Performance of Finite Field Arithmetic in an Elliptic Curve Cryptosystem

Zhi Li, John Higgins, Mark Clement
3361 TMCB
Brigham Young University
Provo, UT 84602
{zli,higgins,clement}@cs.byu.edu

Abstract

As the Internet commerce becomes a more important part of the economy, network security is receiving more emphasis. Time spent in data encryption can be a significant performance bottleneck for many applications. Elliptic Curve Cryptography (ECC) has been shown to provide stronger encryption than conventional integer factorization schemes such as RSA or discrete logarithm-based systems such as Diffie-Hellman. This research explores the efficiency advantages of ECC by implementing and evaluating the underlying finite field arithmetic of the ElGamal encryption protocol using both polynomial basis (PB) and normal basis (NB) representations. The Normal basis implementation performs more than two times as fast as the polynomial basis representation and more than 50 times faster than traditional encryption schemes.

Key Words : Elliptic Curve, Encryption, ElGamel, normal basis, polynomial basis

1 Introduction

Private and public key cryptography [1] are two major tools for data security. Private key schemes such as the Data Encryption Standard (DES) allow fast encryption and message authentication. Public key schemes provide unique ways for key distribution and assurance of sender's authenticity and non-repudiation. There are three families of public key algorithms that have considerable significance in practice. These include integer factorization, discrete logarithm, and elliptic curve-based schemes [2, 3]. Integer factorization-based schemes such as RSA [4] and Discrete Logarithm-based schemes such as Diffie-Hellman [5] and digital signature provide intuitive ways of implementation. However, both schemes are vulnerable to cryptanalysis because of the existence of sub-exponential algorithms for cracking such cryptosystems [7]. In this regard, elliptic curve cryptography, first introduced by Koblitz [2] and Miller [3], is the most secure public key cryptographic method available [6,8]. Since the computational complexity for breaking ECC is totally exponential to the key size, ECC offers smaller key sizes for the same level of cryptographic security than conventional systems. For example, ECC with a key size of 173 bits provides the same level of cryptographic security as RSA with a key size of 1024 bits. This results in smaller system parameters, bandwidth savings, faster implementations and lower power consumption. In addition, elliptic curves over finite fields offer an inexhaustible supply of finite abelian groups, thus allowing more flexible field selections than conventional discrete logarithm schemes. Because of these advantages, ECC has attracted extensive attention in recent years [9,13]. It is also expected that ECC will be widely used for many security applications in the near future.

Previous research on ECC has covered a broad range of topics from security issues and standardization to implementation and performance [14,21]. Many algorithms have been designed to improve the performance of the underlying field arithmetic.

However, most of those methods are limited to polynomial base representations. Normal basis in software has seldom been studied. Several significant questions remain unanswered, including:

- "What is the relative efficiency of the normal and polynomial basis representations for field arithmetic?"
- "How efficiently may theoretical normal basis algorithms be implemented in software?"
- "What is the actual efficiency advantage of ECC when compared with traditional public key schemes?"

This information plays an important role in the design of a public key cryptographic system when speed is important. Different underlying basis representations may lead to drastic changes in the performance of corresponding cryptographic systems. This research provides an implementation and evaluation of finite field arithmetic, elliptic group operations, and an elliptic curve cryptosystem using both polynomial and normal basis representations. A thorough comparison of the software performance of polynomial and normal basis representations is also provided.

2 Elliptic Curve Mathematics

ECC involves several areas of mathematics including finite fields, representations of field elements, and group theory. In this section we describe the mathematics necessary to understand the main algorithms being investigated in this research.

An Elliptic Curve (EC) over a finite field consists of a set of elements of an additive abelian group. Field arithmetic operations affect overall performance significantly. The efficiency of field arithmetic operations presumably depends on how they are represented. In an EC cryptosystem, a message can be embedded as an element of the group, and group operations are applied for encryption / decryption of the message. Such operations are fundamentally nonlinear and automatically randomize the input message resulting in a drastic change in the output .

These schemes make it difficult to deduce the original message from the randomized output [24, 25]. Theoretically, the security of EC cryptosystems relies on the difficulty of solving a discrete logarithm problem (DLP) on an *additive group* of points of an EC, which is much harder than and different from those of the two conventional public key cryptosystems (conventional integer factorization schemes such as RSA or discrete logarithm-based systems such as Diffie-Hellman) [6, 8, 14, 27]. This is because current sub-exponential algorithms for attacking the encryption are only applicable for solving the DLP on a *multiplicative group* of a finite field and are of little value in attacking the analogous elliptic curve problem.

2.1 Polynomial Basis (PB) verses Normal Basis (NB) Representation

There are two equivalent representations for an element of a finite field GF(2^n), PB and NB [19]. A PB is of the form {1, α, α^2, … , α^{n-1}}, where α is a root of an irreducible polynomial f(x) of degree n over field F_2. An element in GF(2^n) can then be represented as a polynomial:

$\{c_{n-1}\alpha^{n-1} + c_{n-2}\alpha^{n-2} + \ldots + c_2\alpha^2 + c_1\alpha + c_0 \mid c_i = 0 \text{ or } 1\}$,

or in vector form $\{c_{n-1}c_{n-2} \ldots c_2c_1c_0\}$.

A NB is of form { α, α^2, $\alpha^{2^\wedge 2}$, … , $\alpha^{2^\wedge(n-1)}$} for some $\alpha \in$ GF(2^n). Similarly, an element in GF(2^n) can be represented in NB as a vector $\{c_{n-1}c_{n-2}\ldots c_2c_1c_0\}$, where c_i = 0 or 1. NB can also be optimized for some values of n, leading to two types of optimal normal basis (ONB) called *Type I* and *Type II*. A field GF(2^n) is of *Type I* if n satisfies conditions: (n + 1) is prime and 2 is primitive in F(n + 1). (Note that "2 is primitive in F(n+1)" means that 2 generates all elements of prime field (n+1). For example, if (n+1) = 5, then 2^0 = 1 mod 5, 2^1 = 2 mod 5, 2^2 = 4 mod 5, 2^3 = 3 mod 5, 2^4 = 1 mod 5. Therefore, 2 generates all elements: 1, 2, 3, 4 in field of 5.) A field GF(2^n) is of *Type II* if n satisfies conditions: (2n + 1) is prime and: either 2 is primitive in F(2n + 1) or (2n + 1) $\equiv$ 3 mod 4 and 2 generates quadratic residues in F(2n + 1).

NB and PB can be converted to each other. For example, for *Type I* ONB, since 2 is primitive in F(n +1), each term (bit) of an element in a NB with the form: $c_i\alpha^{2^\wedge i}$ corresponds to a term (bit) of an element in a PB with the form $c_i\alpha^k$ where k = 2^i mod n. This correspondence can be precomputed into a table. An element in *Type I* ONB can then be converted into an element in PB by permutation of every bit of the element through table-lookup, and vice verse. The irreducible polynomial for the corresponding PB is $(1 + \alpha + \alpha^2 + \ldots + \alpha^{n-1})$. For *Type II* ONB, each term of an element in ONB with the form: $c_i\alpha^{2^\wedge i}$ corresponds to two terms of an element in a PB with the form: $c_i\alpha^k$ where k = 2^i mod n or k = (2n + 1 - 2^i) mod n. Since the irreducible polynomial for the corresponding PB is $(1 + \alpha + \alpha^2 + \ldots + \alpha^{2n})$ in this case, an element in PB is twice as long as the corresponding element in *Type II* ONB.

2.2 Finite Field Arithmetic Operations

The efficiency of EC algorithms heavily depends on the performance of the underlying field arithmetic operations. These operations include *addition*, *subtraction*, *multiplication*, and *inversion*. Given two elements, $(a_{n-1}\ldots a_1a_0)$ and $(b_{n-1}\ldots b_1b_0)$, these operations are defined as follows.

Both *addition* and *subtraction* are same between the two representations:

$(a_{n-1}\ldots a_1a_0) \pm (b_{n-1}\ldots b_1b_0) = (c_{n-1}\ldots c_1c_0)$, where $c_i = a_i + b_i$ over F(2).

Note that in $GF(2^n)$, since $(a_{n-1}\ldots a_1a_0) + (a_{n-1}\ldots a_1a_0) = (0\ldots00)$, each element $(a_{n-1}\ldots a_1a_0)$ is its own additive inverse. *Addition* and *subtraction* can be implemented efficiently as component-wise exclusive OR in both representations.

Multiplication and *inversion* are different between the two representations. In the PB representation, an irreducible polynomial $f(\alpha)$ of degree n over field F_2 is needed and *multiplication* is defined as:

$(a_{n-1}\ldots a_1a_0)\,(b_{n-1}\ldots b_1b_0) = (c_{n-1}\ldots c_1c_0)$, where $(c_{n-1}\alpha^{n-1} + \ldots + c_1\alpha + c_0)$ is the remainder when the polynomial $(a_{n-1}\alpha^{n-1} + \ldots + a_1\alpha + a_0)(b_{n-1}\alpha^{n-1} + \ldots + b_1\alpha + b_0)$ is divided by the polynomial $f(\alpha)$. *Multiplication* contains two steps: *polynomial (partial) multiplication* and *modular reduction*. *Partial multiplication* can be implemented using the "shift and add" idea that has been widely employed for multiplication of integers [22, 23]. Since *addition* in $GF(2^n)$ does not generate carries and is simply component-wise exclusive OR, this step is simplified. Further simplification can also be achieved when the two operands are the same. In this case, *partial multiplication* is implemented with the *squaring* operation. Since:

$$(a_{n-1}\alpha^{n-1} + \ldots + a_1\alpha + a_0)(a_{n-1}\alpha^{n-1} + \ldots + a_1\alpha + a_0) = (a^2_{n-1}\alpha^{2n-2} + \ldots + a^2_1\alpha^2 + a_0)$$

The *squaring* operation can be implemented by inserting a 0 bit between every two original polynomial bits. The next step is *modular reduction*. This step can be optimized by selecting the irreducible polynomial f(x) as a trinomial or pentanomial.

Inversion in the PB representation is defined as follows: Given an element $(a_{n-1}\ldots a_1a_0)$ and an irreducible polynomial f(x) of degree n, find the element

$(b_{n-1}\ldots b_1b_0)$ such that

$$(a_{n-1}\ldots a_1a_0)\,(b_{n-1}\ldots b_1b_0) \equiv 1 \bmod f(x).$$

The best-known *inversion* method is the "almost inverse algorithm" recently proposed by R. Schroeppel et al [15]. Briefly, it first computes an intermediate element $(c_{n-1}\ldots c_1c_0)$ that satisfies the equation:

$$(a_{n-1}\ldots a_1a_0)\,(c_{n-1}\ldots c_1c_0) \equiv x^k \bmod f(x)$$

and then produces the *inversion*, $(b_{n-1}\ldots b_1b_0)$, by dividing out the x^k from $(c_{n-1}\ldots c_1c_0)$.

Inversion in the NB representation is defined as follows: Given an element $(a_{n-1}\ldots a_1a_0)$, find the element $(b_{n-1}\ldots b_1b_0)$ such that $(a_{n-1}\ldots a_1a_0)\,(b_{n-1}\ldots b_1b_0) = 1$, where 1 is defined as:

$$\mathbf{1} = \sum_{\mathbf{i=0}}^{\mathbf{n-1}} \boldsymbol{a}^{2^i} = \boldsymbol{a} + \boldsymbol{a}^2 + \ldots + \boldsymbol{a}^{2^{n-1}}$$

2.3 Elliptic Curve Cryptography

An EC over $GF(2^n)$ is defined to be the set of points (x, y) satisfying an equation of the form $y^2 + xy = x^3 + ax^2 + b$, where a, b, x, y $\in$ $GF(2^n)$, $b \neq 0$, together with an extra point *O*, called the point at infinity [24-26]. The elliptic curve discrete logarithm problem (EC-DLP) is as follows: given an EC over the field $GF(2^n)$, a base point B of order n and a point P on EC, determine the integer x, $0 < x < n-1$, such that $P = xB$, provided such an x exists. The security of ECC relies on the computational complexity of the EC-DLP, which is a much harder problem than either the integer factoring problem or general DLP [6, 14].

All public key cryptographic schemes that make use of the DLP in finite fields can be implemented to work in the EC case [6, 15, 17]. In this research, we choose the ElGamal protocol, implemented an EC version of it, and evaluated the overall performance of the cryptosystem.

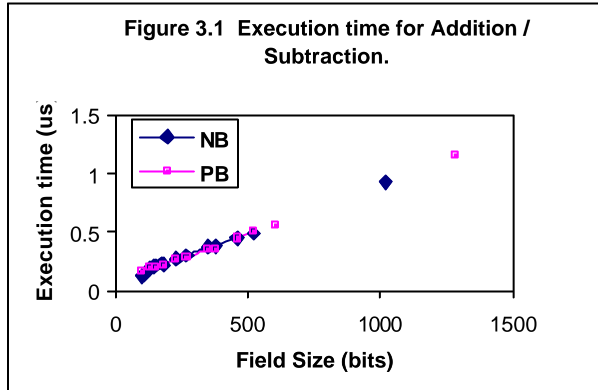
The protocol works as follows: Assume that the plaintext message m has been imbedded as point P_m on an publicly known EC over the field $GF(2^n)$ and a base point B on EC is publicly known. Users A and B start by randomly choosing integers k_A and k_B, which are kept secret. A and B compute their public keys $P_A = k_A B$ and $P_B = k_B B$ respectively. To send P_m to B, A must randomly choose an integer r and sends the points $(rB, P_m + rP_B)$. To read P_m , B must multiply rB by k_B and subtract it from $P_m + rP_B$. Since $P_m = P_m + rP_B - k_B rB$, B is able to recover the message P_m. However, for any third party C to break the encrypted message $(P_m + rP_B)$, they would need to solve EC-DLP.

3 ARITHMETIC OPERATIONS OVER $GF(2^n)$

This section describes the performance of field arithmetic operations over $GF(2^n)$ with field size ranging from 100 to 1279 bits for PB and from 100 to 1019 bits for NB representations. Experiments were performed on a 175 MHz Pentium II architecture with the Linux operating system. Theoretically, once the field size n (where n is from $GF(2^n)$) is selected, the content of the input message should not affect the performance. A message is a byte-string of a given size. Since we can generate a random input message easily, a different random input message was used for each run. Reported execution times are the average of 20 independent runs with 20 different input messages of the same size. For the performance comparison between PB and NB representations, 5-10% difference in timings for the same type of operation is recognized as a significant difference in performance.

3.1 Addition and Subtraction

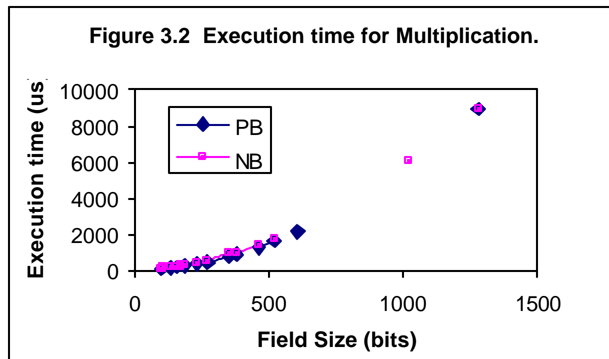
Addition and *subtraction* can be implemented efficiently by exclusive-ORing of two field elements. The performance of these operations is illustrated and Figure 1. It can be seen that the time complexities of *Addition/Subtraction* are proportional to the field



size and vary from 0.13 to 1.15 µs over the field size range from 100 to 1279 bits for both ONB and PB. This is expected from the given algorithm. The fact that the execution time for these two operations is so low for both ONB and PB suggests that the bottleneck for the algorithm does not lie in these operations.

3.2 Multiplication

Polynomial basis: Multiplication in PB contains two steps: *partial multiplication* and *modular reduction*. For *partial multiplication*, we employ the "shift and add" strategy: This algorithm runs in O(n*n/*ws*) time where n is the field size in bits, *ws* is machine word size. Figure 3.2 shows that the performance of *multiplication* in PB varies from 87 to 8960 µs over field sizes ranging from 100 to 1279 bits.

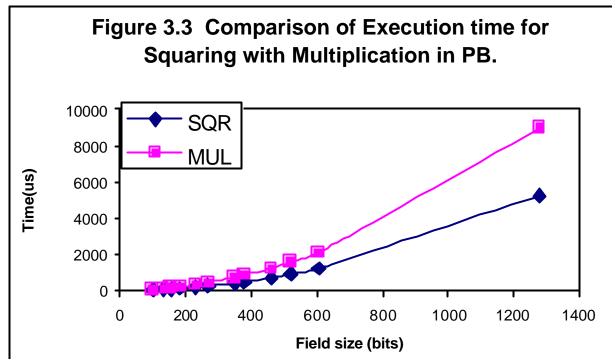


Normal basis: The implementation computes *multiplication* through shift, XOR, and AND. First, we shift one multiplier consecutively and store the results for later lookup. Since the lambda matrix can be pre-computed, we only need to rotate the other multiplier, lookup the lambda matrix table twice, and then carry out a simple XOR and AND. This algorithm takes O(n*n/*ws*) time where n is the field size in bit and *ws* is the machine word size. The execution time for this implementation is shown in Figure 3.2. Execution times vary from 108 to 6066 µs over field sizes ranging from 100 to 1019 bits.

Multiplication in PB is about 17% faster than that of NB. Both algorithms have similar time complexities. The PB implementation uses a trinomial to enhance the performance. The NB implementation uses table-lookup to simplify the computation.

3.3 Squaring

Squaring is just a special case of *multiplication*. Both PB and NB implementations can be simplified for this special case. This algorithm also runs in O(n) (note that the n here is Field size in bits rather than machine word size). The execution time for *squaring* in PB is shown in Figure 3.3. The NB implementation provides a 40% performance improvement on *squaring* over general *multiplication*.



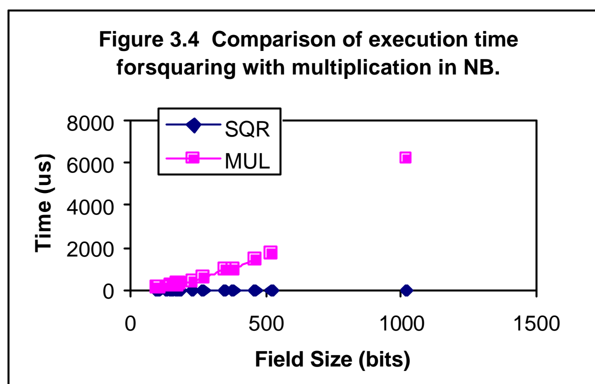


Figure 3.4 shows that the time for *squaring* is negligible when compared to the time for general *multiplication*. This 100% performance improvement is equivalent to (1 - 17%) = 83% improvement in *squaring* over multiplication in NB This is more than

twice the performance improvement that PB achieves (40%). This turns out to be the critical advantage of NB over PB with respect to overall performance.

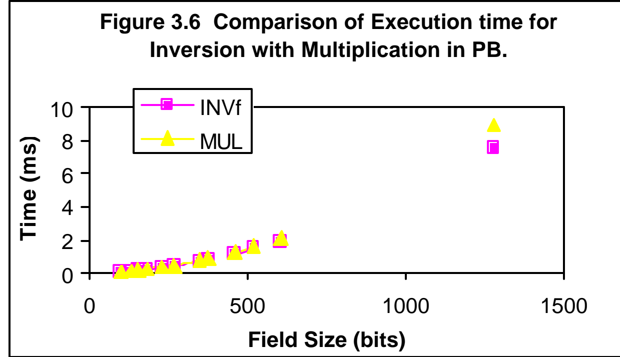


Figure 3.6 Comparison of Execution time for Inversion with Multiplication in PB.

3.4 Inversion

The "almost inverse algorithm" is the best-known method [15] for inversion in PB. The algorithm contains two steps. In the first step, the algorithm computes the intermediate polynomial B by repeatedly decreasing the length of polynomial F and G and increasing the length of polynomial B and C. The time complexity for the first step is roughly $O(n^2 / ws)$ where n is field size in bits and *ws* is a machine word size (32 bits in our case). The second step: dividing out x^k, can be simplified by using a trinomial $x^n + x^t + 1$ where $t > ws$.

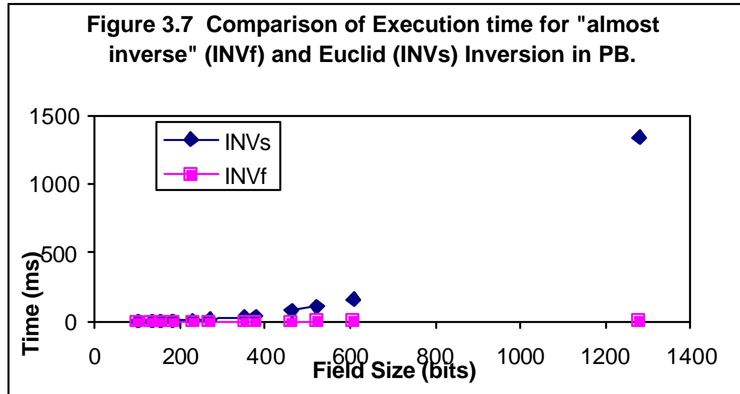
This algorithm runs in $O(n^2)$ where n is field size in machine word. Therefore, the overall time complexity is roughly $O(n^2 / ws)$ where n is field size in bits. Experimental results for *inversion* using the above implementation are given in Figure 3.6. Execution time for *inversion* varies from 77 to 7565 µs as the field size changes from 100 to 1279 bits. This result is 10% faster on average than *multiplication* in PB.

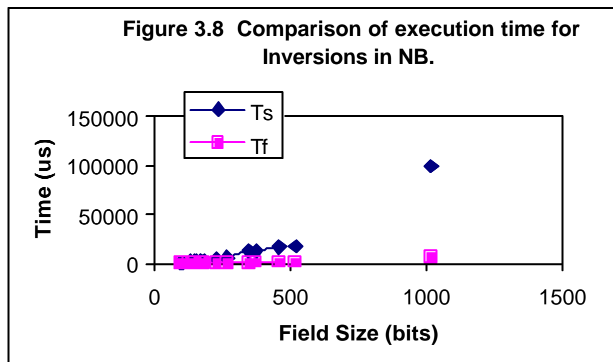
To further investigate any advantage of the "almost inverse" method, we also tested the performance of *inversion* using the conventional extension to Euclid's algorithm [23] and the results are given in Figure 3.7 for comparison purposes. It is clear that the conventional approach is not practical with respect to the run-time performance.

3.4.1 Normal Basis

We implemented and evaluated two approaches for inversion in NB. The first approach is based on Fermat's theorem. Since the algorithm converts *inversion* to a chain of *multiplication* and *squarings*, and *squaring* runs in negligible time, the algorithm takes $\{\log_2(n-1) + \text{the number of bits set in } (n-1) - 1\}$ *multiplications* where n is the field size in bit units. For example, for n = 155, n-1 = 154 = 0x9b, $\log_2(n-1) = 7$, the number of bits set in (n-1) = 4, hence it takes (7 + 4 – 1) = 10 *multiplications.* The performance of

inversion for NB is shown in Figure 3.8. It can be seen that the timings vary from 0.8 to 99.3 ms over the field size from 100 to 1019 bits. These timings are consistent with our analysis. It is clear that this approach is very inefficient with respect to performance.





The second approach is to take the efficiency advantage of the “almost inverse algorithm” in PB. In this approach, a field element is transformed from NB into PB. *Inversion* is then carried out in PB by using the “almost inverse” algorithm. Finally the result is transformed back into NB. The conversion between PB and NB is performed by means of table-lookup. The results of this approach are shown in Figure 3.10. The pure “almost inverse” method provides 10% speed advantages over the combination of the NB and the “almost inverse” method. This result indicates that *inversion* in NB can be implemented almost as efficiently as *inversion* in PB.

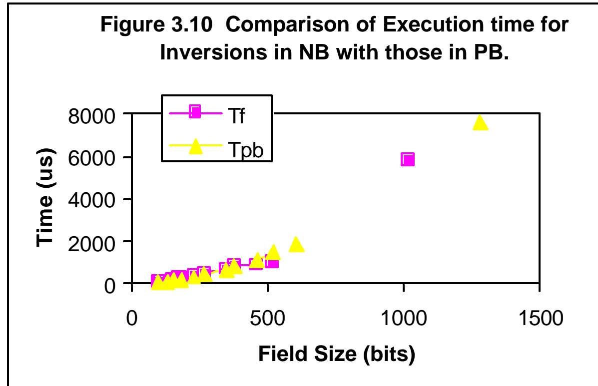


Figure 3.10 Comparison of Execution time for Inversions in NB with those in PB.

4 Elliptic Group Operations

Elliptic group operations include *point negation (PN)*, *point addition (PA)*, *point subtraction (PS), point doubling (PD)*, and *scalar multiplication (SM)*. Even though high-level implementations of these operations are same for PB and NB, the performance of them may be different since subroutines (underlying field operations) called in major functions may perform differently between PB and NB. We have used both the fast and slow algorithms of the underlying field operations as subroutines for these Group operations. We will focus on a discussion of the performance of the fast implementations of the underlying field operations between PB and NB.

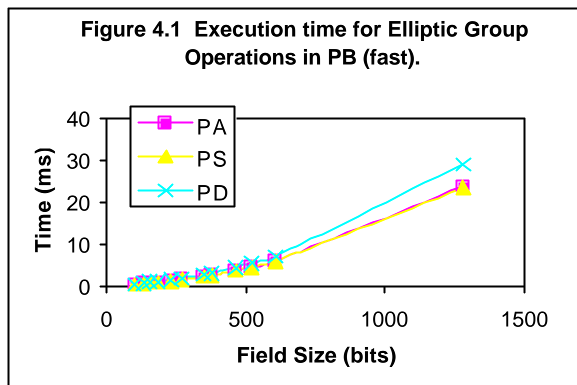


Figure 4.1 Execution time for Elliptic Group Operations in PB (fast).

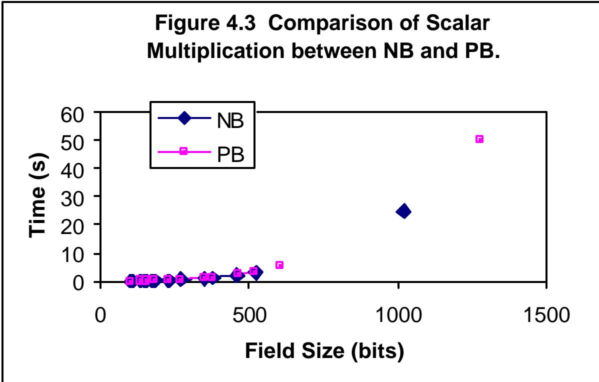
4.1 Polynomial Basis

Figure 4.1 shows that the execution time for *point addition* and *point subtraction* are very close to each other. This makes sense since *point subtraction* consists of *point addition* and *point negation*, and the time for negation is negligible. *Point doubling* is

18.7 % slower than *point addition*. This is probably because *point doubling* contains one extra field squaring operation when compared to *point addition*. *Point addition* consists of 1 *squaring* (S), 2 *multiplications* (M), and 1 *inversion* (I), which are equivalent to: 60% M + 2 M + 90%M =3.5 M based on our measurements of 1 S = 60% M and 1 I = 90% M. Therefore, the extra squaring accounts for (60%M) / (3.5M) = 17% *point addition*. Our measured time for *point doubling* is basically consistent with this analysis. The 1.7% variation is probably attributed to the constant factors in implementation and random errors. *Scalar multiplications* are the most time-consuming group operation. The time for this operation varies from 0.04 to 50 s over field sizes from 100 to 1279 bits. This is more than two orders of magnitude slower than *point addition*. Theoretically, the algorithm takes (fs – 1) *PD* and (fs/2 – 1) *PA* where fs is the field size in bits. Since *PD* = 118.7% *PA* based on our measurements, we can estimate: 1SM ≈ (fs – 1) * 118.7% *PA* + (fs/2 – 1) *PA* = (1.687 fs – 2.187) A. For example, for fs = 155, 1SM = (155*1.687 – 2.187) A = 259 A. Our measurements show the same order of magnitude of this correlation *between scalar multiplication* and *point addition*.

4.2 Normal Basis

The execution time for *point negation* is very similar to *point negations* in PB and negligible compared to timings of other group operations due to the same reasoning discussed earlier. This is also the reason that *point addition* and *point subtraction* have nearly identical execution times. Normal Basis *point doubling* appears to take about the same time as *point addition*. Theoretically, *point doubling* takes an extra *squaring* relative to *point addition* as analyzed before. However, in NB, the time for *squaring* is negligible. This results in the roughly equal timings between *point doubling* and *point addition*. Based on this reasoning, SM = (fs – 1) *D + (fs/2 –1) A = (1.5fs –2) A. For example, for fs = 155, SM = (1.5 * 155 –2) A = 231 A. Our measurements show the same order of magnitude for this correlation between *scalar multiplication* and *point addition*.



Since *scalar multiplication* plays an important role in the overall performance, the performance of *scalar multiplication* is shown in Figure 4.3. SM in NB is about 12% faster on average than SM in PB based on our measurements. One *point addition* (PA) = 1S + 2M + 1I and one *point doubling* (PD) = 2S + 2M +1I. In PB, PA = 3.5 M and PD = 4.1M. Therefore,

SM = (fs – 1) PD + (fs/2 – 1) PA

= (fs – 1) * 4.1M + (fs/2 – 1)* 3.5 M

= (5.85fs – 7.6) M_{PB}.

In NB, PA = PD = 1S + 2M + 1I

= 2M + 1I

= 2M + 0.9M

= 2.9 M_{NB}

Therefore,

SM = (fs – 1) PD + (fs/2 – 1) PA

= (4.45fs – 5.8) M_{NB}

= (4.45fs – 5.8) * 1.18 M_{PB}

= (5.251fs – 6.844) M_{PB} .

The performance advantage of NB over PB can be estimated as:

$(SM_{PB} - SM_{NB})/SM_{PB}$=((5.85fs–7.6)$M_{PB}$ - (5.251fs – 6.844) M_{PB}) / (5.85fs–7.6)M_{PB}

= (0.599fs – 0.756) / (5.85fs – 7.6).

For example, for fs = 155, $(SM_{PB} - SM_{NB})/SM_{PB}$= 10 %.

Our measurements are correlated well with this analysis. The major reason for this result is that the underlying *squaring* in NB runs in negligible time, whereas *squaring* accounts for 60% of *multiplication* in PB. Even though the underlying *multiplication* and *inversion* in PB are slightly faster than in NB, the net effect is that SM in NB is 10 % faster than PB for SM.

5 Elliptic Curve Version of ElGamel

In this section, we report on our implementation and the performance measurements of an EC version and a conventional (non-elliptic curve) discrete logarithm-based version of the ElGamal encryption protocol. There are three major parts of the algorithm that contribute to the overall performance of the protocol. The first part is setting up some key-related data that is used for communications. For example, in the EC version, we need to set up a field size, an elliptic curve, a base point on the EC along with public and private keys on both sides. In the conventional approach, we need to set up a prime number, a primitive root (generator) of

the prime field, as well as public and private keys. Since this part of the algorithm is performed only once and used for later communication between both sides, we assume this information is known and do not address its performance.

The second part is the cost of communication. Since this part of the algorithm relies solely on the performance of a specific network system and varies from system to system, we also omit evaluation of this part. The third part is the cost of data encryption and decryption. Data encryption and decryption are the focus of this research and our analysis is focused on this part of the algorithm.

5.1 Elliptic curve version

The encryption process is implemented as follows:

Input: Point B, P_B; Curve EC; Field_element M.

Output: Point $P_r = rB$; $P = P_M + rP_B$.

Field element r ← random_element.

Point P_r ← Scalar_multiplication (r, B).

Point P' ← Scalar_multiplication (r, P_B).

Point P_M ← embed_message (M).

Point P ← Point_addition($P_M + P'$).

Return P_r, P.

Since Scalar multiplication is much more time-consuming than any other operations in the encryption process, the encryption is expected to run in 2SM time. The decryption process is implemented as follows.

Input: Point B, P, P_r; Curve C; Field_element k_B.

Output: Field_element M.

Point P' ← Scalar multiplication(k_B, P_r).

Point P_M ← Point_subtraction(P, P').

Field element M ← P_M.x.

Return M.

The decryption process is expected to run in SM time since scalar multiplication is much more time-consuming than point subtraction and a simple copy operation. The execution time for the EC version of the ElGamal protocol in PB and NB is shown in Figure 5.1. In both PB and NB, the encryption process takes roughly twice as long as the decryption process. This is consistent with our previous analysis. The encryption in NB is 21.8% faster than the encryption in PB. The decryption in NB is 14.5% faster than the

decryption in PB. Based on our previous measurements for scalar multiplication , NB is 12% faster that that in PB. We expect a 24% enhancement for encryption and a 12% enhancement for decryption for NB over PB. Our testing results show the same order of magnitude for this correlation. The variations are attributed to the implementation and random testing errors.

5.2 Conventional version

The conventional discrete logarithm-based ElGamal encryption process uses the same mathematical operations that a protocol like Diffie-Hellman would use. It is implemented as follows.

Input: Big_integer p, g, y, M.

Output: Big_integer a = g^k mod p; b = y^kM mod p.

Big_integer k ← random_integer.

While gcd(k, p-1) ≠ 1 do:

k ← random_integer.

a ← g^k mod p.

b' ← y^k mod p.

b ← b' M mod p.

Return a, b.

Since the modular exponentiation (ME), x^k mod p, for big integer k is much more time-consuming than any other operations in this protocol, it is expected that the encryption takes 2 ME times.

The decryption process is implemented as follows.

Input: Big_integer a, b, x, p.

Output: Big_integer M = b / a^x mod p.

Big_integer a' ← a^x mod p.

Big_integer b' ← a'^{-1} mod p.

M ← b / b' mod p.

Return M.

This protocol is expected to run in ME time owing to the same reasoning applied above.

It should be noted that there is no efficient method to find a primitive root g (generator) of a prime p field for a very large p. Fortunately, there are two well-known generator / prime pairs [30] available for our test as shown in Table 5.2. One prime is

768 bits, the other is 1024 bits. The performance of the conventional protocol over these two prime fields provides information for our comparisons.

The execution time for the conventional protocol over these two prime fields is shown in Table 5.3. As shown in the table, (encryption, decryption) takes (13.1s, 6.6s) and (29.8s, 15.2s) for the prime field of 768 and 1024 bits respectively. Since the finite fields over these two field sizes are neither Type I nor Type II ONB, and we do not find trinomials for PB over these two field sizes, we cannot make a direct comparison among these protocols. However, we can estimate the time for the corresponding EC version of the protocols on these two field sizes through curve fitting. The time for (encryption, decryption) in PB are extimated to be (32s, 16s) and (66s, 33s) respectively over 768 and 1024 bit fields.

Table 5.3 Timings of CDLBV of ElGamal Protocol.

KEY SIZE	768	1024
Encryption (s)	13.1	29.78
Decryption (s)	6.64	15.23

The timings for (encryption, decryption) in NB are (28s, 14s) and (50s, 25s) respectively over 768 and 1024 bit fields. These data are summarized in Table 5.4.

Table 5.4 Comparison of the EC Version with Conventional DLBV for the ElGamal Protocol.

KEY SIZE	CONVENTIONAL DLBV		EC USING PB		EC USING NB	
	ENCR	DECR	ENCR	DECR	ENCR	DECR
768	13.1	6.64	32	16	28	14
1024	29.78	15.23	66	33	50	25

These results indicate that the conventional protocol is actually about twice as fast as the EC version of the protocols over the same field size. However, the same key size (field size) represents different security levels for different protocols. We need to compare the performance of these protocols under the same security levels. To do this, we calculate key sizes in ECC that correspond to the key size of 768 and 1024 bits in the conventional protocol according to the formula derived from [6]:

$$K_{ECC} = 4.91\,(K_{CONV})^{0.33}\,(\mathrm{Ln}\,(K_{CONV}\,\mathrm{Ln}(2)))^{0.67}$$

For K_{CONV} = 768 bits, K_{ECC} = 151; for K_{CONV} = 1024 bits, K_{ECC} = 173 bits. Since key sizes of 151 and 173 bits are not available because of the non-existence of either trinomial in PB or ONB in NB, we use the results for 155, 183 bit key sizes in PB, and the results for 155, 174 bit key sizes in NB to compare with the results of 768 and 1024 bit key size in conventional protocols. Note that under such circumstances, the EC version represents slightly higher security levels than the corresponding conventional protocols. The comparisons are given in Table 5.5. It is clear that the EC version under both PB and NB performs much faster (over 50 times) than conventional protocols.

Table 5.5 Comparison of CDLBV ElGamal with EC Version of ElGamal Under Same Security Level.

KEY SIZE	**CDLBV**		**EC USING PB**		**EC USING NB**	
	ENCR	**DECR**	**ENCR**	**DECR**	**ENCR**	**DECR**
768	13.1	6.64	0.3	0.139	0.248	0.123
1024	29.78	15.23	0.46	0.212	0.357	0.179

6 Conclusions

We have implemented and evaluated finite field arithmetic, elliptic group operations, and an EC version of the ElGamal encryption protocol over $GF(2^n)$ for n = 100 to 1279 bits using both polynomial and normal basis representations in software using a Pentium II Linux platform together with a conventional discrete logarithm-based version of the ElGamal encryption protocol.

Finite field arithmetic operations include addition, subtraction, multiplication, squaring, and inversion. Our results show that both addition and subtraction can be implemented very efficiently and the differences between PB and NB are small. Multiplication in PB using a trinomial as the irreducible polynomial is 17% faster than multiplication in NB. Squaring, a special case of multiplication, can be implemented 40% faster than multiplication in PB, 100% faster than multiplication in NB. Inversion in NB using a combination of basis conversion and the "almost inverse" method runs much faster than that using Fermat's theorem-based approach, but 10% slower than inversion in PB with the "almost inverse" method.

Elliptic group operations include point negation, point addition, point subtraction, point doubling, and scalar multiplication. Our results show that point negation can be implemented very efficiently in both PB and NB and the time is small compared to other group operations. Point addition and subtraction runs in similar time in both PB and NB. Point doubling runs 18.7% slower than point addition in PB. It runs in similar time to point addition in NB. Scalar multiplication runs 12% faster in NB than in PB probably because of the efficient implementation of the underlying squaring in NB.

Our evaluation also shows that EC version of ElGamal encryption runs 22% faster in NB than in PB. An EC version of ElGamal decryption runs 15% faster with NB than with PB. EC versions of the ElGamal encryption / decryption protocols are more

than twice as slow as conventional discrete logarithm-based ElGamal protocol under the two tested field size of 768 and 1024 bits. When compared at the same level of security, EC version of ElGamal protocols runs more than 50 times faster than the conventional ElGamal protocol.

6.1 Future Work

We have demonstrated that EC implementations of cryptographic systems have obvious performance advantages over conventional systems and the trade-off between PB and NB implementations. Possible areas for future work include:

- Investigation of finite filed arithmetic in software for general prime field GF(p) under both PB and NB.
- Improvement on the performance of EC protocols for large field sizes (> 500 bits).
- Improvement of the implementation of scalar multiplication.
- Faster field multiplication algorithm in both PB and NB.
- Building a library of irreducible trinomials over a broad range of field sizes (as large as 2048 bits).
- Performance of EC versions of other cryptographic systems (such as digital signature).

7 Bibliography

[1] B. Schneier, "Applied Cryptography," 2nd ed., J. Wiley & Sons, Inc., 1996.

[2] N. Koblitz, "Elliptic Curve Cryptosystems," Math. Compu. Vol. 48, No. 177, Jan. 1987, pp.203-209.

[3] V. S. Miller, "Use of Elliptic Curves in Cryptography," Advances in Cryptology-CRYPTO'85, LNCS 218, Springer-Verlag, 1986, pp.417-426.

[4] R. L. Rivest, A. Shamir, and L. M. Adleman, "A method for Obtaining Digital Signatures and Public-Key Cryptosystems," Communications of the ACM, v. 21, n. 2, Feb 1978, pp. 120-126.

[5] W. Diffie and M.E. Hellman, "New directions in cryptography," IEEE Transactions on Information Theory, IT-22: pp. 644-654, 1976.

[6] I. Blake, G. Seroussi, and N. Smart, Elliptic Curves in Cryptography, Cambridge University press, 1999.

[7] P. Buhler, H.W. Lenstra, and C. Pomerance. "The development of the number field sieve," Volume 1554 of Lecture Notes in Computer Science, Springer-Verlag, 1994.

[8] "The Elliptic Curve Cryptosystem: Remarks on the Security of the Elliptic Curve Cryptosystem," a white paper by Certicom Corporation, 1997.

[9] C. H. Lim and H. S. Hwang, "Fast Implementation of Elliptic Curve Arithmetic in GF (pn)," Public Key Cryptography-CRYPTO'2000, LNCS 1751, Springer-Verlag, 2000, pp.405-421.

[10] N. P. Smart, "The Discrete Logarithm Problem on Elliptic Curves of Trace One," J. Cryptology (1999) 12: 193-196.

[11] T. Kobayashi, H. Morita, K. Kobayashi, and F. Hoshino, "Fast Elliptic Curve Algorithm Combining Frobenius Map and Table Reference to Adapt to Higher Characteristic," Advances in Cryptology-CRYPTO'99, LNCS 1592, pp.176-189.

[12] K. Koyama, Y. Tsuruoka, N. Kumihiro, "Modulus Search for Elliptic Curve Cryptosystems," Advances in Cryptology-CRYPTO'99, LNCS 1716, pp.1-7.

[13] Y. Han, P. Leong, P. Tan, and J. Zhang, "Fast Algorithms for Elliptic Curve Cryptosystems over Binary Finite Field," Advances in Cryptology-CRYPTO'99, LNCS 1716, pp.75-85.

[14] K. Sakurai and H. Shizuya, "A Structural Comparison of the Computational Difficulty of Breaking Discrete Log Cryptosystems," J. Cryptology (1998) 11: 29-43.

[15] A. Schroeppel, H. Orman, S. O. Malley, and O. Spatschek, "Fast key Exchange with Elliptic Curve Systems," Advances in Cryptology-CRYPTO'95, LNCS 963, Springer-Verlag, 1995, pp. 43-56.

[16] N. P. Smart, "Elliptic Curve Cryptosystems over Small Fields of Odd Characteristic," J. Cryptology (1999) 12: 141-151.

[17] J. Guajardo and C. Paar, "Effic ient Algorithms for Elliptic Curve Cryptosystems," Advances in Cryptology-CYYPTO'97, LNCS 1462, pp.342-356.

[18] D. M. Gordon, "A Survey of fast exponentiation methods," J. Algorithmss, 27, 1998, pp.129-146.

[19] T. Itoh and S. Tsujii, "A Fast Algorithm for Computing Multiplication Inverses in GF(2m) Using Normal Bases," Information & Computation, 78, 1988, pp.171-177.

[20] J. A. Solinas, "An Improved Algorithm for Arithmetic on a Family of Elliptic Curves," Advances in Cryptology-CRYPTO'97, LNCS 1294, Springer-Verlag, 1997, pp. 357-371.

[21] E. D. Win, A. Bosselaers and S. Vandenberghe, "A Fast Software Implementation for Arithmetic Operations in GF(2n), Advances in Cryptology-ASIACRYPTO'96, LNCS 1163, Springer-Verlag, 1996, pp. 65-76.

[22] E. R. Berlekamp, "Algebraic Coding Theory," NY, McGraw-Hill, 1968.

[23] D. E. Knuth, "Seminumerical Algorithms," MA, Addison-Wesley, 1981.

[24] N. Koblitz, A Course in Number Theory and Cryptography, 2nd Eds., Springer-Verlag, 1994.

[25] N. Koblitz, Introduction to Elliptic Curves and Modular Forms, 2nd Ed., Springer-verlag, 1993.

[26] D. F. Lawden, Elliptic Functions and Applications, Springer-Verlag, 1989.

[27] A. Menezes, T. Okamoto, and S. Vanstone, "Reducing Elliptic Curve Logorithms to Logorithms in a Finite Field," IEEE Transactions on Information Theory, 39(5): 1639-1646, September, 1993.

[28] R. Lercier and F. Morain, "Counting the Number of Points on Elliptic Curves over Finite Fields: Strategies and Performance," in EUROCRYPT'95, Berlin: Springer-Verlag, 1995.

[29] R. C. Mullin, I. M. Onyszchuk, S. A. Vanstone, and R. M. Wilson, "Optimal Normal Bases in GF(p^n)," Discrete Applied Mathematics, No. 22, pp. 149-161, 1988/89.

[30] http://www.dns.net/dnsrd/rfc/rfc2539.html#Appendix A.