A method to compute squares and square roots in $GF(2^n)$

Brian King
briking@iupui.edu

Abstract

Here we provide a new method for computing squares in $GF(2^n)$. The basis of this method is to exploit the fact that element in $GF(2^n)$ can be represented as a sum of its odd and even parts. This method has provided an insight into how to efficiently compute a square root in $GF(2^n)$.

1 Introduction

Binary fields denoted by $GF(2^n)$ provide a suitable setting for numerous applications within a communication system. The field $GF(2^n)$ is constructed by selecting an irreducible polynomial p of degree n in $\mathbf{Z}_2[x]$, then the field is $GF(2^n) = \mathbf{Z}_2[x]/p(x)$. Our interest in binary field is the application of public-key cryptography, in particular elliptic curve cryptography (or ECC).

Public-key cryptography is an essential ingredient in today's e-commerce applications (as well as several other applications). Two important applications of public-key cryptography are digital signatures and key derivation. Elliptic curve cryptography has several advantages over integer based cryptosystem. Some of these advantages include faster signature generation and the smaller bandwidth. A typical private key required for an RSA signature (an example of an integer-based cryptosystem) is approximately 1024 bits, whereas a comparable key (in terms of security) for an elliptic curve cryptosystem is approximately 160 bits. Consequently manufacturers of lightweight devices like PDA's, cell-phone, etc. are considering utilizing ECC for their ecommerce applications. In addition, if and/or when signature generation is required on such devices, ECC will become even more attractive.

The most practical application of using the field $GF(2^n)$ in public-key cryptosystem is ECC, however it is possible to adapt the discrete-log based public-key cryptographic primitives that have been traditionally defined over prime fields to the setting $GF(2^n)$. There will be a price to pay in the sense that the private key for a discrete-log cryptosystem in $GF(2^n)$ will need to be larger than the private key of the same primitive defined over a prime field [9].

In [5, 7], Koblitz and Miller independently proposed to use elliptic curves over a finite field to implement cryptographic primitives. As we have stated there may be significant advantages to using elliptic curve cryptography. In practice an elliptic curve, as well as the underlying finite field, may be fixed by some standard, for example the WAP forum [15] (wireless application protocol, i.e. wireless internet), has standardized the elliptic curves to be used in their WTLS standard (wireless version of SSL), as well as the underlying field. A popular choice for the underlying field is to use a binary field $GF(2^n)$. Typically the generating polynomial is either a trinomial or a pentanomial. There are multiple ways to represent elements in the field $GF(2^n)$, for example one could use a polynomial basis or a normal basis. All our work is based on the assumption that elements are represented using a polynomial basis. This assumption is natural, most implementations in industry will use a polynomial basis.

The necessary elliptic curve computation is to compute the multiple of a point, that is compute kP (where $P = (x, y)$ belongs to the curve). This computation is analogous to the "square and multiply" method used in $\mathbf{Z}_N$ to compute an element to a power. The two basic group operations on an elliptic curve are the *double* and the *add*. The required field operations needed to perform these group operations are:

field multiplication, field add, field square, and field inverse. The frequency of each of these operations vary dependent on the representation of a point of the elliptic curve. Suffice to say that the square is an integral field operation and in some point representations, several squares will be needed to compute a single *double* and/or *add*.

2 Background

2.1 Squaring in a binary field

If one uses a normal basis to represent elements in the field then a square is a circular shift. However in practice one usually will use a polynomial basis to represent a field element. A square can be computed as a field multiply, but this does not take advantage that both operands are identical. In general, one should expect that a software implementation of a square will take approximately one-eighth of a multiply.

The following approach to computing a square has been discussed extensively in literature, for example [3, 10]. If $\mu_k x^k + \mu_{k-1} x^{k-1} + \cdots + \mu_1 x + \mu_0$ is a field element then $\mu_k x^{2k} + \mu_{k-1} x^{2(k-1)} + \cdots + \mu_1 x^2 + \mu_0$ is its square. This latter term needs to be reduced (it's degree most likely will exceed or equal the degree of the generating polynomial). Consequently using this approach, from an implementation perspective, one needs to insert zeros between terms and then perform a modular reduction.

In [13], Wu has discussed in detail the precise cost of computing a square in $GF(2^n)$ and how to perform the square. The algorithms were discussed at the bitwise level, such a discussion is not always useful when developing an implementation in software. The focus of this work is such that our algorithm is very software friendly. In terms of the number of additions, our squaring algorithm is optimal for the cases we discuss.

2.2 Odd/Even polynomials

If $\mu \in GF(2^n)$, then $\mu = \mu_{n-1} x^{n-1} + \cdots + \mu_1 x + \mu_0$. Using the definition of odd and even functions, we can represent μ as a sum of its odd terms plus its even terms. For example if n is odd then $\mu = \mathcal{O} + \mathcal{E}$ where $\mathcal{E} = \mu_{2\frac{n-1}{2}} x^{2\frac{n-1}{2}} + \cdots + \mu_2 x^2 + \mu_0$ and $\mathcal{O} = \mu_{2\frac{n-1}{2}-1} x^{2\frac{n-1}{2}-1} + \cdots + \mu_3 x^3 + \mu_1 x$. Observe that when n is odd then the length of the even part will exceed the length of the odd part by 1. Otherwise the lengths are the same.

Recall that the sum/difference of an odd (even) function is an odd (even) function, and the product of an even (odd) function with an odd function is even (odd). The product of two odd functions are even and the product of two even functions are even. Lastly, notice that the multiplication of μ by x^J is analogous to performing a right shift of μ. Of course the resulting product may have degree $\geq n$, consequently a modular reduction will be needed.

2.3 Notation

Let f be a k-tuple $f = (f_k, \ldots, f_0)$. We will use $(\,)_E$ to denote an "even polynomial" in $GF(2^n)$. We define $(f)_E = f_0 + f_1 x^2 + \cdots + f_{\frac{n-1}{2}} x^{2(\frac{n-1}{2})}$, such that if $2k < n$, then we define $f_i = 0$ for $k+1 \leq i \leq (n-1)/2$. Observe that if $2k \geq n$, then we are "truncating all f_i for $i \geq (n-1)/2 + 1$. We will use $(\,)_O$ to denote an "odd polynomial" in $GF(2^n)$ where $(f)_O = f_0 x + f_1 x^3 + \cdots + f_{\frac{n-3}{2}} x^{2(\frac{n-3}{2})+1}$. Again depending upon k (here we compare $2k+1$ with n), our representation maybe including several f_i which are zero *or* we maybe truncating some of the f_i. Lastly if $g = (\rho_w, \ldots, \rho_0)$ then both $(\rho_w, \ldots, \rho_0)_O$ and $(\rho_w, \ldots, \rho_0)_E$ are meaningful, they are odd and even polynomials of degree less than n. Furthermore $(\rho_w, \ldots, \rho_0)_O = x \cdot (\rho_w, \ldots, \rho_0)_E$.

We will use the following notation to demonstrate a shift of a polynomial (because we wish to demonstrate the effects of a polynomial of degree $\geq n$ we will truncate the shifted polynomial at the

degree n term). The notation we will adopt is as follows: let $C \in GF(2^n)$, and assume that C is an *odd* function, $C = (\eta_k, \ldots, \eta_0) = \eta_k x^{2k+1} + \cdots + \eta_1 x^3 + \eta_0 x$ then

$$x^{2i}C = (\eta_k, \ldots, \eta_0, \underbrace{0, \ldots, 0}_{i \text{ zeros}}) = \eta_{t+k} x^{2(k+i)+1} + \cdots + \eta_1 x^{2i+3} + \eta_0 x^{2i+1},$$

$x^i C$ is still an odd function.

$$x^{2i+1}C = (\eta_k, \ldots, \eta_0, \underbrace{0, \ldots, 0}_{i+1 \text{ zeros}}) = \eta_k x^{2(k+i)+2} + \cdots + \eta_1 x^{2i+4} + \eta_0 x^{2i+2},$$

$x^{2i+1}C$ is an even function. Of course both field elements $x^{2i}C$ and $x^{2i+1}C$ should be reduced modulo the generating polynomial and so they will be reduced to a degree which is less than n.

We can as well truncate the representations of $x^{2i}C$ and $x^{2i+1}C$ after the $n-1^{st}$ degree term. Do note that once we truncate the polynomial then we no longer have equivalent field elements. Our only interest in truncating polynomials will be truncating polynomials which are odd or even. Recall $()_O$, thus $(x^{2i}C)_O$ is an odd polynomial of degree less than n. Observe that $(x^{2i}C)_O$ does not necessarily equal $x^{2i}C$. Similarly $(x^{2i+1}C)_E$ is an even polynomial of degree less than n, and $(x^{2i+1}C)_E$ does not necessarily equal $x^{2i+1}C$.

We now consider right shifts. Of course their are two types of right shifts to consider one a shift where the result is an odd function and the other a shift where the result is an even function. To illustrate let $C \in GF(2^n)$, where C is an odd function. Then

$$\frac{C}{x^{2i}} = (0, \ldots, 0, \eta_k, \ldots, \eta_{0+i})$$

causes a right shift in the originally odd function C such that $\frac{C}{x^{2i}}$ is as well an odd function (here there are i many zeros). $\frac{C}{x^{2i+1}} = (0, \ldots, 0, \eta_k, \ldots, \eta_{0+i+1})$ causes a right shift in the originally odd function C such that $\frac{C}{x^{2i}}$ is as well an even function.

3 Our method for squaring

As discussed earlier the suggested method of performing a squaring when utilizing a polynomial basis consists of insertion of zeros, generating a polynomial of degree nearly twice the length of a field element, then to finally complete the computation perform a modular reduction. An observation is that the expanded polynomial is "special" in that every other coefficient is zero. Thus some of the field additions (that are required in the modular reduction) will require a nonzero coefficient added to zero.

At the core of our suggested implementation of the square is that we perform field additions on polynomials whose length is approximately half of the field length. This is achieved by partitioning the element into it's odd and even part. Let $\zeta = \mu^2$ where $\mu = (\mu_{n-1}, \mu_{n-2}, \ldots, \mu_1, \mu_0)$, we will assume that n is odd. Then $\zeta = (\mu_{n-1} 0, \mu_{n-2}, 0, \ldots, 0, \mu_1, 0, \mu_0)$. We partition μ as follows: let $\mathcal{A} = (\mu_{\frac{n-1}{2}}, \ldots, \mu_1, \mu_0)$ and $\mathcal{B} = (\mu_{n-1}, \ldots, \mu_{\frac{n-1}{2}+1})$. Then $\mu = \mathcal{B} x^{(n-1)/2} + \mathcal{A}$ (we assume that n is odd). Observe that $\zeta = (\mathcal{B})_O x^n + (\mathcal{A})_E$. Since the generating polynomial of $GF(2^n)$ is $x^n + p(x) + 1$, we can express ζ as $(\mathcal{B})_O(p(x)+1) + (\mathcal{A}_E$. Consequently we can see that

$$\mathcal{O} = \text{odd part of } \zeta = \mathcal{B} + f_1$$

$$\mathcal{E} = \text{even part of } \zeta = \mathcal{A} + f_2$$

where $f = f_1 + f_2$ is dependent on the reduction of $p(x)(\mathcal{B})_O$. Once we have computed $\mathcal{O}$ and $\mathcal{E}$, to complete the computation of ζ we need to interleave the elements of $\mathcal{O}$ and $\mathcal{E}$, to compute ζ.

Throughout the paper we will assume that n is odd. In applications of ECC the work [2] has brought into question the security of elliptic curve defined over fields $GF(2^n)$ when n is composite. Since then most literature has suggested to use n prime. Let $L = \lfloor \frac{n-1}{2} \rfloor = \frac{n-1}{2}$.

4 Computing squares where generating polynomial is $x^n + x^m + 1$

Let $\mu \in GF(2^n)$, then $\mu = (\mu_{n-1}, \ldots, \mu_L, \mu_{L-1}, \ldots, \mu_0)$ We will use $\mathcal{A} = (\mu_L, \ldots, \mu_0)_E$ and $\mathcal{B} = (\mu_{n-1}, \ldots, \mu_{L+1})_O = (\mu_{L+L}, \ldots, \mu_{L+1})_O$. We denote the square of μ by ζ, i.e. $\zeta = \mu^2$.

4.1 $m = 2I + 1$ odd

Assume that $2m \leq n - 1$. $\zeta = \mathcal{A} + \mathcal{B} + x^m\mathcal{B}$

$$
\begin{aligned}
x^m\mathcal{B} &= x^{2I+1}(\mu_{L+L}, \ldots, \mu_{L+1})_O \\
&= x^{2I+1}(\mu_{L+L}x^{2L-1} + \cdots + \mu_{L+2}x^3 + \mu_{L+1}x) \\
&= \mu_{L+L}x^{2I+2L} + \cdots + \mu_{L+2}x^{2I+4} + \mu_{L+1}x^{2I+2} \\
&= (\mu_{L+L}x^{2I+2L} + \cdots + \mu_{L+(L+1-I)}x^{2I+2(L-I)+2}) + (\mu_{L+2(L-I)}x^{2I+2(L-I)} + \cdots + \mu_{L+2}x^{2I+4} + \mu_{L+1}x^{2I+2} \\
&= (\mu_{L+L}x^{2I+2L} + \cdots + \mu_{L+(L+1-I)}x^{2L+2}) + (\mu_{L+(L-I)}x^{2L} + \cdots + \mu_{L+2}x^{2I+4} + \mu_{L+1}x^{2I+2}) \\
&= (\mu_{L+L}x^{2I-2} + \cdots + \mu_{L+L+1-I})x^{n+1} + (\mu_{L+L-I}x^{n-1} + \cdots + \mu_{L+1}x^{2I+2}) \\
&= (\mu_{L+L}x^{2I-2} + \cdots + \mu_{L+L-I+1})(x^{m+1} + x) + (\mu_{L+L-I}x^{n-1} + \cdots + \mu_{L+1}x^{2I+2}) \\
&= (\mu_{L+L}x^{2I+2I} + \cdots + \mu_{L+L-I+1}x^{2I+2}) + (\mu_{L+L}x^{2I-1} + \cdots + \mu_{L+L-I+1}x) \\
&\quad + (\mu_{L+L-I}x^{n-1} + \cdots + \mu_{L+1}x^{2I+2}) = x^{2I+2}(\frac{\mathcal{B}}{x^{2(L-I)+1}})_E + (\frac{\mathcal{B}}{x^{2(L-I)}})_O + (x^{2I+1}\mathcal{B})_E \\
&= x^{2I+1}(\frac{\mathcal{B}}{x^{2(L-I)}})_O + (\frac{\mathcal{B}}{x^{2(L-I)}})_O + (x^{2I+1}\mathcal{B})_E
\end{aligned}
$$

Note that the first term of the above equation is an even polynomial. Also, recall $2L + 1 = n$ and that $2I + 1 = m$. Thus $2(L - I) = n - m$. We summarize the above equations.

$$\mathcal{O} = \text{odd part of } \zeta = \mathcal{B} + (\frac{\mathcal{B}}{x^{n-m)}})_O$$

$$\mathcal{E} = \text{even part of } \zeta = \mathcal{A} + x^m(\frac{\mathcal{B}}{x^{n-m}})_O + (x^m\mathcal{B})_E$$

Remark: A comment on some of the above calculations. The assumption that $m \leq n - 1$ implies the relation $4I < n$. If the assumption $m \leq n - 1$ is removed then further reductions (although straightforward) would be required. Now $L + 1$ bits are required to represent $\mathcal{A}$ and L bits are required to represent $\mathcal{B}$. Of course we represent $\mathcal{A}$ and $\mathcal{B}$ using the same amount of memory. Consider the calculation $(\mathcal{B})_E$, this is a division of x, because we have modified the odd function to an even one, in terms of actual data stored, no modification will need to take place. i.e. the data stored to represent $(\mathcal{B})_O$ is the same as $(\mathcal{B})_E$, consequently division by x is "free". To compute $\frac{\mathcal{B}}{x^{2(L-I)+1}} = \frac{(\mathcal{B})_E}{x^{2(L-I)}}$, we perform a right shift of $(\mathcal{B})_E$, $L - I$ places. Lastly to compute $x^{2I+2}(\frac{\mathcal{B}}{x^{2(L-I)+1}})_E$ requires a left shift of $(\frac{\mathcal{B}}{x^{2(L-I)+1}})_E$, $I + 1$ places. Total cost to compute this element was $L + 1$. Comparable statements can be made concerning the other terms in the equation. In terms of the number of additions required. To compute the odd part $\mathcal{O}$, it requires I many additions. To compute the even part $\mathcal{E}$, it requires $3(L - I)$ additions. The remaining cost is the cost of interleaving the odd and even parts.

4.2 $m = 2I$ even

Assume that $2m \leq n-1$. $\zeta = \mathcal{A} + \mathcal{B} + x^m\mathcal{B}$

$$\begin{aligned}
x^m\mathcal{B} &= x^{2I}(\mu_{L+L}x^{2L-1} + \cdots + \mu_{L+1}x) \\
&= \mu_{L+L}x^{2L+2I-1} + \cdots + \mu_{L+1}x^{2I+1} \\
&= (\mu_{L+L}x^{2L+2I-1} + \cdots + \mu_{L+L-I+1}x^n) + (\mu_{L+L-I}x^{n-2} + \cdots + \mu_{L+1}x^{2I+1}) \\
&= (\mu_{L+L}x^{2I-2} + \cdots + \mu_{L+L-I+1})(x^m+1) + (\mu_{L+L-I}x^{n-2} + \cdots + \mu_{L+1}x^{2I+1}) \\
&= (\mu_{L+L}x^{4I-2} + \cdots + \mu_{L+L-I+1}x^{2I}) + (\mu_{L+L}x^{2I-2} + \cdots + \mu_{L+L-I+1}) \\
&\quad +(\mu_{L+L-I}x^{n-2} + \cdots + \mu_{L+1}x^{2I+1}) \\
&= x^{2I}\left(\frac{\mathcal{B}}{x^{2(L-I)+1}}\right)_E + \left(x^{2I}\mathcal{B}\right)_O + \left(\frac{\mathcal{B}}{x^{2(L-I)+1}}\right)_E
\end{aligned}$$

$$\mathcal{O} = \text{odd part of } \zeta = \mathcal{B} + \left(x^{2I}\mathcal{B}\right)_O = \mathcal{B} + (x^m\mathcal{B})_O$$

$$\mathcal{E} = \text{even part of } \zeta = \mathcal{A} + x^{2I}\left(\frac{\mathcal{B}}{x^{2(L-I)+1}}\right)_E + \left(\frac{\mathcal{B}}{x^{2(L-I)+1}}\right)_E = \mathcal{A} + x^m\left(\frac{\mathcal{B}}{x^{n-m}}\right)_E + \left(\frac{\mathcal{B}}{x^{n-m}}\right)_E$$

It requires $L-I$ additions to compute $\mathcal{O}$ and $2I$ additions to compute $\mathcal{E}$.

5 Computing squares when generating polynomial is a pentanomial

As stated earlier in most applications of ECC defined over fields $GF(2^n)$, the generating polynomial is either a trinomial or a pentanomial. Assume that the generating polynomial is a polynomial of the form $x^n + x^a + x^b + x^c + 1$, which we denote by $x^n + p(x) + 1$. In practical applications, to achieve an efficient setting, i.e. speed, suitable pentanomials usually possess the property that the degree of $p(x)$ is small. For example in the NIST list of suggested elliptic curves [8], all the generating polynomials which are pentanomials satisfy that the degree of $p(x)$ is ≤ 12

Let $\mu = (\mu_{n-1}, \ldots, \mu_L, \mu_{L-1}, \ldots, \mu_0) \in GF(2^n)$. We will use $\mathcal{A} = (\mu_L, \ldots, \mu_0)$ and $\mathcal{B} = (\mu_{n-1}, \ldots, \mu_{L+1})$ Thus $\mu = \mathcal{B}x^{L+1} + \mathcal{A}$. Set $\zeta = \mu^2$.

$$\zeta = \mathcal{A} + \mathcal{B} + x^a\mathcal{B} + x^b\mathcal{B} + x^c\mathcal{B}$$

Both polynomials $x^a\mathcal{B}$ and $x^b\mathcal{B}$ will have degrees $\geq n$. For such terms of degree $\geq n$, we will have to apply $x^n = x^a + x^b + x^c + 1$. The polynomial $x^c\mathcal{B}$ will have degree $\geq n$, whenever $c > 1$.

It follows that $\zeta = \mathcal{A} + (\mathcal{B}x^a + \mathcal{B}x^b + \mathcal{B}x^c + \mathcal{B})x$. Here $\mathcal{B}$ is an odd function. Multiplications by monomials like x^{a+1} will require shifting $\mathcal{B}$ (the result may be odd or even dependent on the powers). Some of the terms will exceed x^{n-1} so further reduction would be needed. However, these reductions can easily be tracked, especially since in practice the degree of $p(x)$ will be small.

Example Let us consider a generating polynomial which is the modulus for an elliptic curve that is utilized in many standards (for example both the NIST list [8] as well as the WAP list [15]). The generating polynomial is the pentanomial $x^{163} + x^7 + x^6 + x^3 + 1$. Here $p(x) = x^7 + x^6 + x^3$.
Let $\mu \in GF(2^{163})$ and $\zeta = \mu^2$. Let $\mathcal{A} = (\mu_{81}, \ldots, \mu_0)$ and $\mathcal{B} = (\mu_{162}, \ldots, \mu_{82})$.

$$\mathcal{O} = \text{odd part of } \zeta = \mathcal{B} + (x^6\mathcal{B})_O + f_1$$

$$\mathcal{E} = \text{even part of } \zeta = \mathcal{A} + (x^7\mathcal{B})_E + (x^3\mathcal{B})_E + f_2$$

We represent the remaining terms as $f = f_1 + f_2$ where
$f = \mu_{162}x(p(x)+1) + (\mu_{162}x^4 + \mu_{161}x^2 + \mu_{160})(xp(x) + x) + (\mu_{162}x^4 + \mu_{161}x^2 + \mu_{160})(p(x)+1)$.

This can be simplified:

$$\begin{aligned} f &= \mu_{162}x^{12} + (\mu_{162} + \mu_{162})x^{11} + (\mu_{161} + \mu_{162})x^{10} + +(\mu_{161} + \mu_{161})x^9 \\ &\quad +(\mu_{160} + \mu_{161} + \mu_{162} + \mu_{162})x^8 + (\mu_{160} + \mu_{160} + \mu_{162} + \mu_{162})x^7 \\ &\quad +(\mu_{161} + \mu_{160})x^6 + (\mu_{161} + \mu_{162})x^5 \\ &\quad +(\mu_{160} + \mu_{162} + \mu_{162})x^4 + (\mu_{160} + \mu_{161})x^3 + \mu_{161}x^2 + (\mu_{160} + \mu_{162})x + \mu_{160} \\ &= \mu_{162}x^{12} + (\mu_{161} + \mu_{162})x^{10} + (\mu_{160} + \mu_{161})x^8 \\ &\quad +(\mu_{161} + \mu_{160})x^6 + (\mu_{161} + \mu_{162})x^5 \\ &\quad +\mu_{160}x^4 + (\mu_{160} + \mu_{161})x^3 + \mu_{161}x^2 + (\mu_{160} + \mu_{162})x + \mu_{160} \end{aligned}$$

Again once $\mathcal{O}$ and $\mathcal{E}$ are computed, then one interleaves the elements of $\mathcal{O}$ and $\mathcal{E}$ to compute ζ.

6 Computing square roots in $GF(2^n)$

6.1 Applications

Computing square roots has several applications, they include: A significant reason to consider ECC is the small bandwidth. To transmit a signature and/or a public key requires to transmit a point on the elliptic curve. Hence an ordered pair. To assure small bandwidth, it is possible to compress the ordered pair to the size of the field element. To expand the point will require the computation of a square root [14]. In such a case only one square root will be required for each signature/key derivation.

One advantage to utilizing ECC is the improved performance. An enormous amount of research has taken place in this area trying to explore further ways to develop increased performance. One alternative is to use anomalous binary curves which have been characterized as Koblitz curves [12]. Such curves have "special" algebraic properties that can be exploited to improve performance. By applying the frobenius map to the key [12], one can express the private key in $\mathbf{Z}[\tau]$ where $\tau^2 - \tau + 2 = 0$. For $P = (x, y)$ belonging to the elliptic curve $\tau(x, y) = (x^2, y^2)$. Thus to compute kP we perform a the "τ and add" method. Further algebraic properties implies that we can determine an equivalent key in $\mathbf{Z}[\tau^{-1}]$, hence one can use a "τ^{-1} and add" method to compute kP. Here $\tau^{-1}(x, y) = (\sqrt{x}, \sqrt{y})$ and so we see another application of square roots.

A third application of square roots is again in the area of improving performance of ECC. In [6, 11], Knudsen and Schroepel separately developed the method to use halving a point to compute kP. In Knudsen's method, he relies on determining the square root of a field element for every bit of the private key. The halving of a point algorithm provides a promising method of improving performance in ECC, as well as does not require one to resort to curves which have special algebraic properties.

A last application of using square roots lies in the area of protecting a key. In [4] described several approaches for protecting the private key against a DPA attack. In one of the methods computing τ^{-1} of a point was a necessary computation.

6.2 How to compute square roots

If one uses a normal basis to represent elements in $GF(2^n)$, then one can compute a square root by performing a circular shift.

The IEEE P1363 Public key cryptography working group [14] described how to perform needed algebraic operations. Let $\zeta \in GF(2^n)$. In [14] they described how to $\sqrt{\zeta}$ as $\sqrt{\zeta} = \zeta^{n-1}$

In [6], Knudsen described a method to compute square roots as follows. For $\zeta \in GF(2^n)$, $\zeta = \xi_0 + \xi_1 x + \cdots + \xi_{n-1}x^{n-1}$, then

$$\sqrt{\zeta} = \sum_{\substack{i=0 \\ i\ even}}^{n-1} \xi_i x^{i/2} + \sqrt{x} \sum_{\substack{i=0 \\ i\ odd}}^{n-1} \xi_i x^{(i-1)/2}$$

To compute square roots, one could precompute the square root of x and then perform the field multiplication of $\sqrt{x}$ with an element of $GF(2^n)$ of length $\frac{n-1}{2}$. This is a significant improvement on the IEEE P1363 method. As we shall soon see, there exists an even more efficient algorithm to compute a square root.

6.3 How to compute square roots when generating polynomial is a trinomial

The relationship between μ and μ^2, provides great insight into how to compute square roots. Let $\zeta = \mu^2$, our goal is to compute μ. We write the generating polynomial as $x^n + x^m + 1$. Again for ECC purposes n should be chosen to be prime. We will assume that $2m \leq n-1$, this assumption allows us to demonstrate this computations in few steps. This assumption can be removed, the consequence is that the computation will take more steps, each step is comparable to steps we illustrate. Again we use L to denote $\frac{n-1}{2}$. Let $\zeta = (\xi_{L+L}, \ldots, \xi_{L+1}, \xi_L, \ldots, \xi_0)$. Let $\mathcal{O} = (\xi_{L+L-1}, \ldots, \xi_3, , \xi_1)_O$ and let $\mathcal{E} = (\xi_{L+L}, \ldots, \xi_2, \xi_0)_E$, $\zeta = \mathcal{O} + \mathcal{E}$. Our goal is to compute $\mathcal{A}$ and $\mathcal{B}$, where $\mu = x^{L+1}\mathcal{B} + \mathcal{A}$. That is we start with ζ, and express $\zeta = \mathcal{O} + \mathcal{E}$. From $\mathcal{O}$ our goal will be to compute $\mathcal{B}$. Once we have $\mathcal{B}$ we compute $\mathcal{A}$. Then $\sqrt{\zeta} = \mathcal{B}x^{L+1} + \mathcal{A}$.

6.4 $m = 2I + 1$ is odd

Let $\mathcal{O}$ denote the odd part of ζ. By recalling the derivation of the square as given earlier, we have $\mathcal{O} = \mathcal{B} + (\frac{\mathcal{B}}{x^{2(L-I)}})_O$. Performing a right shift of $\mathcal{O}$ of suitable number of I bits (i.e. divide by x^{2I}) will shift out the term $(\frac{\mathcal{B}}{x^{2(L-I)}})_O$, the result is equivalent to performing a right shift of $\mathcal{B}$ by I bits. The consequence will be that we now have the high $L-I$ bits of $\mathcal{B}$, since $L > I$ we have recovered more than half of the bits of $\mathcal{B}$. To compute the remaining bits of $\mathcal{B}$ we perform a right shift by a suitable number of bits of the known high bits of $\mathcal{B}$, and then add this to $\mathcal{O}$ the result will be the remaining lower bits of $\mathcal{B}$. (This recovery of $\mathcal{B}$ takes two steps due to our assumption that $2m \leq n-1$) The irony is that for this case, the calculations required to compute $\mathcal{B}$ can be simplified as:

$$\mathcal{B} = \mathcal{O} + (\frac{\mathcal{O}}{x^{2(L-I)}})_O = \mathcal{O} + (\frac{\mathcal{O}}{x^{n-m}})_O$$

Once $\mathcal{B}$ is known we can calculate $\mathcal{A}$ by

$$\mathcal{A} = \mathcal{E} + x^m(\frac{\mathcal{B}}{x^{n-m}})_O + (x^m\mathcal{B})_E$$

Remark
Observe that the relationship between $\mathcal{B}$ and $\mathcal{O}$ in the square root algorithm is precisely the same as the relationship between $\mathcal{O}$ and $\mathcal{B}$ in the square algorithm.

This algorithm to compute a square root (for this type of generating polynomial) is a significant improvement to the algorithm which requires a field multiplication. We also see that the cost of computing square root is precisely the same as the cost of squaring.

We illustrate the computations to compute the square root. Consider the field $GF(2^{409})$ with the generating polynomial $x^{409} + x^{87} + 1$. This represents the underlying field for one of the curves in the NIST list of elliptic curves [8]. Let $\zeta = \mathcal{O} + \mathcal{E}$. To compute $\sqrt{\zeta}$ we compute $\mathcal{A}$ and $\mathcal{B}$. $\mathcal{B} = \mathcal{O} + (\frac{\mathcal{O}}{x^{322}})_O$ and $\mathcal{A} = \mathcal{E} + x^{87}(\frac{\mathcal{B}}{x^{322}})_O + (x^{87}\mathcal{B})_E$. Then $\sqrt{\zeta} = \mathcal{B}x^{L+1} + \mathcal{A}$.

6.5 $m = 2I$ even

Again we are assuming that m satisfies $2m \leq n-1$ (this case mimics the case considered for computing squares). Recalling the derivation of the square, the odd part $\mathcal{O}$ of ζ, satisfies $\mathcal{O} = \mathcal{B} + (x^m\mathcal{B})_O$. Again we perform a series of additions of $\mathcal{O}$ with shifts of $\mathcal{O}$ to compute $\mathcal{B}$. However since $2m \leq n-1$, the number of terms in $(x^m\mathcal{B})_O$ exceeds L, so we will have to perform at least two additions (and most possibly more). Consequently

$$\mathcal{B} = \mathcal{O} + (x^m\mathcal{O})_O + \left(x^{2m}\mathcal{O}\right)_O + \cdots + \left(x^{km}\mathcal{O}\right)_O$$

where integer k is the smallest positive integer satisfying $(k+1) \geq n$ (which is equivalent to $(k+1)I > L$). Once $\mathcal{B}$ is computed we compute $\mathcal{A}$ as

$$\mathcal{A} = \mathcal{E} + x^m\left(\frac{\mathcal{B}}{x^{n-m}}\right)_E + \left(\frac{\mathcal{B}}{x^{n-m}}\right)_E$$

Remark

We illustrate the computations to compute the square root. Consider the field $GF(2^{233})$ with the generating polynomial $x^{233} + x^{74} + 1$. This represent the underlying field for one of the curve in the NIST list of elliptic curves [8]. Let $\zeta = \mathcal{O} + \mathcal{E}$. To compute $\sqrt{\zeta}$ we compute $\mathcal{A}$ and $\mathcal{B}$. Here $\mathcal{B} = \mathcal{O} + (x^{74}\mathcal{O})_O + (x^{148}\mathcal{O})_O + (x^{222}\mathcal{O})_O$ and $\mathcal{A} = \mathcal{E} + x^{74}(\frac{\mathcal{B}}{x^{159}})_E + (\frac{\mathcal{B}}{x^{159}})_E$.

6.6 Computing square roots – other generating polynomials

We could describe the method to compute square roots for the remaining cases of generating polynomials which are trinomials. However due to space we omit them here. We point out that for cryptographic purposes such polynomials usually are not used due to efficiency.

Our method for computing a square root for the case when the generating polynomial is a pentanomial is successful for only a subset of the pentanomials. (By our method, we are referring to the method of expressing $\zeta = \mu^2$ as $\zeta = \mathcal{O} + \mathcal{E}$ and then computing $\mathcal{B}$ from $\mathcal{O}$. Once $\mathcal{B}$ is known, $\mathcal{A}$ can easily be computed and $\sqrt{\zeta} = \mathcal{B}x^{L+1} + \mathcal{A}$.) For example, our method is successful for computing square roots in $GF(2^{163})$ where the generating polynomial is $x^{163} + x^7 + x^6 + x^3 + 1$ (due to space we omit the details). At the same time our method cannot compute $\sqrt{\zeta}$ in $GF(2^{163})$ when the generating polynomial is $x^{163} + x^8 + x^2 + x + 1$ which is the underlying field for one of the elliptic curve in the WAP list of curves [15].

7 Conclusion

This work has provided a new approach towards computing squares in $GF(2^n)$. This algorithm is such that it is straightforward to develop a software implementation. In applications within ECC, the size of n is such that this new approach will not provide significant improvements for an ECC implementation. However for a $GF(2^n)$ discrete-log cryptosystem the key size would be extremely large, hence our squaring algorithm could possibly provide improved performance.

Our approach towards computing squares has provided insight towards a new method for computing square roots in $GF(2^n)$. For the cases when the generating polynomial is a trinomial this new method is significantly more efficient than previously known methods.

Future work is to explore how this new method of computing square roots may be used. In particular to explore how it may affect the performance of the halving a point algorithm that is used to compute in ECC.

References

[1] I.F. Blake, Nigel Smart, and G. Seroussi, *Elliptic Curves in Cryptography*. London Mathematical Society Lecture Note Series. Cambridge University Press, 1999.

[2] Gaudry, P. Hess, F. and Smart, N. Constructive and destructive facets of Weil descent on elliptic curves, *J. of Cryptology*, 15 (2002), 19-46.

[3] Hankerson, D., Hernandez, J., and Menezes, A. "Software implementation of elliptic curve cryptography over binary fields", *Proceedings of CHES 2000, Lecture Notes in Computer Science*, 1965 (2000), p. 1-24.

[4] Hasan, M.A. "Power analysis attacks and algorithmic approaches to their countermeasures for Koblitz curve cryptosystems" *IEEE Transactions on Computers*, Volume: 50 Issue: 10 , Oct. 2001 Page(s): 1071 -1083

[5] Neal Koblitz, *Elliptic curve cryptosystems*, Mathematics of Computation, Vol. 48, No. 177, 1987, 203-209.

[6] Erik Woodward Knudsen. "Elliptic Scalar Multiplication Using Point Halving". In *Advances in Cryptology - ASIACRYPT '99*. LNCS Vol. 1716, Springer, 1999, p. 135-149

[7] Victor S. Miller, "Use of Elliptic Curves in Cryptography", In *Advances in Cryptology CRYPTO 1985*,Springer-Verlag, New York, 1985, pp 417-42

[8] NIST, *Recommended elliptic curves for federal use*, http://www.nist.gov

[9] Odlyzko, A. M. "Discrete logarithms: The past and the future", *Designs, Codes, and Cryptography*, 19 (2000), pp. 129-145.

[10] Richard Schroeppel, Hilarie Orman, Sean W. O'Malley, Oliver Spatscheck: "Fast Key Exchange with Elliptic Curve Systems", In *Advances in Cryptology - CRYPTO '95*, Lecture Notes in Computer Science, Vol. 963, Springer, 1995, pp 43-56.

[11] Rich Schroeppel. "Elliptic Curves: Twice as Fast!". In*Rump session of CRYPTO 2000.*

[12] Solinas, J., "Efficient arithmetic on Koblitz curves". *Journal of Designs, Codes and Cryptography*, 19 (2000), 195-249.

[13] Huapeng Wu. "Bit-parallel finite field multiplier and squarer using polynomial basis". *IEEE Transactions on Computers*, Volume: 51 Issue: 7 , July 2002 Page(s): 750 -758.

[14] *IEEE P1363 Appendix A*. http://www.grouper.org/groups/1363

[15] *WTLS Specification*, http://www.wapforum.org