Fast Montgomery-like Square Root Computation over $GF(2^m)$ for All Trinomials[☆]

Yin Li[a], Yu Zhang[a,*]

[a]Department of Computer Science and Technology, Xinyang Normal University, Henan, P.R.China

Abstract

This letter is concerned with an extension of square root computation over $GF(2^m)$ defined by irreducible trinomials. We introduce a new type of Montgomery-like square root formulae, which is more efficient compared with classic square root operation. By choosing proper Montgomery factor regarding to different types of trinomials, the space and time complexities of our proposal outperform or match the best results. Furthermore, a practical application of the Montgomery-like square root in exponentiation computation is also presented.

Keywords: Finite field, Square root, Exponentiation, Cryptography

1. Introduction

Arithmetic in finite field $GF(2^m)$ has many important practical applications such as public key cryptography and coding theory [1, 2]. These applications usually require high efficient implementations of such arithmetic operations, e.g., field addition, multiplication, inversion and squaring. Particularly, square root operation has attracted many attentions during recent years as it has become an important building block in the design of some elliptic curve primitives [4, 5]. In addition, since the square root is somewhat analogous to squaring, it has been applied in some parallel multiplicative inversion or exponentiation algorithms [6, 8], where both squaring and square root are utilizing as main building blocks.

Generally speaking, the efficiency of hardware implementation of field arithmetic is evaluated by space and time complexity. The former is expressed as the number of logic gates (XOR and AND) and the latter is expressed as the total gates delay of the circuit, denoted by T_A and T_X, respectively.

Consider an irreducible polynomial $f(x)$ over $\mathbb{F}_2$. Then, a binary extension field can be defined as $GF(2^m) \cong \mathbb{F}_2[x]/(f(x))$. Let A be an arbitrary element in the field $GF(2^m)$. The field square root computation of A, denoted as $\sqrt{A}$ or $A^{1/2}$, is to find $D \in GF(2^m)$ such that $D^2 = A$. Note that square root operation is simply a circular operation under normal base (NB) representation. Hence, most works for efficient square root computation are based on polynomial representation (PB).

Obviously, a straightforward approach for square root computation is based on Fermat's Little Theorem [3], which requires $m-1$ squarings in all. Fong [7] proposed more efficient algorithms by utilizing the pre-computation value $x^{1/2}$. However, this algorithm requires one $m/2$ bits field multiplication by a precomputed constant $x^{1/2}$, which is still an expensive operation. In [8], Rodríguez-Henríquez et al. proposed an alternative method using the inversion of the multiplicative matrix, which is constructed for squaring operation. Their approach theoretically can be applied to any type of generating polynomials. When the generating polynomial $f(x)$ is a trinomial, they derived the explicit formulae for square root computation. It is shown that such square root operations can be implemented with no more logic gates than those associated with squaring of the same field. Based upon this assertion, the authors [8] also proposed a parallel exponentiation that performs squaring based and square root based exponentiation algorithms, simultaneously. Therefore, such an algorithm could achieve twice implementation efficiency compared with classic squaring based exponentiation.

Rodríguez-Henríquez et al. approach is efficient. Nevertheless, for certain type of irreducible trinomials,

[☆]This work is supported by the Natural Science Foundation of China (No.61402393, 61601396).

[*]Corresponding author

Email addresses: yunfeiyangli@gmail.com (Yin Li), willow1223@126.com (Yu Zhang)

the circuit delay of the square root and squaring is different. For example, if $f(x) = x^m + x^k + 1$ with odd m and even k, the squaring costs T_X delay while square root costs $3T_X$. In this case, if these two operations are applied to perform parallel exponentiation (inversion), the slower one will influence the performance of the whole algorithm. In this contribution, we consider a new type of square root operator by adding a specific Montgomery-like factor. Explicit formulae of such operations are presented when $GF(2^m)$ is generated with an irreducible trinomial. As a result, it is shown that these formulae are quite simple and have smaller space and time complexities in parallel implementation. Coincidentally, this type of Montgomery-like square root operator has the same circuit delay compared with Montgomery squaring [9] that defined in the same finite field. Therefore, we then describe an improved parallel exponentiation algorithm that using Montgomery squaring and Montgomery-like square root operation.

The rest of this paper is organized as follows: in Section 2, we introduce the definition of the Montgomery-like square root and some other notations. Then, explicit formulae for this type of square root operation are given in section 2. In addition, the space and time complexities are also evaluated. Section 3 describes a parallel exponentiation algorithm based on Montgomery squaring and Montgomery-like square root operators. Finally, some conclusions are drawn in Section 4.

2. Montgomery-like Square Root

Let $A = \sum_{i=0}^{m-1} a_i x^i$ be an arbitrary element of $GF(2^m)^*$ using PB representation. To compute the square root of A, we first partition A into two parts according to the degree parity of its intermediate, i.e.,

$$A = A_{even}^2 + xA_{odd}^2,$$

where

$$\begin{cases} A_{even} = \sum_{i=0}^{(m-1)/2} a_{2i} x^i,\ A_{odd} = \sum_{i=0}^{(m-3)/2} a_{2i+1} x^i, (m \text{ is odd}), \\ A_{even} = \sum_{i=0}^{m/2-1} a_{2i} x^i,\ A_{odd} = \sum_{i=0}^{m/2-1} a_{2i+1} x^i, (m \text{ is even}). \end{cases}$$

Therefore, the square root of A is given by

$$A^{\frac{1}{2}} = A_{even} + x^{\frac{1}{2}} A_{odd}. \tag{1}$$

It is obvious that the computation of $x^{1/2}$ is crucial to the square root computation. Actually, the element $x^{1/2}$ is a constant that can be pre-computed. Thus, the square root operation can be implemented as performing a field multiplication between A_{odd} and $x^{1/2}$ and then adding with A_{even} [7]. However, this method requires a constant multiplication, which is still an expensive operation. Inspired by Montgomery squaring operation, we multiply the square root by a proper factor and introduce a Montgomery-like square root operation. Such a factor can help us simplify the calculation of $x^{1/2}$ and avoid previous complex operation, which can accelerate the implementation of the Montgomery-like square root operation.

We first give the definition of this new type of square root operation.

Definition 1. *Let A be an arbitrary element and ω be a fixed element of $GF(2^m)^*$, respectively. Then the Montgomery-like square root of A is defined as $A^{1/2} \cdot \omega$, while ω is named as the Montgomery-like factor.*

Let D be the result of $A^{1/2} \cdot \omega$. When $GF(2^m)$ is defined by an irreducible trinomial $f(x) = x^m + x^k + 1$, we investigate the choice of ω and the explicit formulae with respect to D. Note that here $x = x^m + x^k$ and

$$x^{\frac{1}{2}} = x^{\frac{m}{2}} + x^{\frac{k}{2}}.$$

It follows that the optimal ω relies on the computation of $x^{1/2}$, which varies according to the parities of m and k. Thus, four cases are considered:

Case 1: m even, k odd. In this case, we know that $2 \mid m$, but $2 \nmid k$. Let $n = m/2$, then we have

$$\begin{aligned} & x^{\frac{1}{2}} = x^n \cdot x^{\frac{1}{2}} + x^{\frac{k+1}{2}} \\ \Rightarrow\quad & x^{\frac{k+1}{2}} = (1 + x^n) \cdot x^{\frac{1}{2}}, \\ \Rightarrow\quad & x^{\frac{1}{2}} = (1 + x^n) \cdot x^{-\frac{k-1}{2}}. \end{aligned}$$

Plug the above expression to (1), one can check that

$$A^{\frac{1}{2}} = A_{even} + x^{\frac{1}{2}} A_{odd} = A_{even} + (A_{odd} + A_{odd} x^n) \cdot x^{-\frac{k-1}{2}}.$$

Note that A_{odd} consists of n terms, thus, there is no overlap between A_{odd} and $A_{odd}x^n$. Let ω be $x^{(k-1)/2}$, the Montgomery-like square root of this case is

$$A^{\frac{1}{2}} \cdot \omega = A^{\frac{1}{2}} \cdot x^{\frac{k-1}{2}} = A_{even} x^{\frac{k-1}{2}} + (A_{odd} + A_{odd} x^n).$$

Since the degree of A_{even} and A_{odd} are $n-1$, $\deg(A_{even}x^{(k-1)/2}) = n - 1 + (k-1)/2 \le n + (m-1)/2 = m - 1$ and $\deg(A_{odd} + A_{odd}x^n) = n + n - 1 = m - 1$. Therefore, no further reduction is needed in above expression. At this time, $D = \sum_{i=0}^{m-1} d_i x^i = A^{1/2} \cdot x^{(k-1)/2}$. Then,

$$d_i = \begin{cases} a_{2i+1}, & 0 \le i \le \frac{k-3}{2}, \\ a_{2i-k+1} + a_{2i+1}, & \frac{k-1}{2} \le i \le n-1, \\ a_{2i-k+1} + a_{2i+1-m}, & n \le i \le n-1+\frac{k-1}{2}, \\ a_{2i+1-m}, & n + \frac{k-1}{2} \le i \le m-1, \end{cases} \tag{2}$$

where $0 < k \leq m-1$. In addition, if $k = m-1$, (2) can be simplified further:

$$d_i = \begin{cases} a_{2i+1}, & 0 \leq i \leq n-1, \\ a_{2i-m} + a_{2i+1-m}, & n \leq i \leq m-1 \end{cases} \quad (3)$$

Case 2: m even, $k = m/2$ odd. This case is actually a special case of Case 1. It is easy to check that $x^{1/2} = (1 + x^k) \cdot x^{-(k-1)/2}$. Also notice that $m = 2k$ and $x^m = x^k + 1$. Hence, the preceding expression can be rewritten as $x^{1/2} = x^{2k-(k-1)/2}$. So,

$$A^{\frac{1}{2}} = A_{even} + A_{odd} x^{k+\frac{k+1}{2}}.$$

Notice that there are at most $(k+1)/2$ terms of which degrees are out of the range $[0, m-1]$ and the reduction is relatively simple. Thus, let $\omega = 1$ and $D = \sum_{i=0}^{m-1} d_i x^i = A^{1/2}$, we have

$$d_i = \begin{cases} a_{2i} + a_{2i+k}, & 0 \leq i \leq \frac{k-1}{2}, \\ a_{2i}, & \frac{k+1}{2} \leq i \leq k-1 \\ a_{(2i+k) \bmod m}, & k \leq i \leq m-1. \end{cases} \quad (4)$$

This formula simply coincides with the result presented in [8].

Case 3: both m and k are odd. Let $n = \frac{m+1}{2}$, then

$$x^{\frac{1}{2}} = x^n + x^{\frac{k+1}{2}}.$$

One can check that the above formula is very easy and the corresponding square root computation needs no ω to simplify its computation. Analogous with Case 2, let $\omega = 1$ and $D = \sum_{i=0}^{m-1} d_i x^i = A^{\frac{1}{2}}$, then

$$d_i = \begin{cases} a_{2i}, & 0 \leq i \leq \frac{k-1}{2}, \\ a_{2i} + a_{2i-k}, & \frac{k+1}{2} \leq i \leq \frac{m-1}{2}, \\ a_{2i-k}, & k \leq i \leq m-1. \end{cases} \quad (5)$$

Case 4: m odd, k even. Let $n = \frac{m-1}{2}$, then

$$\begin{aligned} & x^{\frac{1}{2}} = x^{n+1} + x^{\frac{k}{2}} \cdot x^{\frac{1}{2}} \\ \Rightarrow \quad & x^{n+1} = (1 + x^{\frac{k}{2}}) \cdot x^{\frac{1}{2}}, \\ \Rightarrow \quad & x^{-\frac{1}{2}} = (1 + x^{\frac{k}{2}}) \cdot x^{-(n+1)}, \\ \Rightarrow \quad & x^{\frac{1}{2}} = (1 + x^{\frac{k}{2}}) \cdot x^{-n}. \end{aligned}$$

Plug the above expression to (1), then

$$\begin{aligned} A^{\frac{1}{2}} &= A_{even} + x^{\frac{1}{2}} A_{odd} \\ &= A_{even} + (A_{odd} + A_{odd} x^{\frac{k}{2}}) \cdot x^{-n} \end{aligned}$$

Let ω be x^n, then the Montgomery-like square root is

$$A^{\frac{1}{2}} \cdot \omega = A^{\frac{1}{2}} x^n = A_{even} x^n + A_{odd} + A_{odd} x^{\frac{k}{2}}.$$

Please note that A_{odd} consists of n terms and A_{even} consists of $n+1$ terms. Thus, $A_{even} x^n$ and A_{odd} are non-overlapping with each other. Also notice that $\deg A_{odd} = n-1$, $\deg A_{even} = n$ and $k < m$, so $\deg A_{odd} x^{\frac{k}{2}} = n - 1 + k/2 \leq n + (m-1)/2 = m - 1$, $\deg A_{even} x^n = 2n = m - 1$. Therefore, no further reduction is required by $A^{1/2} x^n$. Let $D = \sum_{i=0}^{m-1} d_i x^i = A^{\frac{1}{2}} \cdot x^n$, then

$$d_i = \begin{cases} a_{2i+1}, & 0 \leq i \leq \frac{k}{2} - 1, \\ a_{2i+1} + a_{2i+1-k}, & \frac{k}{2} \leq i \leq n-1, \\ a_{2i+1-k} + a_{2i-m+1}, & n \leq i \leq n + \frac{k}{2} - 1, \\ a_{2i-m+1}, & n + \frac{k}{2} \leq i \leq m-1. \end{cases} \quad (6)$$

where $0 < k < m-1$. If $k = m-1$, then

$$d_i = \begin{cases} a_{2i+1}, & 0 \leq i \leq n-1, \\ a_{2i-m+1} + a_{2i-m+2}, & n \leq i \leq m-2, \\ a_{2i-m+1}, & i = m-1. \end{cases} \quad (7)$$

According to expression (2) to (7), we summarize the space and time complexities of the Montgomery-like square root in Table 1. As a comparison, the space and time complexities of ordinary square root operation [8] are also given. Specially, we also give the complexities for Montgomery squaring $A^2 x^{-k} \bmod x^m + x^k + 1$ for $1 \leq k \leq m/2$, which had been investigated in [9].

It is clear that, by choosing proper factor ω, the Montgomery-like square root costs at most $m/2$ (or $(m-1)/2$) XOR gates with only 1 T_X gate delay, which outperforms or matches the best square root computation algorithms [8]. Moreover, the proposed new type of square root coincidentally has the identical circuit delay compared with Montgomery squaring, while their space complexities are almost the same.

Example: Consider a finite field $GF(2^9)$ defined by $x^9 + x^4 + 1$. According to previous description, $x^9 + x^4 + 1$ satisfies case 4, we have $\omega = x^{(9-1)/2} = x^4$. Given an arbitrary element $A = \sum_{i=0}^{8} a_i x^i$, the Montgomery-like square root of A is $A^{1/2} \cdot x^4 = \sum_{i=0}^{8} d_i x^i$, where

$$\begin{aligned} & d_0 = a_1, && d_3 = a_7 + a_3, && d_6 = a_4, \\ & d_1 = a_3, && d_4 = a_0 + a_5, && d_7 = a_6, \\ & d_2 = a_5 + a_1, && d_5 = a_2 + a_7, && d_8 = a_8. \end{aligned}$$

The computation of above coefficients d_i requires 4 X-OR gates and T_X in parallel.

3. Exponentiation based on Montgomery squaring and square root

In [8], the authors proposed a parallel exponentiation algorithm over $GF(2^m)$. Their algorithm comprises two sub-exponentiations that based on squaring and square

Table 1: Space and time complexities for different square root operations and squaring operations

	Montgomery square root			Ordinary[8]		Montgomery squaring [9]		Squaring [8]	
Cases	ω	$\#XOR$	Delay	$\#XOR$	Delay	$\#XOR$	Delay	$\#XOR$	Delay
1. m even, k odd	$x^{\frac{k-1}{2}}$	$\frac{m}{2}$	T_X	$\frac{m+k-1}{2}$	$2T_X$	$\frac{m}{2}$	T_X	$\frac{m+k-1}{2}$	$2T_X$
2. m even, $k=\frac{m}{2}$ odd	1	$\frac{m+2}{4}$	T_X	$\frac{m+2}{4}$	T_X	$\frac{m}{2}$	T_X	$\frac{m+2}{4}$	T_X
3. m, k odd	1	$\frac{m-1}{2}$	T_X	$\frac{m-1}{2}$	T_X	$\frac{m-1}{2}$	T_X	$\frac{m-1}{2}$	$2T_X$
4. m odd, k even	$x^{\frac{m-1}{2}}$	$\frac{m-1}{2}$	T_X	$\frac{m+k-1}{2}$	$3T_X$	$\frac{m-1}{2}$	T_X	$\frac{m+k-1}{2}$	T_X

root operation, respectively. For square root based exponentiation, it mainly utilized the equation

$$A^{2^{m-i}} = A^{2^{-i}}, i = 1, 2, \cdots, m-1,$$

where $A \in GF(2^m)$ is an arbitrary nonzero element. The above equation can be easily proved using Fermat Little Theorem [3]. Let $e = (e_{m-1}, \cdots, e_1, e_0)_2$ be a m-bit nonzero integer. By substituting half the squaring operations with square root operations, the exponentiation A^e can be written as

$$\begin{aligned} A^e &= \prod_{i=0}^{m-1} A^{2^i e_i} = \prod_{i=n}^{m-1} A^{2^i e_i} \cdot \prod_{i=0}^{n-1} A^{2^i e_i} \\ &= \prod_{i=n}^{m-1} A^{2^{-(m-i)} e_i} \cdot \prod_{i=0}^{n-1} A^{2^i e_i}. \end{aligned}$$

where $0 < n < m$. In [8], the parameter n is chosen as $\lfloor \frac{m}{2} \rfloor$. One can check that the two sub-exponentiations presented in above expression can be performed simultaneously, thus, the whole algorithm achieved better performance compared with the classic one. As shown in Table 1, we note that the Montgomery-like square root operator and Montgomery squaring are usually faster than ordinary square root and squaring operations. Algorithm 1 describes an improved exponentiation algorithm for A^e, which utilized Montgomery squaring and square root operator.

Algorithm 1 Exponentiation based on Montgomery squaring and Montgomery-like Square root

Input: $A \in GF(2^m), f(x), e = (e_{m-1}, e_{m-2}, \cdots, e_1, e_0)_2$
Output: $B = A^e \bmod f(x)$

1: $B = C = 1$;
2: $e_m = 0$;
3: $n = \lceil \frac{m}{2} \rceil$;
4: **for** $i = n-1$ **down to** 0 **do**
5: $\quad B = B^2 \cdot x^{-k}$;
6: $\quad$ **if** $e_i == 1$ **then**
7: $\quad\quad B = B \cdot A \bmod f(x)$;
8: $\quad$ **end if**
9: **end for**
10: $B = B \cdot C$;
11: $B = B \cdot \omega^*$;
12: **return** B;

for $j = n$ **to** m **do**
$\quad C = C^{\frac{1}{2}} \cdot \omega$;
$\quad$ **if** $e_j == 1$ **then**
$\quad\quad C = C \cdot A \bmod f(x)$;
$\quad$ **end if**
end for

Description: The two procedures presented in steps 4-9 are running in parallel. Also notice that the choice of n in step 3 is slightly different from [8], as we found that this selection can make the number of squaring and square root almost equal, which can save the whole algorithm delay. Since we utilize the Montgomery squaring and Montgomery-like square root operator instead of the original ones, ω^* in step 11 is a compensatory parameter that used to correct the final exponentiation. The form of ω^* is given in following proposition.

Proposition 1. *The compensatory parameter ω^* in Algorithm 1 can be obtained as*

$$\omega^* = \omega^{2^n-2} \cdot (x^k)^{2^n-1}. \tag{8}$$

Proof. According to steps 4-7 of Algorithm 1, it totally perform $m - n + 1$ Montgomery-like square root operations and n Montgomery squarings. Each time we execute the body in the loops, there are one x^{-k} (for squaring) and one ω (for square root) multiplying the results. In the end, it follows that the extra parameter is

$$\begin{aligned} &\omega \cdot \omega^{\frac{1}{2}} \cdot \omega^{\frac{1}{4}} \cdots \omega^{\frac{1}{2^{m-n}}} \cdot x^{-k} \cdot (x^{-k})^2 \cdots (x^{-k})^{2^{n-1}} \\ &= \omega^{2-2^{-(m-n)}} \cdot (x^{-k})^{2^n-1} = (\omega)^{2-2^n} \cdot (x^k)^{1-2^n} \\ &= \omega^{2-2^n} \cdot (x^k)^{1-2^n} \end{aligned} \tag{9}$$

Since ω^* is the compensatory parameter that used to correct the final result, then we know that (9) is equal to the inversion of ω^*. Then, the result is direct. □

According to proposition, the explicit formula of ω^* depends on the form of ω. For example, when m, k satisfy the condition of case 1, we have $\omega = x^{(k-1)/2}$ and $n = m/2$. Plug these formulae into expression (8), we have $\omega^* = x^{(k-1)(2^{n-1}-1)+k(2^n-1)}$. The explicit formulae for ω^* of other cases are summarized in Table 2. Furthermore, the constant multiplication $B \cdot \omega^*$ can be performed using Mastrovito approach. Since ω^* is constant, the corresponding product matrix is fixed, no AND gate is needed. Only certain XOR gates are required by corresponding matrix-vector multiplication,

Table 2: Explicit formulae of ω^*

Cases	ω^*
1. m even, k odd	$x^{(k-1)(2^{n-1}-1)+k(2^n-1)}$
2. m even, $k=\frac{m}{2}$ odd	$x^{k(2^n-1)}$
3. m, k odd	$x^{k(2^n-1)}$
4. m odd, k even	$x^{(m-1)(2^{n-1}-1)+k(2^n-1)}$

which leads to no more than $m^2 - m$ XOR gates with at most $\lceil \log_2 m \rceil$ XOR gate delay.

Then we give a comparative analysis between the original exponentiation algorithm [8] and ours. On the one hand, both their algorithm and ours cost $W(e)$ multiplications, $m-n+1$ square root and n squaring operations (classic or Montgomery), where $W(\cdot)$ is the Hamming weight of the binary representation of e. In addition, our algorithm requires one more extra constant multiplication. Assume that the number of bit operations for a multiplication and a constant multiplication are $2m^2 - 1$ and $m^2 - m$ [1], respectively. We compare the bit operations of the two algorithms in the following table. As shown in the table, our algorithm roughly

Table 3: The bit operations for different exponentiation

Algorithms	Bit operations
[8]	$(2^m - 1) \cdot W(e) + O(m^2/2)$
This paper	$(2^m - 1) \cdot W(e) + O(3m^2/2)$

costs m^2 more bit-operations than that in [8]. In fact, we have checked the space complexity of $B \cdot \omega^*$ in $GF(2^m)$ defined by all the trinomials of degree $m \in [100, 1023]$ with cryptographic interests. It is argued that $B \cdot \omega^*$ only costs less than $m^2/2$ XOR gates in roughly two thirds of such finite fields. In this case, our algorithm only costs about $m^2/2$ more bit-operations.

On the other hand, according to Table 1, it is clear that the gate delays for the ordinary squaring and square root are identical only when m, k satisfy case 2. Note that the circuit delay for parallel implementation of ordinary squaring and square root is in fact equal to that of the slower operation. Thus, in other cases, it requires at least $2T_X$. Conversely, our algorithm can save at least one T_X for each squaring (square root) computation in the loop. Note that the constant multiplication presented in step 11 requires at most $\lceil \log_2 m \rceil$ extra XOR gate delays. We finally save at least $(\lceil \frac{m}{2} \rceil - \lceil \log_2 m \rceil)T_X$ circuit delay except case 2. In case 2, the algorithm in [8] is preferred to ours, as it has smaller space and time complexities. Notwithstanding, in other cases, our algorithm takes slightly more space but less time.

[1] The constant multiplication needs no AND gates for bitwise multiplication, as the product matrix is fixed.

4. Conclusion

In this paper, we have proposed a new type of Montgomery-like square root operation. By choosing a proper Montgomery factor, the proposed scheme has only one T_X delay and its space complexity is at least as good as the best result. As an important application, we show that this type of square root combined with Montgomery squaring can speed up parallel exponentiation algorithm that based on ordinary squaring and square root.

References

[1] R. Lidl and H. Niederreiter, Introduction to finite fields and their applications. Cambridge University Press, New York, NY, USA, 1994.

[2] R. Lidl and H. Niederreiter, Finite Fields, Cambridge University Press, New York, NY, USA, 1996.

[3] J.V.Z. Gathen and J. Gerhard, Modern Computer Algebra (3rd ed.). Cambridge University Press, New York, NY, USA, 2013.

[4] D. Hankerson, A. Menezes and S. Vanstone, Guide to Elliptic Cryptography, Springer-Verlag, 2004.

[5] R. Schroeppel, C. Beaver, R. Gonzales, R. Miller and T. Draelos, A Low-Power Design for an Elliptic Curve Digital Signature Chip, Proc. Fourth Int'l Workshop Cryptographic Hardware and Embedded Systems, 2002, pp. 366-380.

[6] F. Rodríguez-Henríquez, G. Morales-Luna, N. Saqib,and N. CruzCortés, Parallel Itoh-Tsujii Multiplicative Inversion Algorithm for a Special Class of Trinomials, Reconfigurable Computing: Architectures, Tools and Applications, LNCS 4419, 2007, pp. 226-237.

[7] K. Fong, D. Hankerson, J. López, and A. Menezes, Field Inversion and Point Halving Revisited, IEEE Trans. Computers, 2004, 53, (8), pp. 1047-1059.

[8] F. Rodríguez-Henríquez, G. Morales-Luna and J. Lópz, Low-Complexity Bit-Parallel Square Root Computation over $GF(2^m)$ for All Trinomials, IEEE Transactions on Computers, 2008, 57, (4), pp. 1-9.

[9] H. Wu, Montgomery multiplier and squarer for a class of finite fields, IEEE Transactions on Computers, 2002, 51, (5), pp. 521-529.

[10] Recommended Elliptic Curves for Federal Government Use, special publication, Nat'l Inst. Standards and Technology, http://csrc.nist.gov/csrc/fedstandards.html, July 1999.