not necessary for some input elements to compute such a large power of $x^{(p^m-1)/2}$. If $x_0 = 1$, then we can assert that x is a QR. If not, we repeatedly compute the squares of x_0 until the product becomes -1. This is just our proposed QR test.

Let t be the amount of the expected square computations as shown in Fig.2(a). When $t < T$, x is a QR such as the x of Fig.2(a); when $t = T$, x is a QNR such as the c of Fig.2(b). Note that when $x_0 = -1$, we do not need to compute the square of x_0, which implies that $t = 0$. For the convenience of the succeeding SQRT computation, we first compute $x^{(s-1)/2}$, and then multiply $x^{(s-1)/2}$ by x to get $x^{(s+1)/2}$, finally multiply $x^{(s+1)/2}$ and $x^{(s-1)/2}$ together to get x_0.

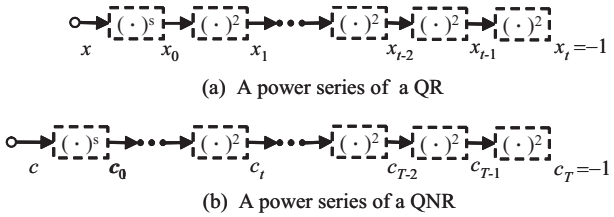

Figure 2: Proposed QR test for $x \in GF(p^m)$

2.2 Complexity of the QR test

In the conventional QR test, we directly compute the $(p^m-1)/2$−th power of x, which requires the following multiplications by using binary method [7]:

$$\begin{aligned} \#M_m &= \left\lfloor \log_2(\frac{p^m-1}{2}) \right\rfloor + w\left(\frac{p^m-1}{2}\right) - 1 \\ &= \lfloor \log_2(s) \rfloor + w(s) + T - 1, \end{aligned} \quad (4)$$

where T and s are given by Eq.(2), and $\lfloor \cdot \rfloor$ and $w(\cdot)$ denote the maximal integer less than $\cdot$ and the Hamming weight of $\cdot$. In what follows, we abbreviate the expression $\lfloor \log_2(\cdot) \rfloor + w(\cdot)$ to $LW(\cdot)$ for the convenience. Since s is an odd number, the following relations hold: $w(s) = w(\frac{s-1}{2}) + 1$ and $\lfloor \log_2(s) \rfloor = \lfloor \log_2(\frac{s-1}{2}) \rfloor + 1$. We, therefore, obtain

$$\#M_m = LW\left(\frac{s-1}{2}\right) + 1 + T. \quad (5)$$

In the proposed QR test, when using binary method to compute $x^{(s-1)/2}$, we require $LW(\frac{s-1}{2}) - 1$ multiplications in $GF(p^m)$, and then multiply $x^{(s-1)/2}$ by x to get $x^{(s+1)/2}$, finally multiply $x^{(s+1)/2}$ and $x^{(s-1)/2}$ together to get x_0. It follows that the following multiplications are required for x_0:

$$\#M_m = [LW(\frac{s-1}{2}) - 1] + 2. \quad (6)$$

Note that when $s = 1$, we do not require any computations to get x_0. In other words, if $s = 1$ then $\#M_m = 0$.

If $x_0 = 1$, then we can assert that the input element x is a QR. If not, we need t multiplications to get $x_t = -1$ as shown in Fig.2. Note that the value of t depends on the input element x that can not be known in advance. Since $0 \leqslant t \leqslant T$, considering all the possibilities we can know that the QR test on average requires the following multiplications for $s > 1$:

$$\begin{aligned} \#\overline{M}_m &= LW(\frac{s-1}{2}) + 1 + \frac{1 \cdot 2^1 + 2 \cdot 2^2 + \cdots + T \cdot 2^T}{2^{T+1}} \\ &= LW(\frac{s-1}{2}) + 1 + (T-1) + 2^{-T}. \end{aligned} \quad (7)$$

On the other hand, $\#\overline{M}_m = (T-1) + 2^{-T}$ for $s = 1$. Comparing Eq.(5) and Eq.(7), we see that the proposed QR test is little faster than the conventional QR test.

3 Smart algorithm over $GF(p^m)$

Smart algorithm is well known as a conventional method to compute the square root. In this section, we first review Smart algorithm over $GF(p^m)$ and then estimate its complexity. Note that Smart algorithm over $GF(p^m)$ is considered under the condition of Eq.(2). Of course, we should execute the conventional QR test prior to Smart algorithm.

3.1 Smart algorithm

Smart algorithm

Input: A nonzero QR $x \in GF(p^m)$.

Output: A square root $\sqrt{x} \in GF(p^m)$.

Preparation:

1. Factorize $(p^m-1)/2$ as shown in Eq.(2).
2. Find an appropriate QNR $\theta \in GF(p^m)$ and compute $a = \theta^s$.

Procedure:

Step1 : Compute $b = x^{(s-1)/2}$ and set $t_0 = 0, k = 0$.

Step2 : Iteratively compute the following expressions as k increases by 0 to $T-1$:

$$t_{k+1} = t_k + 2^k c_k, \tag{8a}$$

where

$$c_k = \begin{cases} 0 & \text{if } \left((a^{t_k}b)^2 \cdot x\right)^{2^{T-1-k}} = 1 \\ 1 & \text{if } \left((a^{t_k}b)^2 \cdot x\right)^{2^{T-1-k}} = -1 \end{cases}. \tag{8b}$$

Step3 : Output the square root $\sqrt{x} = (a^{t_T}b)x$.

Since the finite fields $GF(p^m)$ are given before the SQRT computation, a, s and T can be prepared in advance. Note that when $T=0$, **Step2** is skipped and Smart algorithm still remains valid.

3.2 Complexity of Smart algorithm

The SQRT computation is performed in the given finite fields $GF(p^m)$, so the computational complexity of the preparation for Smart algorithm becomes unimportant. In what follows, we only evaluate computational complexity of the main procedure.

In **Step1**, we can use the binary method to compute the $(s-1)/2$-th power of x, which requires the following multiplications:

$$\#M_m = LW(\frac{s-1}{2}) - 1, \quad s > 1, \tag{9a}$$
$$\#M_m = 0, \quad s = 1. \tag{9b}$$

In **Step2**, $a^{t_k}b$ shown in Eq.(8b) have the form

$$b,\ a^{c_0}b,\ a^{c_0}a^{2c_1}b, \cdots,\ \Big(\prod_{i=0}^{T-1} a^{2^i c_i}\Big)b \tag{10}$$

for each $k = 0,\ 1,\ 2, \cdots,\ T$. Note that we only need to increase k up to $T-1$ in **Step2**; however, k is increased by 0 to T in Eq.(10). The reason is that the last expression of Eq.(10) i.e. $a^{t_T}b$ is required for **Step3**.

Since the first expression of Eq.(10) i.e. b does not need any computation, in what follows we only consider the remaining terms, whose computational complexity depends on the values of c_k for $k = 0,\ 1,\ 2, \cdots,\ T-1$. When every c_k is equal to 0, its computational complexity reaches the minimum; when every c_k is equal to 1, its computational complexity reaches the maximum. Since the values of c_k depend on the input element x that can not be predicted in advance, we consider all the possibilities of c_k and get the following average computational complexity of Eq.(10) (see A):

$$\#\overline{M}_m = \frac{(3T-4)2^{T-1}+2}{2^T} = \frac{3T-4}{2} + 2^{1-T}. \tag{11}$$

Next, for each $k = 0, 1, \cdots, T-1$ we compute the square of $a^{t_k}b$, and then multiply by x as shown in Eq.(8b). This process requires

$$\#M_m = 2T. \tag{12}$$

In addition to the above, we require the exponentiations to 2^{T-1-k}-th power for each k as shown in Eq.(8b) and finally multiply $a^{t_T}b$ by x in **Step3**; then this process requires

$$\#M_m = \sum_{i=1}^{T-1} i + 1 = \frac{T(T-1)}{2} + 1. \tag{13}$$

Based on Eqs.(9), (11), (12) and (13), Smart algorithm on average requires the following multiplications:

$$\#\overline{M}_m = LW(\frac{s-1}{2}) + \frac{T^2+6T-4}{2} + 2^{1-T}, s > 1, \tag{14a}$$
$$\#\overline{M}_m = \frac{T^2+6T-4}{2} + 2^{1-T} + 1, s = 1. \tag{14b}$$

4 MW-ST algorithm over $GF(p^m)$

In this section, we first present the proposed algorithm called the MW-ST algorithm over $GF(p^m)$ ($m=1$ or 2), and then evaluate its computational complexity in $GF(p)$ and $GF(p^2)$ respectively. Note that the MW-ST algorithm is considered under the condition of Eq.(2).

4.1 MW-ST algorithm

When $x_0 = 1$ in Fig.2(a) i.e. $x^s = 1$, multiplying its both sides by x and taking SQRT, we have $\sqrt{x} = x^{(s+1)/2}$. In what follows, we mainly consider the case of $x_0 \neq 1$. In this case, QR test is implemented as shown in Fig.2, where if $t < T$, as shown in Fig.2 (a), then x is a QR ; if not, as shown in Fig.2 (b), then x, denoted by c, is a QNR.

From Fig.2, it is clear that $c_T x_t = 1$. Moving to the windows of c_{T-1} and x_{t-1} shown in Fig.2, we define $\sigma(1) = c_{T-1}x_{t-1}$. Since c_{T-1} and x_{t-1} are the SQRTs of c_T and x_t respectively, and since the SQRT of $c_T x_t$ i.e. the SQRT of 1 must be 1 or -1 in any finite fields, we can assert that $\sigma(1) = \pm 1$. Then, we test the sign of $\sigma(1)$; if $\sigma(1) = 1$, moving to the windows of c_{T-2} and x_{t-2}, we define $\sigma(2) = c_{T-2}x_{t-2}$; if not,

multiplying $\sigma(1)$ by c_T to get $c_T c_{T-1} x_{t-1} = 1$ and then moving to the windows of c_{T-1}, c_{T-2} and x_{t-2}, we define $\sigma(2) = c_{T-1} c_{T-2} x_{t-2}$. This is the reason why the proposed algorithm is called the moving window-sign testing (MW-ST) algorithm.

In this way, we have

$$\sigma(k) = c_{T-1}^{i_{k-1}} c_{T-2}^{i_{k-2}} \cdots c_{T-k-1}^{i_1} c_{T-k} x_{t-k}, \qquad (15)$$

and same as $\sigma(1)$, we can also assert $\sigma(k) = \pm 1$. For $1 \leqslant k \leqslant t$, if $\sigma(k) = 1$ then $i_k = 0$ and if $\sigma(k) = -1$ then $i_k = 1$. When $k = t$, we have

$$\sigma(t) = c_{T-1}^{i_{t-1}} c_{T-2}^{i_{t-2}} \cdots c_{T-t-1}^{i_1} c_{T-t} x_0 = \pm 1. \qquad (16)$$

Since $x_0 = x^s$ and $c_T = -1$, Eq.(16) can be developed as

$$c_T^{i_t} c_{T-1}^{i_{t-1}} \cdots c_{T-t-1}^{i_1} c_{T-t} x^s = 1, \qquad (17)$$

where for $1 \leqslant k \leqslant t$, i_k have the same definition as in Eq.(15). Multiplying the both sides of Eq.(17) by x and taking SQRT, we can easily get

$$\sqrt{x} = c_{T-1}^{i_t} c_{T-2}^{i_{t-1}} \cdots c_{T-t-2}^{i_1} c_{T-t-1} x^{(s+1)/2}. \qquad (18)$$

We summarize the above method in what follows:

Moving window-sign testing algorithm over $GF(p^m)$ ($m = 1$ or 2)

Input: A nonzero QR $x \in GF(p^m)$.

Output: A square root $\sqrt{x} \in GF(p^m)$.

Preparation:

1. Factorize $(p^m - 1)/2$ as shown in Eq.(2).
2. Execute the proposed QR test as shown in Fig.2 (a) and store the intermediate values $x_0, x_1, \cdots, x_t$ and $x^{(s+1)/2}$.
3. Find an appropriate QNR $c \in GF(p^m)$, compute $c_0,\ c_1,\ \cdots, c_T$ as shown in Fig.2 (b) and store them in the memory.

Main Procedure:

1. Check whether x_0 is equal to 1 or not. If $x_0 = 1$, then output $\sqrt{x} = x^{(s+1)/2}$ and input another element. If not, execute **Procedure** 2 and **Procedure** 3 in order.
2. Let $\tau_0 = T-1$, $\mu = 1$ and $k = 1$, and then repeatedly execute **Step**$1 - 3$ until k becomes t.

Step 1 : Compute $\sigma = x_{t-k} \prod_{i=0}^{\mu-1} c_{\tau_i}$. If $\sigma = -1$, then $\tau_\mu = T - 1$ and $\mu = \mu + 1$. If not, the values of τ_μ and μ are not modified.

Step 2 : For $0 \leqslant i < \mu$, let $\tau_i = \tau_i - 1$.

Step 3 : Let $k = k + 1$.

3. Output the value of $\sqrt{x} = x^{(s+1)/2} \prod_{i=0}^{\mu-1} c_{\tau_i}$.

Remark:

- The register τ_i is for the index memory of the requried multiplicand c_q shown in Eq.(18) for $0 \leqslant q \leqslant T - 1$.
- μ shows the number of multiplicand c_q appearing in Eq.(18) for each k.

When $t = 0$, **Procedure2** is skipped and the MW-ST algorithm remains valid. Note that $x_0, x_1, \cdots, x_t$ and $x^{(s+1)/2}$ have been computed in the proposed QR test and saved in the memory. Also note that $c_0, c_1, \cdots, c_T$ have been prepared in advance. So we only need to call those data in the MW-ST algorithm, which makes the SQRT computation more efficient. We should also note that it is not necessary for the MW-ST algorithm to implement exponentiations to i_k-th power shown in Eq.(18), because i_k are equal to 1 or 0 for $1 \leqslant k \leqslant t$. For example, if $i_1 = 1$ in Eq.(18), then we save the value of c_{T-t-2} into the memory; if $i_1 = 0$, then we ignore the value of c_{T-t-2}.

As described above, when $m = 1$, of course all the computations required for the MW-ST algorithm are implemented in $GF(p)$. When $m = 2$, the factors of $(p-1)/2$, $(p-1)/2$ and $p+1$, can be factorized as

$$(p-1)/2 = 2^{T_1} s_1, \quad p + 1 = 2^{T_2} s_2, \qquad (19)$$

and from Eq.(2) it follows that $T = T_1 + T_2$ and $s = s_1 s_2$.

Since $x^{p+1} \in GF(p)$ [8] for any $x \in GF(p^2)$, it follows that $(x^{p+1})^{s_1}$ is always an element of $GF(p)$. From Eq.(19), we know

$$(x^{p+1})^{s_1} = (x_0)^{2^{T_2}} \quad \text{for } x_0 = x^s. \qquad (20)$$

We thus have $(x_0)^{2^{T_2}} \in GF(p)$ for any $x \in GF(p^2)$.

As described in Sec.1, p is a prime number, which implies $2 \,|\, p - 1$. It follows that there remain only two possibilities: $4 \nmid p - 1$ or $4 \,|\, p - 1$. Since only one of the four continuous integers $p - 1,\ p,\ p + 1$ and $p + 2$ can be divided by 4, it implies that

(i). $T_1 \equiv 0$ when $4 \nmid p - 1$;

Table 1: Computational Complexity and Running Time

		I. $p=2^{164}-2^6+1$				II. $p=2^{161}+2^{24}+1$				III. $p=2^{160}+2^3-1$			
		$q=2^{81}-2^6+1$				$q=2^{81}-2^{24}+1$				$q=2^{84}+2^2-1$			
Field	Method	A.Complexity			B.time	A.Complexity			B.time	A.Complexity			B.time
		$\#M_1$	$\#A_1$	S	[ms]	$\#M_1$	$\#A_1$	S	[ms]	$\#M_1$	$\#A_1$	S	[ms]
$GF(p)$	C.QR	319	0	0	50.1	161	0	0	25.6	161	0	0	25.7
	P.QR	318	0	0	50.1	160	0	0	25.3	161	0	0	26.5
	Smart	339	0	0	52.8	469	0	0	74.1	160	0	0	26.0
	MW-ST	8	0	0	1.2	138	0	0	22.1	0	0	0	0.1
$GF(q^2)$	C.QR	717	1434	239	120.4	717	1434	239	120.9	510	1020	170	93.2
	P.QR	714	1428	238	120.5	714	1428	238	120.3	507	1014	169	92.8
	Smart	798	1596	266	125.4	1716	3432	502	289.4	519	1038	173	94.6
	MW-ST	14	6	1	2.1	155	6	1	22.1	6	12	2	1.1

Remark 1: Using C++ with NTL on Pentium4 (2.6GHz) to get the running time.
Remark 2: C.QR and P.QR are the abbreviations of conventional QR test and proposed QR test, respectively.

(ii). $T_2 \equiv 1$ when $4 \,|\, p-1$.

When $4 \nmid p-1$, based on (i) and $(x_0)^{2^{T_2}} \in GF(p)$, we have $(x_0)^{2^T} \in GF(p)$ for any $x \in GF(p^2)$, which implies

$$x,\ x_0, \cdots, x_{t-1},\ c,\ c_0, \cdots, c_{T-1} \in GF(p^2). \tag{21}$$

When $4 \,|\, p-1$, based on (ii) and $(x_0)^{2^{T_2}} \in GF(p)$, we have $(x_0)^2 \in GF(p)$ for any $x \in GF(p^2)$, which implies

$$x_1, \cdots x_{t-1},\ c_1, \cdots, c_{T-1} \in GF(p). \tag{22}$$

This shows that in the MW-ST algorithm over $GF(p^2)$ $(4 \,|\, p-1)$ some computations can be executed in $GF(p)$.

4.2 Complexity of the MW-ST algorithm over prime fields $GF(p)$

From what described in Sec.4.1, it is easy to know that the computational complexity required for the MW-ST algorithm depends on the values of t and i_k for $k=1, 2, \cdots, t$ shown in Eq.(18): when $t=0$ and $i_k \equiv 0$, its computational complexity reaches the minimum; when $t = T-1$ and $i_k \equiv 1$, its computational complexity reaches the maximum. Since the values of t and i_k depend on the input element x that can not be predicted in advance, we consider all the possibilities of i_k and t to get the following average computational complexity (see B):

$$\#\overline{M}_1 = \frac{2^{T-2}(T^2+T)}{2^T} = \frac{T^2+T}{4}. \tag{23}$$

4.3 Complexity of MW-ST algorithm over extension fields $GF(p^2)$

As described in Sec.4.1, for $4 \nmid p-1$ and $4 \,|\, p-1$, we should evaluate the computational complexity of the MW-ST algorithm over $GF(p^2)$ respectively.

When $4 \nmid p-1$, we have Eq.(21), which implies the computational complexity of the MW-ST algorithm over $GF(p^2)$ is the same as that over $GF(p)$ described in Sec.4.2 except that all the computations are implemented in $GF(p^2)$. Therefore, the MW-ST algorithm over $GF(p^2)$ on average needs the following multiplications:

$$\#\overline{M}_2 = \frac{T^2+T}{4}. \tag{24}$$

From Eq.(22) and all the computations requried for the MW-ST algorithm described in Sec.4.1, we found that the MW-ST algorithm over $GF(p^2)$ $(4 \,|\, p-1)$ on average requires the following operations (see B):

$$\#\overline{M}_1 \ = \ \frac{2^{T-2}(T^2+T+6)-4}{2^T}$$

$$= \frac{T^2+T+6}{4} - 2^{2-T}, \quad (25a)$$

$$\#\overline{M}_2 = \frac{2^{T-1}}{2^T} = \frac{1}{2}. \quad (25b)$$

5 Simulation results and conclusion

In this section, Smart and the MW-ST algorithms over $GF(p^m)$ are implemented on a Pentium4 (2.6GHz) with C++ programing language by using NTL, where the characteristic p and the extension degree m of finite fields $GF(p^m)$ are assumed as follows:

$$p=2^{160}+2^3-1 \text{ and } m=1, \text{ where } 4\nmid p-1, \quad (26a)$$
$$p=2^{164}-2^6+1 \text{ and } m=1, \text{ where } 4\,|\,p-1, \quad (26b)$$
$$p=2^{161}+2^{24}+1 \text{ and } m=1, \text{ where } 4\,|\,p-1, \quad (26c)$$
$$p=2^{84}+2^2-1 \text{ and } m=2, \text{ where } 4\nmid p-1, \quad (26d)$$
$$p=2^{81}-2^6+1 \text{ and } m=2, \text{ where } 4\,|\,p-1, \quad (26e)$$
$$p=2^{81}-2^{24}+1 \text{ and } m=2, \text{ where } 4\,|\,p-1. \quad (26f)$$

Note that $GF(p^2)$ are constructed as Optimal Extension Field (OEF) [9] by adopting the following binomials as the modular polynomials:

$$x^2-5 \text{ for } p=2^{81}-2^6+1, \quad (27a)$$
$$x^2-3 \text{ for } p=2^{84}+2^2-1 \text{ and } 2^{81}-2^{24}+1. \quad (27b)$$

Based on Eqs.(2) and (26), we can get the values of T and s. Inputting T and s to Eqs.(5), (7), (14), (23), (24) and (25), we can evaluate the computational complexity of Smart and the MW-ST algorithms over $GF(p)$ and $GF(p^2)$ such as $\#M_1$, $\#M_2$. Since 1 multiplication in $GF(p^2)$ requires 3 multiplications in $GF(p)$, 6 additions in $GF(p)$ and 1 shift computation [9], we can thus get $\#A_1$ and $\#M_1$ shown in the columns A of Table 1.

Inputting $600,000$ QRs randomly generated, the running time for the two algorithms was measured on average in the columns B of Table 1. Table 1 clearly shows that the computational complexities of the MW-ST algorithm over $GF(p^2)$ are 10 times, 60 times and 90 times less than those of Smart algorithm over $GF(p^2)$ for $p=2^{81}-2^{24}+1$, $p=2^{84}+2^2-1$ and $p=2^{81}-2^6+1$, which makes that the proposed algorithm over $GF(p^2)$ is on average 10 times, 60 times and 90 times faster than Smart algorithm over $GF(p^2)$ for $p=2^{81}-2^{24}+1$, $p=2^{84}+2^2-1$ and $p=2^{81}-2^6+1$, respectively. In $GF(p)$, the proposed algorithm is on average 3 times, 40 times and 200 times faster than Smart algorithm for $p=2^{161}+2^{24}+1$, $p=2^{164}-2^6+1$ and $p=2^{160}+2^3-1$, respectively.

Since a QR test must be implemented prior to a SQRT computation, when counting the running time of a SQRT computation, we should thus consider the total running time of the QR test and the SQRT algorithm. From the columns B of Table 1, we can see that the sum of running time of the conventional QR test and Smart algorithm is 2 times more than that of the proposed QR test and the MW-ST algorithm.

From Eqs.(14), (23), (24) and (25), we can see that for the about same bits length p, if T (given by Eq.(2)) becomes larger, then the computational amounts of Smart and the MW-ST algorithms also become larger, which implies that both Smart algorithm and the MW-ST algorithm become slower. This can also be proved by the experiment results (comparing the column I and the column II of Table 1). In additi on, some computations required for the MW-ST algorithm over $GF(p^2)$ $(4\,|\,p-1)$ can be executed in $GF(p)$. So, the MW-ST algorithm over $GF(p^2)$ for $4\,|\,p-1$ is faster than that for $4\nmid p-1$, where p has the around same size in the both cases (comparing the column I and the column III of Table 1). However the ratio of the running time for the conventional QR test and Smart algorithm to that for the proposed QR test and the MW-ST algorithm is invariable. As is well known, OEFs are used to achieve fast finite field arithmetics. Therefore, to fast realize ECC, we had better choose those prime numbers p as the characteristic of OEFs such that $4\,|\,p-1$ and T is a small integer. Consequently we can conclude that the proposed QR test and the MW-ST algorithm will play a significant role in the implementation of ECC.

References

[1] Y. Nogami, A. Saito, and Y. Morikawa, IEICE Trans. Fundamentals, vol.E86-A, no 9, 2003, 2376 – 2387.

[2] Y. Nogami and Y. Morikawa, Memories of the Faculty of Engineering Okayama University, vol. 37, no. 2, 2003, 73 – 88.

[3] M. Yamamichi, M. Mambo, and H. Shizuya, IEICE Trans. Fundamentals, vol.E84-A, no 1, 2001, 140 – 145.

[4] F. Wang, Y. Nogami, and Y. Morikawa, Proc. of International Conference on Information and

Communications Security (ICICS2003), 2003, 1 − 10.

[5] http://www. ieee. org/groups/1363

[6] I. Blake, G. Seroussi, and N. Smart: Elliptic Curves in Cryptography, LMS 265, Cambridge University Press, 1999.

[7] D. V. Bailey, A Thesis submitted to the Faculty of the Worcester Polytechnic Institute in partial fulfillment of the requirements for the Degree of Master of Science in Computer Science, 2000.

[8] R. Lidl and H. Niederreiter: Encyclopedia of Mathematics and Its Applications Volume 20: Finite Fields, Cambridge University Press, 1997.

[9] D. V. Bailey and C. Paar, Proc. of 18th Annual International Cryptology Conference (Crypto1998), 1998, 472 − 485.

[10] D. E. Knuth: The Art of Computer Programming. 3rd ed. Addison-Wesley, vol. 2, 1997.

A Proof of Eq.(11)

For $T=5$, Eq.(10) without the first term has the form:

$$a^{c_0}b, a^{c_0}a^{2c_1}b, a^{c_0}a^{2c_1}a^{2^2c_2}b, \cdots, \Big(\prod_{i=0}^{4} a^{2^i c_i}\Big)b \quad \text{(A.1)}$$

To get Eq.(11), we first use this example to show the average computational complexity of Eq.(A.1).

Same as described in Sec.3, the complexity of Eq.(A.1) depends on the values of c_k; however the values of c_k lie on the input element x that can not be predicted in advance. Since c_k for $k=0,1,2,3,4$ must be 0 or 1, we can enumerate all the possibilities of $c_0c_1c_2c_3c_4$ as shown in Table 2, where "11001", for example, denotes $c_0=c_1=c_4=1$ and $c_2=c_3=0$. In the case of "11001", Eq.(A.1) has the form

$$ab,\ aa^2b,\ aa^2a^{2^4}b, \quad \text{(A.2)}$$

where 1 multiplication is required for ab and 2 multiplications are required for aa^2b and 4 multiplications are required for $aa^2a^{2^4}b$. Note that computing $aa^2a^{2^4}b$, we need the following multiplications:

$$a^2 \times a^2, a^{2^2} \times a^{2^2}, a^{2^3} \times a^{2^3}, a^{2^4} \times (ab), \quad \text{(A.3)}$$

where ab has been computed in the first term of Eq.(A.2) and a^2 has been computed in the second

Table 2: All the possibilities of Eq.(A.1)

pattern	$k=0$	$k=1$	$k=2$	$k=3$	$k=4$	$\#M$
00000	0	0	0	0	0	0
10000	1	0	0	0	0	1
01000	0	2	0	0	0	2
00100	0	0	3	0	0	3
00010	0	0	0	4	0	4
00001	0	0	0	0	5	5
10001	1	0	0	0	5	6
01001	0	2	0	0	4	6
00101	0	0	3	0	3	6
00011	0	0	0	4	2	6
10010	1	0	0	4	0	5
01010	0	2	0	3	0	5
00110	0	0	3	2	0	5
10100	1	0	3	0	0	4
01100	0	2	2	0	0	4
11000	1	2	0	0	0	3
11001	**1**	**2**	**0**	**0**	**4**	**7**
10101	1	0	3	0	3	7
10011	1	0	0	4	2	7
01101	0	2	2	0	3	7
01011	0	2	0	3	2	7
00111	0	0	3	2	2	7
11010	1	2	0	3	0	6
10110	1	0	3	2	0	6
01110	0	2	2	2	0	6
11100	1	2	2	0	0	5
11101	1	2	2	0	3	8
11011	1	2	0	3	2	8
10111	1	0	3	2	2	8
01111	0	2	2	2	2	8
11110	1	2	2	2	0	7
11111	1	2	2	2	2	9
Total complexity for all the possibilities						178

Table 3: Total Computations and Complexities Required for all the Events of MW-ST Algorithm over $GF(p^m)$ ($m = 1$ and 2)

T	t	Operations Required for Each Event
1	0	$x_0 = 1 \Rightarrow \sqrt{x} = x^{(s+1)/2}$
		$c_1 x_0 = 1 \Rightarrow \sqrt{x} = c_0 \cdot x^{(s+1)/2}$
2	0	$x_0 = 1 \Rightarrow \sqrt{x} = x^{(s+1)/2}$
		$c_2 x_0 = 1 \Rightarrow \sqrt{x} = c_1 \cdot x^{(s+1)/2}$
	1	$c_2 x_1 = 1 \Rightarrow c_1 \bullet x_0 = \begin{cases} 1 \Rightarrow \sqrt{x} = c_0 \cdot x^{(s+1)/2} \\ -1 \Rightarrow \sqrt{x} = c_1 \cdot c_0 \cdot x^{(s+1)/2} \end{cases}$
3	0	$x_0 = 1 \Rightarrow \sqrt{x} = x^{(s+1)/2}$
		$c_3 x_0 = 1 \Rightarrow \sqrt{x} = c_2 \cdot x^{(s+1)/2}$
	1	$c_3 x_1 = 1 \Rightarrow c_2 \bullet x_0 = \begin{cases} 1 \Rightarrow \sqrt{x} = c_1 \cdot x^{(s+1)/2} \\ -1 \Rightarrow \sqrt{x} = c_2 \cdot c_1 \cdot x^{(s+1)/2} \end{cases}$
	2	$c_3 x_2 = 1 \Rightarrow c_2 \odot x_1 = \begin{cases} 1 \Rightarrow c_1 \bullet c_0 = \begin{cases} 1 \Rightarrow \sqrt{x} = c_0 \cdot x^{(s+1)/2} \\ -1 \Rightarrow \sqrt{x} = c_2 \cdot c_0 \cdot x^{(s+1)/2} \end{cases} \\ -1 \Rightarrow c_2 \bullet c_1 \bullet x_0 = \begin{cases} 1 \Rightarrow \sqrt{x} = c_1 \cdot c_0 \cdot x^{(s+1)/2} \\ -1 \Rightarrow \sqrt{x} = c_2 \cdot c_1 \cdot c_0 \cdot x^{(s+1)/2} \end{cases} \end{cases}$

Remarks: $\odot$, $\bullet$ and $\cdot$ mean 4, 2 and 1 multiplications, respectively.

term of Eq.(A.2). It follows that 7 multiplications in $GF(p^m)$ are required for the event of "11001" printed in bold style in Table 2. In the same way, we can process all the events to get the data shown in Table 2. From Table 2, we know that Eq.(A.1) has 32 events and the total computational complexity is equal to 178; it follows that the average computational complexity is equal to 178/32.

In what follows, S_{T-1} denotes the total computational complexity for all the possibilities of Eq.(10), where T is given by Eq.(2). From Table 2, it follows, in particular, that S_4=178. In the same way, we can consider the cases of $T=1$, $T=2$ and so on. Same as S_4, we can thus know that $S_0=1$, $S_1=6$, $S_2=22$ and $S_3=66$, where we fortunately find that $S_1=2S_0+2^1(1+2)-2$, $S_2=2S_1+2^2(1+2)-2$ and $S_3=2S_2+2^3(1+2)-2$. Using mathematical induction, we can prove

$$S_n=2S_{n-1}+2^n(1+2)-2, \tag{A.4}$$

from which we can get

$$S_{T-1}=(3T-4)2^{T-1}+2, \tag{A.5}$$

where T is given by Eq.(2).

From what described above, we know that Eq.(10) has 2^T possibilities in total. In addition, choosing an input element at random from $GF(p^m)$, the probability that the input element belongs to each event is all the same. More detailedly, it is equal to $s/(2^T s)$ i.e. $1/2^T$. So, the average computational complexity of Eq.(10) can be expressed by Eq.(11).

B Proof of Eq.(23) and Eq.(25)

Based on the MW-ST algorithm over $GF(p^m)$ described in Sec.4.1, we can know which computations are required for the SQRT. In Table 3, we enumerate the required computations for $T = 1, 2, 3$, where T is defined by Eq.(2). From Table 3, we can easily get the column A of Table 4. As shown in Table 3, four events all require the multiplication $\odot$, two events both require the multiplication $\bullet$, and one event requires the multiplication $\cdot$. Therefore, when counting the numbers of the multiplications required for all the events,

Table 4: Total Complexity of the MW-ST Algorithm over $GF(p^m)$

T	A. The Number of Events	B. Total Complexity		
		$m=1$	$m=2$	
		$\#M_1$	$\#M_1$	$\#M_2$
1	2^1	1	0	1
2	2^2	6	8	2
3	2^3	24	32	4

$\odot$ means 4 multiplications, $\bullet$ means 2 multiplications and $\cdot$ means 1 multiplication.

In the MW-ST algorithm over $GF(p)$, let W_{T-1} denote the total complexity for all the events. From Table 3, it follows that $W_0=1$, $W_1=6$ and $W_2=24$, shown in the column B of Table 4, from which we fortunately find that $W_1=2W_0+2^1(0+2)$ and $W_2=2W_1+2^2(1+2)$. In the same way, we can consider the other cases of $T=4$, $T=5$ and so on. Using mathematical induction, we can prove

$$W_n=2W_{n-1}+2^n(n+1), \tag{A.6}$$

from which we can get the following computational complexity required for all the events of the MW-ST algorithm over $GF(p)$:

$$W_{T-1}=2^{T-2}(T^2+T). \tag{A.7}$$

From the column A of Table 4, we can assert that the computations required for the MW-ST algorithm have 2^T events in total. In addition, choosing an input element at random from $GF(p)$, the probability that the input element belongs to each event is all the same. More detailedly, it is equal to $s/(2^T s)$ i.e. $1/2^T$. Therefore, the average computational complexity of the MW-ST algorithm over $GF(p)$ can be expressed by Eq.(23).

In the MW-ST algorithm over $GF(p^2)$ $(4 \mid p-1)$, we require not only multiplications in $GF(p)$, but also multiplications in $GF(p^2)$. So, let U_{T-1} and V_{T-1} denote the total numbers of the required multiplications in $GF(p)$ and those in $GF(p^2)$, respectively.

We know if $a, b \in GF(p)$, then 1 multiplication in $GF(p)$ is required for ab, if $a \in GF(p)$ and $b \in GF(p^2)$, then 2 multiplications in $GF(p)$ are required for ab, and if $a, b \in GF(p^2)$, then 1 multiplication in $GF(p^2)$ is required for ab. From what described above and Table 3, we thus have $U_0=0$, $U_1=8$, $U_2=32$ and $V_0=1$, $V_1=2$, $V_2=4$, shown in the column B of Table 4, from which we fortunately find that $U_1=2U_0+2^1(0+2)+4$, $U_2=2U_1+2^2(1+2)+4$ and $V_1=2V_0$, $V_2=2V_1$. In the same way, we can consider the other cases of $T=4$, $T=5$ and so on. Using mathematical induction, we can prove

$$U_n=2U_{n-1}+2^n(n+1)+4, \tag{A.8a}$$

$$V_n=2V_{n-1}, \tag{A.8b}$$

from which we can get the following computational complexities required for all the events of the MW-ST algorithm over $GF(p^2)$:

$$U_{T-1}=2^{T-2}(T^2+T+6)-4, \tag{A.9a}$$

$$V_{T-1}=2^{T-1}. \tag{A.9b}$$

Since the computations required for the MW-ST algorithm have 2^T events in total and since the probability that an arbitrary input element belongs to each event is always equal to $1/2^T$, the average computational complexity of the MW-ST algorithm over $GF(p^2)$ $(4 \mid p-1)$ can be expressed by Eqs.(25).

C Fundamental operations in $GF(p^2)$

This appendix briefly shows how to implement the fundamental operations such as additions and multiplications in $GF(p^2)$ which is constructed as an OEF by adopting the modular polynomial x^2-3.

Let $\omega \in GF(p^2)$ be a root of the irreducible polynomial, it follows [2] that $\{1, \omega\}$ is the basis of $GF(p^2)$ over $GF(p)$. Using the basis, the arbitrary elements $A, B \in GF(p^2)$ can be uniquely represented as follows:

$$A = a_0+a_1\omega, \quad a_0, a_1 \in GF(p). \tag{A.10a}$$

$$B = b_0+b_1\omega, \quad b_0, b_1 \in GF(p). \tag{A.10b}$$

The sum and difference of A and B are given by

$$A \pm B = (a_0 \pm b_0)+(a_1 \pm b_1)\omega. \tag{A.11}$$

Using Karatsuba method [10], the product AB is given by

$$AB = (C_0+3C_1)+(C_2-C_1-C_0)\omega, \tag{A.12}$$

where $C_0=a_0b_0$, $C_1=a_1b_1$ and $C_2=(a_0+a_1)(b_0+b_1)$. Note that $3C_1=2C_1+C_1$, where $2C_1$ can be implemented by a shift computation. Therefore, 3 multiplications in $GF(p)$ and 6 additions in $GF(p)$ and a

shift computation are required for 1 multiplication in $GF(p^2)$.