2005

ElGamal-type signature schemes in modular arithmetic and Galois fields

Gerard Nealon

ElGamal-Type Signature Schemes in Modular Arithmetic and Galois Fields

Gerard J. Nealon
Masters Project
Department of Computer Science
Rochester Institute of Technology
Rochester, NY USA
gjn3855@cs.rit.edu

July 15, 2005

Committee Members:

Chair: Stanisław Radziszowski

Date / Sign

Reader: Sidney Marshall

Date / Sign

Observer: Zack Butler

Date / Sign

Contents

Abstract

A digital signature is like a handwritten signature for a file, such that it ensures the identity of the person responsible for the file and prevents any unauthorized changes to the original file. Digital signatures use the same technology as most public key cryptosystems in which there is a public and private key. Most mathematical operations are done over a field $\mathbb{Z}_p$ where p is some large prime. It is possible to do the same operations over other finite fields. This paper explains and studies the different finite fields that can be used as well as ways to implement and experiment with them. It turns out that operations over $\mathbb{Z}_p$ run the fastest, but with polynomial basis in a close second. Normal basis did not prove to be efficient at all. These results turned out to be against most claims of others, especially in hardware implementations. Large integer libraries are so efficient and fast that is was hard to beat the times with custom bit manipulation structures. Various secure signature schemes have proven to be practical and it is likely that they will be used much more in the near future in many applications.

1 Introduction

Digital signatures were first invented in the 1970s, they were extremely popular since they are better than a handwritten signature, unforgeable, and uncopyable. Today, they are a vital component to business around the world. Numerous laws, state and now federal, have codified digital signatures into law. An example of a fully functional program that signs documents which is widely used today is PGP, also know as "Pretty Good Privacy". PGP has a beautiful interface in which the user can select any file he/she wishes and with a certain passphrase a signature is generated using a secret key. There are some people out there who highly disagree with digital signature. Some say it is way too easy for a rogue program to infiltrate PGP and send

private information to whomever. Some people say there is no way of verifying that "Alice" really signed the message since the computer is really the one who is doing the calculations. I do not agree with people who make these accusations since factors such as those are a problem in any situation. With any cryptographic application there exists the threat of a third person intercepting information, especially with the abundance of spyware on the web. It is vital that users check and see if the software they are using is authentic before they attempt to send cryptographic data through it.

There are many ways of implementing digital signatures, some more efficient than others, some easier than others. In my project, I am going to implement ElGamal-type digital signatures along with two different ways of implementing them; modular arithmetic and Galois fields. I will also discuss the use of hash functions within digital signatures and some of the attacks and weaknesses that were developed when hash functions and digital signatures were brought together. There are two main ways of representing Galois fields, polynomial basis representation and normal basis representation both of which will be implemented in this project. I will go into the theory behind finite fields and focus mainly on the efficiency differences between polynomial basis representation and normal basis representation of Galois fields, along with irreducible polynomials that are required within. After understanding the ElGamal signature scheme and Galois fields independently, I will show how I will implement the ElGamal signature scheme over a Galois field.

2 Digital Signatures

When people are asked to authenticate themselves, such as at a bank or a food store, they are often asked to sign their signature on a piece of paper. Once a person's signature is on paper, anybody can compare the signature to another signature and verify if the two signatures are the same or not.

Unfortunately this method is often used in our society, regardless of it's highly insecure nature. Handwritten signatures have many problems; forging someone else's signature is easily accomplished and frequently people do not sign their name exactly the same way each time. What if we wanted to attach our signature to a document stored on a computer? There are many cases when people would like to know who is responsible for a particular document, and they would also like to be assured that the requested document was not tampered with by a nonauthoritative entity. Digital signatures are a means of signing a document with all of the properties of a handwritten signature plus additional security. A digital signature is unforgeable without unreasonable resources. This means that there can be many different signatures for a given plaintext because of the random nature of the algorithm. It is a computational proof that the signature is based on the document. Digital signatures convince the receiver of the document that it has been signed by the claimed signer. A digital signature is not reusable. Once a document is signed, it can't be moved to another document. If the document is altered or the signature is moved to another document then the signature is no longer valid.

2.1 Primitive Operations Needed

There are some basic operations that are needed in order to implement the ElGamal signature scheme that operates over binary Galois fields. For example you must be able to add, subtract, multiply, exponentiate, and perform multiplicative inverse. Also you must be able to manipulate a large number of bits and operate with them. Some basic operation needed to operate with these bits are: shifting, XOR, and copying.

2.2 Different Domains

The major part of this project is not about digital signatures, it is more about the implementation that allowed me to perform a digital signature. The work presented here can easily be ported to other domains such as Elliptic Curve, RSA, DSA, AES and any others that need operations over fields.

3 Finite Fields

In abstract algebra, a finite field or Galois field (named in honor of Evariste Galois) is a field that contains only finitely many elements. Finite fields are important in number theory, algebraic geometry, Galois theory, cryptography, and coding theory. Since every field of characteristic 0 contains the rationals and is therefore infinite, all finite fields have prime characteristic. However, the converse is not true. There exist infinite fields of prime characteristic.

The multiplicative group of every finite field is cyclic. This means that if $\mathbb{F}$ is a finite field with q elements, then there always exists an element $x \in \mathbb{F}$ such that $\mathbb{F} = 0, 1, x, x^2, ..., x^{q-2}$.

The element x is not unique. If we fix one, then for any non-zero element a in $\mathbb{F}_q$, there is a unique integer $n \in 0, ..., q-2$ such that $a = x^n$. The value of n for a given a is called the discrete log of a. In practice, although calculating x^n is relatively trivial given n, finding n for a given a is a computationally difficult process, and so has many applications in cryptography.

A finite field, $\mathbb{F}_q$, consists of a set of elements $\mathcal{F}$ and two binary operations on $\mathcal{F}$ called addition and multiplication that satisfy some arithmetic properties. These properties are defined as such:

- Addition in $GF(p)$ is described as, given $a, b \in \mathbb{F}_p$, then $a + b = z$,

where z is $a + b \pmod{p}$

- Multiplication in $GF(p)$ is described as, given $a, b \in \mathbb{F}_p$, then $a \times b = z$, where z is $a \times b \pmod{p}$
- Inversion in $GF(p)$ is described as, given $a \in \mathbb{F}_p : a > 0$, the inverse of a, denoted as a^{-1}, is c such that $a \times c \pmod{p} = 1$

If q is prime, then $\mathbb{Z}_q$ is the same as $\mathbb{F}_q$. A finite field containing q elements also exists when $q = p^m$, where p is prime. For efficiency and security reasons, most cryptosystems and digital signature schemes operate over $\mathbb{F}_p$ where p is prime or $\mathbb{F}_{2^m}$. These fields, discovered by Galois, have order or number of elements equal to p and 2^m, respectively. $\mathbb{F}_p$, $\mathbb{F}_{2^m}$, and $\mathbb{F}_{p^m}$ where p is prime are called Galois fields, or also known as $GF(p)$, $GF(2^m)$, and $GF(p^m)$, respectively. In the next couple of sections, I will talk about the two most commonly used finite fields.

3.1 Finite Field $GF(p)$

If p is a prime, the integers mod p form a field with p elements, denoted as $\mathbb{Z}_p$, $\mathbb{F}_p$ or $GF(p)$. Every other field with p elements is isomorphic to this one.

Galois Field $GF(p)$ arithmetic is easy to implement. In my project I will be using a large number package so implementing algorithms in $GF(p)$ will be very easy. Galois Field $GF(p)$ arithmetic, or doing operations mod p, result in field elements in the range $\{0, 1, ..., p-1\}$. Any operation in this field will result in one of the field elements.

3.2 Finite Field $GF(2^m)$

If $q = p^n$ is a prime power, then there exists up to isomorphism exactly one field with q elements, written as $\mathbb{F}_q$ or $GF(q)$. This field can be constructed by finding an irreducible polynomial, $f(x)$ of degree n with co-

efficients in $GF(p)$. With this we can define the field $GF(q)$ as such, $GF(q) = GF(p)[x]/ < f(x) >$. This definition can be broken down to explain the field better. $GF(p)[x]$ stands for the ring of all polynomials with coefficients in $GF(p)$. The $< f(x) >$ part of the definition stands for the ring formed by reducing all polynomials by $f(x)$. The quotient is meant in the sense of factor rings, the set of polynomials with coefficients in $GF(p)$ on division by $f(x)$.

The most commonly used nonmodular finite field in cryptographic applications is the Galois field $GF(2^m)$. There exist many algorithms for representing elements and computing operations in this field very efficiently. The Galois field $GF(2^m)$ is called a binary finite field because it can be represented in its multiplication table as $\{0,1\}^m$ elements. Different algorithms make use of this binary format to manipulate numbers rapidly. For example, addition in this field is nothing more that XORing the array representation of field elements. Another way of representing $GF(2^m)$ is through a polynomial or normal basis. In the subsequent sections I will go into more detail of such representation of binary fields.

3.3 Modular Arithmetic

I used GNU's MP library to perform the large integer arithmetic. This library is very fast and provides very large precision. The library is fairly easy to understand, but when you start adding a lot of operations into a algorithm, the algorithm becomes harder to understand and read. To help with this problem, I wrote a wrapper class with many overloaded operations. This allowed me to do simple operations, which are visibly easier to understand. For example, in order to add two numbers using the library you would have to call the function:

```
mpz_add_ui(mpz_t c, mpz_t a, mpz_t b)
```

with the wrapper class it becomes as easy as doing $c = a + b$

3.4 Polynomial Basis Representations of Galois Fields

Given a irreducible polynomial over $\mathbb{Z}_p$ of the form:

$$f(x) = x^m + a_{m-1}x^{m-1} + ... + a_2x^2 + a_1x + a, \text{ where } a_i \in \mathbb{Z}_p$$

If we choose a polynomial of this form such that it is irreducible, we can use it to define a finite field. It defines a finite field because a irreducible polynomial is analogous to a prime number p which also defines a finite field $\mathbb{Z}_p$. All operations in this polynomial algebra will be reduced modulo $f(x)$, therefore, we consider $f(x)$ to be the *reduction polynomial*. Finite fields, such as $\mathbb{F}_{p^m}$ with a polynomial basis are often denoted as, $\mathbb{Z}_p\ [x]/(f(x))$, where all coefficients are reduced modulo p and the polynomials are reduced modulo $f(x)$. In the case of a binary finite field 2^m, all coefficients are reduced modulo 2 therefore we can represent any field element as a binary string of length m.

3.4.1 Reduction Polynomials

Reduction polynomials need to be chosen carefully and also must be irreducible. A irreducible polynomial is a polynomial $f(x)$ such that it can't be factored into two polynomials $f_1(x)$ and $f_2(x)$ of smaller degrees. A trinomial over $\mathbb{F}_{p^m}$ is a polynomial of the form $a_1x^m + a_0x^k + 1$, where $1 \leq k \leq m-1$ and $a_i \leq p-1$. A pentanomial polynomial over $\mathbb{F}_{p^m}$ is a polynomial of the form $a_3x^m + a_2x^{k_3} + a_1x^{k_2} + a_0x^{k_1} + 1$, where $1 \leq k_1 \leq k_2 \leq k_3 \leq m-1$ and $a_i \leq p-1$. The following are the criteria for choosing a reduction polynomial over $\mathbb{F}_2$:

- If there exists an irreducible trinomial of degree m over $\mathbb{F}_2$, then the reduction polynomial $f(x)$ is recommended to be an irreducible trinomial of degree m over $\mathbb{F}_2$. To maximize the chances for interoperability, ANSI X9.62 recommends that the trinomial used should be x^m+x^k+1 for the smallest possible k such that $0 < k < m$.

- If there does not exist an irreducible trinomial of degree m over $\mathbb{F}_2$, then the reduction polynomial $f(x)$ is recommended to be an irreducible pentanomial of degree m over $\mathbb{F}_2$. To maximize the chances for interoperability, ANSI X9.62 recommends that the pentanomial used should be $x^m + x^{k_3} + x^{k_2} + x^{k_1} + 1$ chosen according to the following criteria such that $0 < k^1 < k^2 < k^3 < m$:

  1. k_3 is as small as possible
  2. For this particular value of k_3, k_2 is as small as possible
  3. For these particular values of k_3 and k_2, k_1 should be as small as possible

3.4.2 Testing for Irreducibility

A polynomial can't be used to define a finite field $\mathbb{F}_{p^k}$ unless it is irreducible over $\mathbb{Z}_p$. Therefore, it is important to have a algorithm that can test to see if a polynomial is irreducible. Although I will not demonstrate how this works, it can be shown that there is at least 1 irreducible primitive polynomial for a finite field $\mathbb{F}_{p^m}$. An irreducible polynomial $f(x) \in \mathbb{Z}_p[x]$ of degree m is called a primitive polynomial if x is a generator of $\mathbb{F}^*_{p^m}$, the multiplicative group of all the non-zero elements in $\mathbb{F}_{p^m} = \mathbb{Z}_p[x]/(f(x))$. [1]

According to the *Applied Handbook of Cryptography* a good algorithm for testing irreducibility of a monic polynomial is as follows:

INPUT: a prime p and a monic polynomial $f(x)$ of degree m in $\mathbb{Z}_p[x]$
OUTPUT: an answer to the question: "Is $f(x)$ irreducible over $\mathbb{Z}_p$?"

1. Set $u(x) \leftarrow x$
2. For i from 1 to $\lfloor \frac{m}{2} \rfloor$ do the following:
   (a) Compute $u(x) \leftarrow u(x)^p \pmod{f(x)}$

(b) Compute $d(x) = gcd(f(x), u(x) - x)$

(c) If $d(x) \neq 1$ then return "reducible"

3. Return "irreducible"

This algorithm can be iterated many times while changing the value of $f(x)$ to find a irreducible polynomial.

3.4.3 Addition and Subtraction

Since this project uses binary Galois fields, $\mathbb{F}_{2^n}$, all coefficients are reduced modulo 2. Addition modulo 2 is just exclusive-or since $1+1 = 0 \mod 2, 0+1 = 1$, and $0 + 0 = 0$. We can add two polynomials extremely fast using a simple XOR operation. And since there is no carry to propagate, it is even faster than ususal addition. Polynomial basis, using binary fields, is very interesting when is comes to addition and subtraction because they turn out to be exactly the same. This is obvious since $1-1 = 0$ and $1+1 = 0 \mod 2$.

3.4.4 Multiplication

Multiplication in polynomial basis is easy. Multiplying two polynomials is done the same way as we were taught in grade school; every term in one polynomial is multiplied by every term in the other. But since we are using polynomial basis to represent a finite field, we have to reduce the coefficients. For example:

$$(x^4 + x^2 + 1) * (x^8 + x^6 + x^4 + x^3 + 1)$$

In this example we would multiply x^4 by $(x^8+x^6+x^4+x^3+1)$ first. The we would take the answer and add it to $x^2*(x^8+x^6+x^4+x^3+1)$ which in turn would be added to $1*(x^8+x^6+x^4+x^3+1)$. For a computer this amounts to shifting and XORing. Lets call the polynomial $(x^8+x^6+x^4+x^3+1)$ A for

simplicity. An algorithm can be devised using nothing more than addition and left shifts. Lets call the shift left operator $<$. For example,

$$(x^4 + x^2 + 1) * (x^8 + x^6 + x^4 + x^3 + 1) = (A < 4) + (A < 2) + (A < 0)$$

3.4.5 Multiplication Improved

The "square and multiply" algorithm explained in the previous section is simple and fast to implement. But there exist many different multiplication algorithm that can be used for polynomial basis. I tried many different algorithms and found one that worked really well. Also this algorithm operates more on the WORDs than bits of a polynomial.

INPUT: $a = (A_{s-1}...A_0)$, $b = (B_{s-1}...B_0)$, $f = (F_{s-1}...F_0)$
OUTPUT: $c = (C_{s-1}...C_0) = ab \mod f$

Set $T_i \leftarrow 0$; $i = 0, ..., 2s - 1$
for j from $w - 1$ down to 0 do
    for i from 0 to $s - 1$ do
        if $a_{iw+j} \neq 0$ then
            for k from 0 to $s - 1$ do
                Set $T_{k+1} \leftarrow T_{k+i} \oplus B_k$
    if $j \neq 0$ then $T \leftarrow xT$
Set $c \leftarrow T \mod f$
return (c)

3.4.6 Squaring

Squaring polynomials in a binary field is a linear operation. The mathematical formula for squaring a polynomial is as follows:

$$(a+b)^2 = a^2 + 2ab + b^2$$

but since we reduce all coefficients modulo 2, the inside term gets canceled and we are left with

$$a^2 + b^2$$

Therefore squaring a polynomial in a binary field is no more that just squaring each term separately. In terms of bit strings, there exists a nice simple algorithm for doing so. Lets take the polynomial $x^5 + x^3 + x^2 + 1$ represented as 101101. If we interlace 0's between each bit we end of with the bit string 10001010001 with is the polynomial $x^{10} + x^6 + x^4 + 1$. Unfortunately there is not a nice algorithm for interlacing zero's between bits, so a square lookup table had to be created. This table converts a 16bit number to its 32bit square. Squaring polynomials using this method drastically reduced the time spent in the exponentiation algorithm.

3.4.7 Reduction Algorithm

A fast reduction algorithm is very important in polynomial basis arithmetic. Since after every operation a reduction is needed, the efficiency of the reduction algorithm is greatly noticed. The reduction algorithm that I choose to implement makes use of either a trinomial or pentanomial reduction polynomial. Lets take for an example the reduction polynomial:

$$x^{155} + x^{62} + 1$$

The reduction of any polynomial $\mod x^{155} + x^{62} + 1$ proceeds by reducing each term in the polynomial by the trinomial reduction polynomial and subtracting it from the result. The is easily done by a sequence of shifts and XOR's. In order to understand this concept it is important to see that

$$x^{155} \equiv x^{62} + 1 \mod F(x)$$

or more generally

$$x^{n} \equiv x^{n-93} + x^{n-155} \mod F(x)$$

In this example, 93 bits of a polynomial can be reduced at a single time by replacing each term in the polynomial by it's congruent two-term expression. In other words, we can take the upper 93 bits and add it into the original polynomial right shifted by 93 bits and right shifted by 155 bits. This algorithm can be repeated over and over until the degree is less than that of the reduction polynomial.

In terms of implementation, ripping out 93 bits of an expression is more overhead than wanted. It is best to only use WORDS at a time lowering the degree of the polynomial by 32 each time. It is important to try and use a trinomial for the reduction polynomial because you only have to xor twice. It is also worth noting the degree of the second term. If the difference between the upper term and the second term is less than 32 then more work is needed to reduce the polynomial. For example, if I choose to use x^{150} instead of x^{62} I would only be able to reduce the polynomial by 5 degrees each time.

3.4.8 Montgomery Algorithm

Montgomery reduction is a technique which allows efficient implementations of modular multiplication without explicitly carrying out the classical modular reduction step. [1] Let m be a positive integer, and let R and T be integers such that $R > m$, $gcd(m, R) = 1$, and $0 \leq T < mR$. A method is described for computing $TR^{-1} \mod m$ without using the classical modular multiplication algorithm. $TR^{-1} \mod m$ is called a Montgomery reduction of T modulo m with respect to R. With a suitable choice of R, a Montgomery reduction can be efficiently computed. [1]

3.5 Normal Basis Representations of Galois Fields

Another way of representing Galois fields is through the so-called normal basis. A normal basis of $\mathbb{F}_{2^m}$ over $\mathbb{F}_2$ is of the form $\{\beta, \beta^2, \beta^{2^2}, ..., \beta^{2^{m-1}}\}$ where $\beta \in \mathbb{F}_{2^m}$. Given any element $a \in \mathbb{F}_{2^m}$, it can be represented in the form $a = \sum_{i=0}^{m-1} a_i \beta^{2^i}$, where $a_i \in \{0, 1\}$. One of the main advantages of representing elements of a finite field in this form is the efficiency of squaring. Without getting into the complicated mathematics, squaring ends up to be a simple rotation of the vector representation. Although multiplication can be very difficult in this representation, many people use an extension of normal basis called Gaussian normal basis which helps out a great deal in multiplication. The Gaussian normal basis records the complexity of the multiplication operation with respect the the basis. Gaussian normal basis contains two types, Type I and Type II. The type is used in selecting a Gaussian normal basis for finite field 2^m.

Normal Basis arithmetic started off to be one of the most challenging aspects of this project, but in the end was not as bad as I had thought. I had developed a class called NormalBasis which allows me to add, subtract, multiply, square, and invert elements in a field F_{2^n}. This class will only accept Type I or Type II Optimal Normal Basis. Keeping it this way allowed the multiplication algorithm to run a lot faster.

3.5.1 Type I ONB

A Type I optimal normal basis exists for the field F_{2^n} with the following conditions:

- $n + 1$ is prime
- 2 is primitive in Z_{n+1}

This algorithm I wrote will test for a Type I ONB:

```
bool NormalBasis::isTypeI(){
    BigInt two = 2;
    if (is_prime(m + 1))
        if (isPrimitive(two, m + 1))
            return true;
    return false;
}
```

3.5.2 Type II ONB

A Type II optimal normal basis exists for the field F_{2^n} with the following conditions:

- $2n + 1$ is a prime p and either
- 2 is primitive modulo p or
- $p \equiv 3(mod 4)$ and the multiplicative order of 2 modulo p is n

  This algorithm I wrote will test for a Type II ONB:

```
bool NormalBasis::isTypeII(){
    BigInt test = (m * 2) + 1;
    BigInt two = 2, exp;

    if (is_prime(test)){
        if (isPrimitive(two, test)){
            return true;
        }
        else{
            if ((test % 4) == 3){
                if (order(two, test) == m)
                    return true;
                else
                    return false;
            }
            else{
                return false;
            }
        }
    }
    else{
```

```
            return false;
        }
    }
```

3.5.3 Addition and Subtraction

Addition and subtraction are very easy in a normal basis. Suppose A, B are elements in the field F_{2^n} Then A and B can be represented like this:

$$A = \sum_{i=0}^{n-1} a_i \beta^{2^i}$$
$$B = \sum_{j=0}^{n-1} b_j \beta^{2^j}$$

Then addition is defined by:

$$A + B = \left(\sum_{i=0}^{n-1} a_i \beta^{2^i}\right) + \left(\sum_{j=0}^{n-1} b_j \beta^{2^j}\right)$$

Since every power in β is linearly independent, addition could be rewritten as:

$$A + B = \sum_{i=0}^{n-1} (a_i + b_i) \beta^{2^i}$$

Therefore since the a_i and b_i are added modulo 2, the implementation just becomes a bit-wise exclusive-OR operation. Also, since 1 and -1 are the same modulo 2 subtraction is the same as addition.

3.5.4 Squaring

Squaring in normal basis is very fast and efficient. Since every power of β is linearly independent squaring a number in normal basis can be computed doing the following:

$$\begin{aligned} A^2 &= \left(\sum_{i=0}^{n-1} a_i \beta^{2^i}\right)^2 \\ &= \sum_{i=0}^{n-1} a_i \left(\beta^{2^i}\right)^2 \\ &= \sum_{i=0}^{n-1} a_i \beta^{2^{i+1}} \\ &= \sum_{i=0}^{n-1} a_{i-1} \beta^{2^i} \end{aligned}$$

And since,

$$\left(\beta^{2n-1}\right) = \beta^{2n} = \beta$$

Squaring an element in the field F_{2^n} turns out to be a simple circular shift left.

3.5.5 Multiplication

Multiplication in normal basis elements can be very expensive. Multiplication over the field F_{2^n} can be defined as follows:

$$A = \sum_{i=0}^{n-1} a_i \beta^{2^i}$$
$$B = \sum_{j=0}^{n-1} b_j \beta^{2^j}$$

and

$$C = A \times B = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} a_i b_j \beta^{2^i} \beta^{2^j} = \sum_{i=0}^{n-1} c_i \beta^{2^i}$$

so

$$\beta^{2^i} \beta^{2^j} = \sum_{k=0}^{n-1} \lambda_{ijk} \beta^{2^k}$$

where $\lambda_{ijk} \in \{0, 1\}$ Therefore multiplication can be written as:

$$c_k = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \lambda_{ijk} a_i b_j$$

or

$$c_k = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \lambda_{ij0} a_{i+k} b_{j+k}$$

The trick to multiplication is the λ matrix which needs to be generated. The λ matrix is generated differently depending of wether it is a Type I or a Type II ONB.

3.5.6 Type I Lambda Matrix Generation

For the finite field F_{2^n} the lambda matrix will consist of a 2-dimensional array of size n x n, consisting of 0's and 1's. For every $[i, j]$ in the matrix,$[i, j] = 1$ iff i and j satisfy one of the two congruences:

$$2^i + 2^j \equiv 1 \mod n + 1$$
$$2^i + 2^j \equiv 0 \mod n + 1$$

Since $2^i + 2^j = 2^j + 2^i$ is it only necessary to compute half of the matrix.

3.5.7 Type II Lambda Matrix Generation

The lambda matrix for a Type II ONB is the same size and consists of the same type elements as the Type I lambda matrix. The only difference is how it is computed. For every $[i, j]$ in the matrix,$[i, j] = 1$ iff i and j satisfy one of the 4 congruences:

$$2^i + 2^j \equiv 2^k \mod 2n + 1$$
$$2^i + 2^j \equiv -2^k \mod 2n + 1$$
$$2^i - 2^j \equiv 2^k \mod 2n + 1$$
$$2^i - 2^j \equiv -2^k \mod 2n + 1$$

3.5.8 Multiplication and Lambda Matrix Improvement

The previously presented multiplication algorithm ran extremely slow. Almost 100 times that of everything else. This pushed me into doing more research on a better multiplication algorithm. The improved algorithm not only runs in $O(n^2)$ as opposed to $O(n^3)$, is saves memory also. Since the lambda matrix of an optimal normal basis contains $2m - 1$ non-zero entries, it us unnecessary to construct an $M \times M$ matrix. Since every row in the lambda matrix contains two non-zero entries, except the first row, we can use this to our advantage. Instead of the traditional matrix presented before, all we need is two tables containing the indices's of the non-zero

lambda matrix entries. If we call these tables t1 and t2, we can reconstruct our multiplication algorithm like the following:

$$c_k = (a_k \cdot b_{t_1[0]+k}) \oplus \left(\bigoplus_{i=1}^{m-1} \left[a_{k+i} \cdot \left(b_{t_1[i]+k} \oplus b_{t_2[i]+k} \right) \right] \right)$$

3.5.9 Inversion

Inversion of an element x is represented as x^{-1} and is defined as:

$$xx^{-1} \equiv 1 \mod n$$

For any element, $x \neq 0$ in F_{2^n} the inverse of x can be derived from Fermat's Little Theorem as:

$$x^{-1} = x^{2^n-2} = \left(x^{2^{n-1}-1} \right)^2$$

3.6 Field Sizes

The size of a field is very important in cryptographic algorithms. Depending on the level of security, the recommended field size of discrete logarithm based cryptosystems is 1024 bits. In my project, the modulus I plan on testing, for finite field $\mathbb{F}_p$, will be a prime number between 512 bits and 2048 bits. My reasoning for this wide range is for testing purposes only. Likewise, I will test the same algorithms implemented over a binary field, $\mathbb{F}_{2^m}$ where m will range from 512 to 2048. In my project I will only test field sizes up to 512 bits. There were a few reasons why I did not go past 512 bits. First, trying to find a 1024 or 2048 bit prime p in which you can factor $p-1$ proved too difficult for my factoring algorithm. Second, finding the prime factors of the order of the field $GF(2^n)$ in order to test a generator was too difficult for my factoring algoritm.

4 The ElGamal Signature Scheme

The ElGamal cryptosystem can't be used as a signature scheme, although it can be modified to become one. The ElGamal signature is a non-deterministic,

probabilistic signature scheme meaning there are many valid signatures for the same plaintext. In order for the scheme to work, the verification algorithm must be able to validate any of the valid signatures. According to Stinson, the ElGamal signature scheme is defined as follows:

Let p be a prime such that the discrete logarithm problem in $\mathbb{Z}_p$ is intractable, and let $\alpha \in \mathbb{Z}_p^*$ be a primitive element. Let $\mathcal{P} = \mathbb{Z}_p^*$, $\mathcal{A} = \mathbb{Z}_p^* \times \mathbb{Z}_{p-1}$, and define

$$\mathcal{K} = \{(p, \alpha, a, \beta) : \beta \equiv \alpha^a \mod (p)\}$$

The values p, α, and β are the public key, and a is the private key. For $K = (p, \alpha, a, \beta)$, and for a secret random number $k \in \mathbb{Z}_{p-1}^*$, define

$$sig_K(x, k) = (\gamma, \delta)$$

where

$$\gamma = \alpha^k \pmod{p}$$

and

$$\delta = (x - a\gamma)k^{-1} \pmod{p-1}$$

For $x, \gamma \in \mathbb{Z}_p^*$ and $\delta \in \mathbb{Z}_{p-1}$, define

$$ver_K(x, (\gamma, \delta)) = true \Leftrightarrow \beta^\gamma \gamma^\delta \equiv \alpha^x \pmod{p}$$

In the verification part of the scheme, it is easy to prove that this actually works. Take the formula, $\beta^\gamma \gamma^\delta \equiv \alpha^x \pmod{p}$ and substitute γ and β from the defined formulas in the ElGamal scheme. We end up with the formula $\alpha^x \equiv \alpha^{a\gamma + k\delta} \pmod{p}$. We can see that the exponents have to be congruent, modulo $p-1$ in order for the entire equation to be congruent. We are left with the formula, $a\gamma + k\delta = x \pmod{p-1}$, from the exponents. Solving this formula for δ we get $\delta = (x - a\gamma)k^{-1} \pmod{p-1}$, which is the original formula defined in the ElGamal scheme.

5 The Generalized ElGamal Signature Scheme

The ElGamal digital signature scheme, described above in the setting of the multiplicative group $\mathbb{Z}_p^*$, can be generalized in a straightforward manner to work in any finite abelian group G. The algorithm requires a cryptographic hash function $h : \{0,1\}^* \longrightarrow \mathbb{Z}_n$ where n is the number of elements in G. It is assumed that each element $r \in G$ can be represented in binary so that $h(r)$ is defined. For example for $G = \mathbb{F}_{2^m}^*$.

5.1 Key Generation

Each entity selects a finite group G, a generator of G and public and private keys. Each entity A should do the following:

- Select an appropriate cyclic group G or order n, with generator α.
- Select a random secret integer a, $1 \leq a \leq n-1$. Compute the group element $y = \alpha^a$
- A's public key is (α, y), together with a description of how to multiply elements in G. A's private key is a.

5.2 Signature Generation

Entity A can sign a binary message m of arbitrary length. In order to sign the message m, A must do the following:

- Select a random secret integer k, $1 \leq k \leq n-1$, with $gcd(k,n) = 1$
- Compute the group element $r = \alpha^k$
- Compute $k^{-1} \mod n$
- Compute $h(m)$ and $h(r)$
- Compute $s = k^{-1}\{h(m) - ah(r)\} \mod n$
- A's signature for m is the pair (r,s)

5.3 Verification

To verify A's signature (r, s) on m, B should so the following:

- Obtain A's authentic public key (α, y)
- Compute $h(m)$ and $h(r)$
- Compute $v_1 = y^{h(r)} \cdot r^s$
- Compute $v_2 = \alpha^{h(m)}$
- Accept the signature if and only if $v_1 = v_2$

6 Hashing and Attacks on Digital Signatures

Hash algorithms produce a specialized size data block from an arbitrary size file. Many computer scientists feel that 160 bits provide enough security for average scenarios. Hashes are often used because it is more efficient and easier to sign a 160 bit data block rather than a arbitrary length file. It is important to keep in mind that digital signatures sign the hash of a file and not the actual file. Because of this, you must be careful not to sacrifice the security of the system. When hash algorithms are used in digital signature schemes, the user generating the signature must hash the data first in order to produce the signature. On the other hand, the user who is verifying a signature must also hash the plaintext before computing the verification process. Hash algorithms tend to run fast, so the double hashing required in a scheme is not a problem. It is convenient to have a hash algorithm as a standard, such has SHA-1, so that both users know what hash algorithm was used in the signature scheme.

Certain attacks have been produced that take advantage of some of the properties of hash algorithms. If one is not careful in choosing a hash algorithm, they can seriously degrade the security of the overall signature.

We can assume that Oscar has a valid signed message, (x, y), and y is the signature of the hash of x, denoted as $h(x)$. Oscar can compute $h(x)$ and try to find $x' \neq x$ such that $h(x') = h(x)$. If Oscar succeeds at this attack, he then came up with another message, x' with a valid signature y. This attack is very much brute force, and Oscar is limited to what x''s he can generate. It is also possible for a person to imitate Alice and pretend that he or she has Alice's private key. If this is possible, Oscar, pretending to be Alice, can generate forged signtaures. It is often the practice of including some identify to a documents to prevent this from happening. Many people also add a timestamp from a third party trusted source, to signify the time a message was signed. It is important for a third party source to generate the time stamp because computer clocks can be easily changed.

6.1 Attack on SHA-1

Over the past few years, reseachers have be improving their analysis in the area of hash functions. Some of the hash algorithm attacks are; original differential attack on SHA0, the near collision attack on SHA0, the multi-block collision techniques, and message modification techniques used in the collision search arrack on MD5. Because of these past improvements, re-searchs have now found a way to reduce the amount of operations on a SHA1 attack. There now exists a very effective way for searching collisions in SHA1. According to [24], they said that collisions of SHA1 can be found with complexity less than 2^{69} hash operations. This is the first attack on the full 80-step SHA1 with complexity less than the 2^{80} theoretical bound. Based on their estimation, they expect that real collisions of SHA1 reduced to 70-steps can be found using today's supercomputers. [24]

7 Representing Bits

There are many ways in which you can implement the representation of bits. Since the majority of computation in polynomial basis and normal basis involves bit manipulation, it is essential that an efficient and fast class be written to do so. I had a hard time making up my mind on a way to represent bits which forced me to write 3 different Binary Galois field element classes to compare. The following is a description of the three methods:

- Array of integers. A simple way of holding a large number of bits would be to make an array of integers and treat them as if each integer were a subarray of bits. The best feature of this method is the speed of XOR. When doing and XOR, it is easy to just XOR the two integers and step to the next integer. Indexing in this method was not too bad. First, you must compute which integer to use in the array. Then, you had to compute which bit along with a mask to extract it. The downfall to this method is shifting. There is no real nice way to shift an array of integers. Shifting once is not as bad as when you have to shift 100 times. The algorithm became long and costly. Because of this, I was forced to think of different alternatives.

- Double linked list of chars. Although it is inefficient to represent a bit using 8 bits, I wanted to see if the sacrifice in memory would in turn make manipulation faster. The majority of bit computation involves shifting, and in this case it is nothing more that pointer arithmetic. The downfall to this method is indexing. If I wanted to get the 100th bit out of 1024 bits, I would have to step through one at a time until I hit the 100th bit. Second to shifting, many operations have to index bits. This forced me to come up with my third method.

- Pointer to a single string. This method is much like the last one where memory is sacrificed in order to increase speed. The nice thing about

this method is the indexing. Indexing is nothing more than computing the index in the string and dereferencing it. The problem I faced in using this method was shifting. I did not want to constantly remake memory and copy it in order to do a shift. My work around was to make a separate pointer that kept track of the beginning of the string. Therefore a single shift was nothing more than a pointer addition. Although shifting was very fast, there was a downfall to this method. Since I kept track of the start of the bits, I had to make sure I did not index out of bounds, meaning when I got to the end of the bit string, I had to wrap around to the beginning and continue. This forced me to write a method to compute the next index. When I had to step through all the bits, that method had to be called which in turn slowed things down.

7.1 Results

Overall I choose to use an array of integers of bits for a number of reasons.

- I no longer use a class to represent bits, instead I use a structure with separate helper functions that operate on that structure. I choose this method to speed up my algorithms by taking out all unnecessary class overhead. Doing this sped the over all project up by a small amount but noticeable.

- Replacing my shift algorithm with assembly inline code seemed to help also. The following inline assembly shows my shift left function.

```
void shiftL(Bits &src){
    asm("movl $0, %%ecx\n\t"        //clear the bit, bit in ecx
        "movl %0, %%ebx\n\t"        //array pointer in ebx
        "LOOP:\n\t"
```

```
            "movl (%%ebx), %%eax\n\t"   //value of array pos in eax
            "shl  $1, %%eax\n\t"        //shift left on position
            "addl %%ecx, %%eax\n\t"     //add in previous high bit
            "movl (%%ebx), %%ecx\n\t"   //calculate current high bit
            "shr $31, %%ecx\n\t"        //shift high bit all the way down
            "movl %%eax, (%%ebx)\n\t"   //store result
            "addl $4, %%ebx\n\t"        //get next pointer address
            "dec %1\n\t"                //decrement array_size
            "jnz LOOP"                  //go back to LOOP if not zero
            :
            : "r" (src.bits), "r" (src.num_words)
            : "eax", "ebx", "ecx", "memory" );
    }
```

- It turns out that my new and improved polynomial multiplication algorithm does not shift bits to the left by more that one position at a time. This eliminated the slow algorithm that went along with shifting many bits either direction.

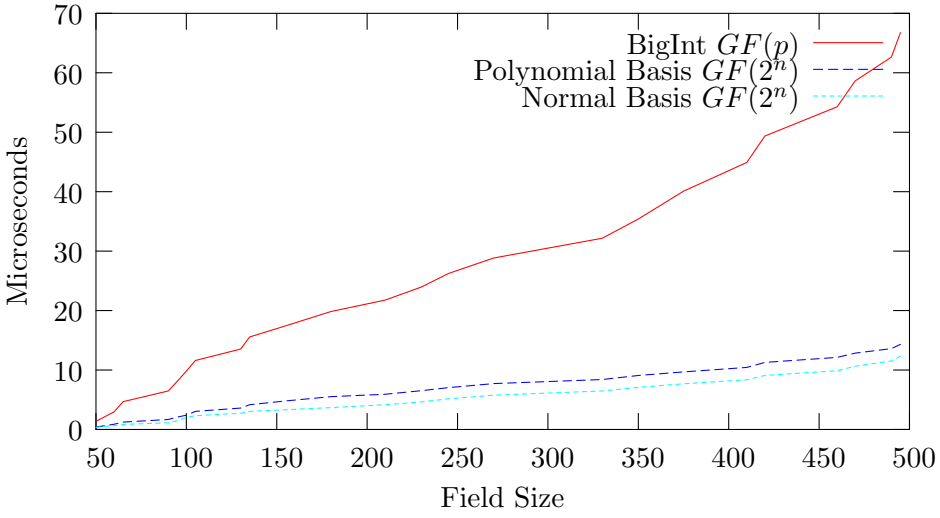
- The most current and efficient algorithms for polynomial basis work on WORDs at a time. It seemed logical to use an array of integers of bits for representing bits because of this reason.

8 Timings

8.1 Addition and Subtraction

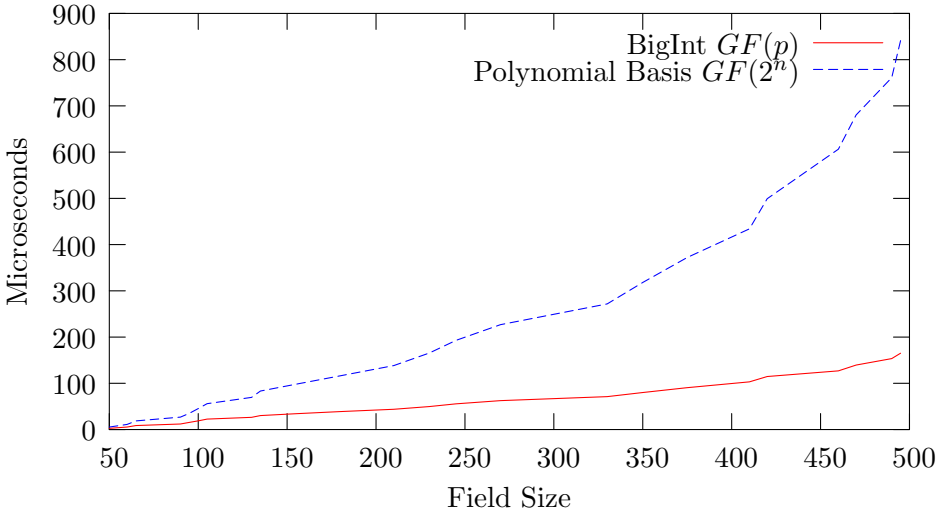
Graph #1 shows the time it take to add or subtract two numbers as the field size grows.

Graph 1: Addition and Subtraction



8.2 Multiplication

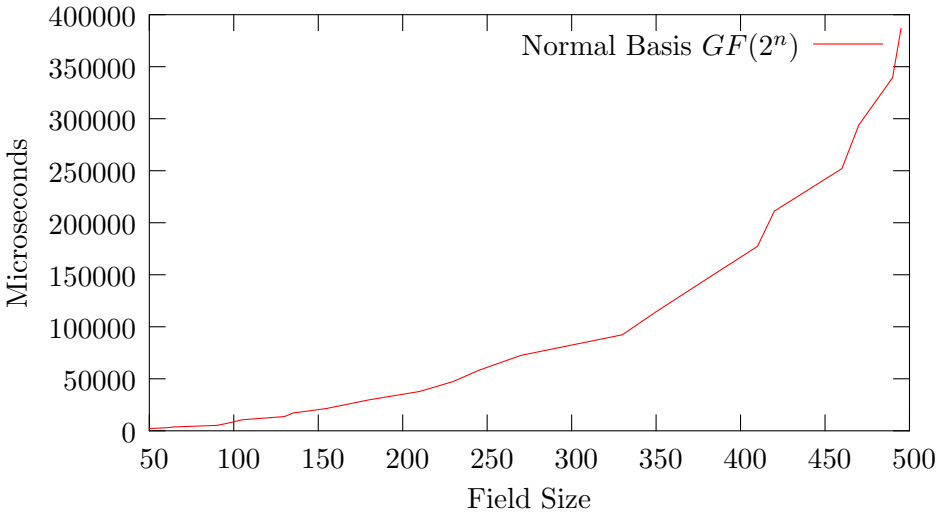
Graph 2: Multiplication



Since normal basis multiplication is much slower than modular arithmetic and polynomial basis, I choose to plot normal basis multiplication in a separate graph. This allows me to better show the differences of modular

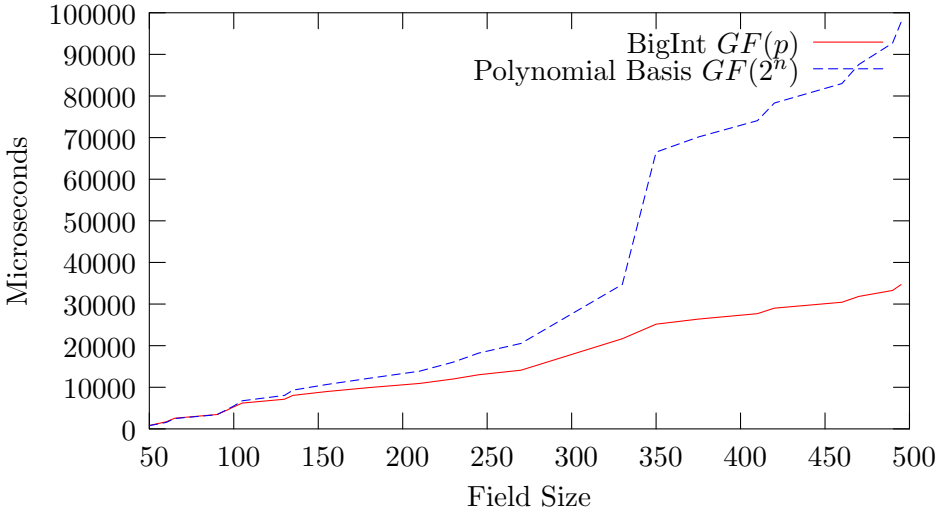
arithmetic and polynomial basis. Graph #3 is the graph for multiplication using normal basis.

Graph 3: Multiplication



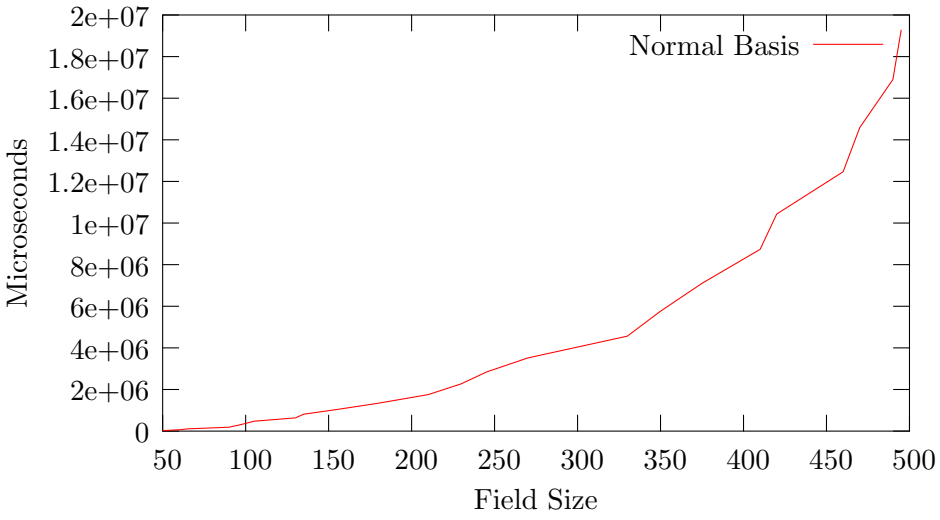
8.3 Exponentiation

Graph 4: Exponentiation



Since normal basis exponentiation is much slower than modular arithmetic and polynomial basis, I choose to plot normal basis exponentiation in a separate graph. This allows me to show the differences of modular arithmetic and polynomial basis more clearer. Graph #5 is the graph for exponentiation using normal basis.

Graph 5: Exponentiation



8.4 ElGamal Key Generation

8.4.1 Modular Arithmetic

- **Calculate** p: Finding a prime p such that you can factor $p-1$ takes a small trick. As long as $p-1$ contains a large prime factor, the security remains unchanged. Therefore, we can calculate p as, $p = (2 * q) + 1$ where q is some large prime. With calculating p in this fashion, we know the factors of $p-1$ to be 2 and q. But, we can not stop here. It is possible that p is still not prime. If p still isn't prime, we must choose another q and try again. This process is repeated until $p = (2 * q) + 1$ is prime.

- **Find generator** α: Knowing the prime factors of $p-1$ comes into

play when trying to find a primitive element of $\mathbb{Z}_p^*$ for a cyclic group of order $p-1$. Since p is prime we know the order of the group is $p-1$. Let us say we have the prime factor of n to be $p_1^{e_1}p_2^{e_2}p_3^{e_3}\cdots p_k^{e_k}$. Take any random element $\alpha \in \mathbb{Z}_p^*$, if $\alpha^{n/p_i} = 1$ for *any* prime factor p_i, then α is not a generator. If this is the case, we must regenerate a random α and try again. This method of finding a generator may not be feasible as n is very large since factoring large numbers can be very time consuming. But, since we choose a p such that $p = (2 * q) + 1$, where q is a large prime number, we can easily factor p into prime factors 2 and q. Now testing α to be primitive is making sure that $\alpha^2 \neq 1$ and $\alpha^q \neq 1$.

- **Calculate** β: Once p and α are found, producing β is easy. We can do this with the following formula, $\beta = \alpha^a$ where a is our secret key.

8.4.2 Polynomial and Normal Basis

Since I will be using the generalized ElGamal signature scheme for polynomial basis and normal basis, generating the keys is identical except for the difference in the set of rules for computation.

- **Generating an Irreducible Polynomial**: Analogous to generating a prime p, we must generate an irreducible polynomial in order to do anything using polynomial basis. Since generating an irreducible polynomial was discussed earlier, I will not get into it here.

- **Forming a Normal Basis**: When it comes to implementation, we really do not need to "form" a normal basis. Instead we are just following a set of rules for addition/subtraction and multiplication. The first step in doing anything would be to construct the lambda matrix. Once the lambda matrix is made, arithmetic can be performed as described earlier.

- **Finding a generator** α: Finding a generator α is much like finding a generator in $\mathbb{Z}_p^*$. The only difference is the exponentiation process.
- **Calculate** β: Calculating β is much like before. Its a simple exponentiation of α and the secret key a.

8.4.3 Results

All test results were performed on a AMD Athlon processor running at 1342.69 MHz. The processor has 256MB of cache and 512MB of system memory. I am running Gentoo Linux r9 with kernel version 2.6.11. All code was compiled with GCC version 3.3.5-20050130.

TABLE 1			
Key Generation Speeds (in seconds)			
Field Size	*Modular Arithmetic*	*Polynomial Basis*	*Normal Basis*
23	0.003028	0.012890	0.084034
51	0.023213	0.014249	0.267478
89	0.074626	0.013057	0.770186
131	0.083480	0.015179	1.971897
179	0.107673	0.019041	4.569792
231	0.150032	0.047263	12.850076
281	0.204232	0.024185	11.344309
359	6.157651	0.075664	36.325744
413	16.621062	0.086534	45.913973
473	19.234599	0.177986	104.070822
519	24.863229	0.163944	112.165402

Since much of the key generation involves exponentiation, normal basis key generation becomes very costly as the size of the field increases. Key generation using modular arithmetic grew fast also but not nearly as fast as normal basis. Using polynomial basis arithmetic produced the fastest key

generation speeds. It is important to calculate the average of many runs when computing the time it takes to generate a signature. The time it takes to generate a signature can vary drastically because of the choices the computer makes when generating a random number. For instance, if the computer generates a bad p or α it has to go back and retry. This adds a considerable amount of time to the key generation process.

8.5 ElGamal Signature Generation

8.5.1 Modular Arithmetic

Signing a message using modular arithmetic is straight forward. Given the keys previously generated, p, α, β with secret key a and a random number $k \in \mathbb{Z}_{p-1^*}$ we can produce a signature for any message we want. It is important that $k \in \mathbb{Z}_{p-1^*}$, meaning the $gcd(k, p-1) = 1$ otherwise the signature will not verify correctly. The signature is formed as follows,

$$\gamma = \alpha^k \mod p$$
$$\delta = (x - a\gamma)k^{-1} \mod p - 1$$

8.5.2 Polynomial and Normal Basis

Since signing a message using polynomial basis and normal basis is the same except for the rules of arithmetic, I will put both sections together. It is not possible to use the same signing method as modular arithmetic. For example using polynomial basis, if we were to compute γ as in the first half of the signature, γ would turn out to be a polynomial. But this does us no good since the second part of the signing algorithms looks for γ to be an integer. Therefore, we must use the generalized ElGamal signature scheme to sign messages. Since we are dealing with binary Galois fields, the order of the field is different than before. We know from the rules of finite fields, the order of the multiplicative group of a field $\mathbb{F}_{2^n}$ is $2^n - 1$. With this new

order, it is important to pick a random k such that $gcd(k, 2^n - 1) = 1$. With this, we can produce a signature as follows,

$$r = \alpha^k$$
$$s = k^{-1} \{h(m) - ah(r)\} \mod n$$

where $h(m)$ and $h(r)$ is the hash of message m and r respectively.

8.5.3 Results

Table #2 shows in seconds how long it takes for a signature to be generated. It is clear that modular arithmetic in $GF(p)$ was the winner. Polynomial basis wasn't too far behind. Normal basis proved to be inefficient.

TABLE 2			
Signature Generation Speeds (in seconds)			
Field Size	*Modular Arithmetic*	*Polynomial Basis*	*Normal Basis*
23	0.000297	0.000341	0.001221
51	0.000632	0.000611	0.015022
89	0.000953	0.001035	0.080900
131	0.001626	0.001922	0.202436
179	0.002201	0.003060	0.479125
231	0.003284	0.004766	1.125449
281	0.004715	0.006789	2.209569
359	0.007193	0.011656	4.259099
413	0.008144	0.016000	6.549010
473	0.009085	0.021403	9.177136
519	0.010674	0.027787	13.799125

Table #3 shows how large of a field we can use when signing a message for a fixed time. These times do not include the time it takes to hash the message but they do include the time it takes to hash variables in the Generalized ElGamal Signature Scheme that was used for polynomial and

normal basis. The hash algorithm used was MD5. This test proved difficult where generating a large prime p such that you can factor $p-1$. It wasn't time consuming to generate a very large prime q, > 2046 bits, it was time consuming finding the p such that $p = 2q + 1$. The largest I tried was 2000 bits, and that ran for over an hour. In this graph, I estimated the size for modular arithmetic based on its growth patterns in the .5, 1.0 and 2.0 time rows.

TABLE 3			
Size of Signature Generated With Fixed Time			
Time	*Modular Arithmetic*	*Polynomial Basis*	*Normal Basis*
.001	89	81	23
.002	169	131	29
.005	311	251	39
.01	509	353	51

8.6 ElGamal Signature Verification

8.6.1 Modular Arithmetic

In order to sign a message using modular arithmetic, we first must obtain the public keys α, β and p. Once this is obtained, we can verify the signature $x, (\gamma, \delta)$ in the following fashion:

Accept the message if

$$\beta^{\gamma}\gamma^{\delta} \equiv \alpha^{x} \mod p$$

otherwise reject the message

8.6.2 Polynomial and Normal Basis

Before we can verify a message using polynomial basis or normal basis we first must obtain the field size. This is important since we also must form a field to verify the signature. Once a field is formed, and we obtain the public keys α and β we can verify the signature $x, (r, s)$ in the following fashion:

Given:

$$v_1 = \beta^{h(r)} r^s$$

$$v_2 = \alpha^{h(m)}$$

where $h(r)$ and $h(m)$ is the hash of r and the message respectively

Accept the message if

$$v_1 = v_2$$

otherwise reject the message

8.6.3 Results

Table #4 shows in seconds how long it takes to verify a signature. It is clear that modular arithmetic in $GF(p)$ was the winner. Polynomial basis wasn't too far behind. Normal basis proved to be inefficient.

TABLE 4			
Signature Verification Speeds (in seconds)			
Field Size	*Modular Arithmetic*	*Polynomial Basis*	*Normal Basis*
23	0.001724	0.002372	0.018652
51	0.002272	0.002941	0.074646
89	0.003014	0.004526	0.261936
131	0.004016	0.007458	0.631102
179	0.005084	0.007746	1.282755
231	0.006575	0.009409	2.200204
281	0.009182	0.012738	3.713323
359	0.010805	0.030659	7.059375
413	0.013019	0.032599	10.604554
473	0.016488	0.040902	14.750914
519	0.018105	0.045287	18.664613

Table 5 shows how large of a field we can use when verifying a message for a fixed time. As in the last fixed time test, this proved difficult where generating a large prime p such that you can factor $p-1$. Since there is such

a difference between the timing of Normal Basis vs. Polynomial Basis and modular arithmetic, it was difficult to compare. For this reason, I had to leave out Normal Basis. The smallest jump in field size using normal basis exceeded the possibility of even computing the keys for a comparable field size of the other representations.

TABLE 5		
Size of Signature Verification With Fixed Time		
Time	*Modular Arithmetic*	*Polynomial Basis*
.002	47	23
.005	179	95
.01	359	241
.02	537	291

9 Algorithm Complexity

9.1 Polynomial Basis

9.1.1 Addition and Subtraction

Addition and subtraction are noting more that an XOR in binary Galois fields. Since XOR's two list of words takes stepping through the words, the complexity for addition and subtraction is $O(n)$ where n is the amount of words in the list. The following algorithm implements addition and subtraction.

```
void XOR(Bits &dest, Bits &src1, Bits &src2){
    for (unsigned int i = 0; (i < dest.num_words) &&
                             (i < src1.num_words) &&
                             (i < src2.num_words); i++) {
        dest.bits[i] = src1.bits[i] ^ src2.bits[i];
    }
}
```

9.1.2 Multiplication

Multiplication is a pretty costly algorithm measuring in at $O(nlog(n))$ in it's worse case. Here is the algorithm on how to multiply two polynomials.

```
void PolynomialBasis::mult(Bits &dest, Bits &x, Bits &y){

    unsigned int s = x.num_words;
    Bits T;
    unsigned int i, k;
    int j;

    allocate_words(T, (2 * s));
    for (j = WORD_B - 1; j >= 0; j--){
        for (i = 0; i <= (s - 1); i++){
            if (get_bit(x, (i * WORD_B) + j)){
                for (k = 0; k <= (s - 1); k++){
                    T.bits[k + i] = T.bits[k + i] ^ y.bits[k];
                }
            }
        }
        if (j != 0)
            shiftL(T);
    }
    reduce(T);

    for (unsigned int i = 0; i < dest.num_words; i++)
        dest.bits[i] = T.bits[i];

    free_bits(T);
```

```
}
```

The nested for loops is where most of the computation is done. You can see here that if every term in x was set, then the complexity would be $O(WORDB * n * n)$ or just $O(n^2)$.

9.1.3 Squaring

Squaring a polynomial is really fast. Using the table lookup method described earlier, squaring a polynomial can be done in $O(n)$, where n is the amount of WORDs needed to represent a polynomial. Here is the squaring algorithm for reference.

```
void PolynomialBasis::square(Bits &dest, Bits &src){
    Bits T;
    unsigned int index = 0;

    allocate_words(T, (2 * src.num_words));

    for (unsigned int i = 0; i < src.num_words; i++){
        T.bits[index++] = square_lookup[src.bits[i] & 0xFFFF];
        T.bits[index++] = square_lookup[src.bits[i] >> 16];
    }
    reduce(T);

    for (unsigned int i = 0; i < dest.num_words; i++)
        dest.bits[i] = T.bits[i];

    free_bits(T);
}
```

9.1.4 Exponentiation

Exponentiation is a combination of squares and multiplies. Therefore, it's worst case is because of multiplication with was $O(n^2)$. Here is the exponentiation algorithm for reference.

```
void PolynomialBasis::expon(Bits &dest, Bits &x, BigInt amt){
    BigInt r = 0;
    unsigned int count = 0;
    clearWords(dest);
    dest.bits[0] = 0x1;

    Bits T;
    allocate_words(T, 2 * x.num_words);

    // get the exponent in reverse
    while (amt != 0){
        r = r * 2;
        r = r + (amt & 1);
        amt = amt / 2;
        count++;
    }

    while (count--){
        square(dest, dest);

        if (r & 1){
            mult(dest, dest, x);
        }
        r = r / 2;
```

```
    }
    free_bits(T);
}
```

9.2 Normal Basis

9.2.1 Lambda Matrix Generation

Generating the lambda matrix is not quick. The complexity falls under $O(m)$ where m is the size of the field. Here is the algorithm to show where I got the complexity from.

```
void NormalBasis::generate_lambda1(){
    BigInt foo, two, zero, one;
    zero = 0; one = 1; two = 2;
    bool gotfirst = false;

    for (unsigned int i = 0; i < m; i++){
        for (unsigned int j = 0; j < m; j++){
            foo = (two ^ i) + (two ^ j);
            foo = foo % (m + 1);
            if ((foo == one) || (foo == zero)){
                if (gotfirst){
                    table2[i] = j;
                }
                else{
                    table1[i] = j;
                    gotfirst = true;
                }
            }
        }
```

```
        gotfirst = false;
    }
}
```

9.2.2 Addition and Subtraction

Addition and subtraction is the same as in polynomial basis. It is a simple step through of WORD and XOR operations. Therefore, the complexity of normal basis addition and subtraction is $O(n)$. Since it uses the same algorithm as in polynomial basis, I will not list the code here.

9.2.3 Multiplication

Multiplication in normal basis is slow. The complexity for the multiplication algorithm is $O(m^2)$ where m is the size of the field. Here is the multiplication algorithm for reference:

```
void NormalBasis::mult(Bits &dst, Bits &x, Bits &y){
    unsigned int tmp1, tmp2;
    tmp1 = tmp2 = 0;
    for (unsigned int k = 0; k < dst.num_bits; k++){
        tmp1 = get_bit(x, k) * get_bit(y, (table1[0] + k) % y.num_bits);

        for (unsigned int i = 1; i < x.num_bits; i++){
            tmp2 ^= (get_bit(x, (k + i) % x.num_bits) *
                (get_bit(y, (table1[i] + k) % y.num_bits) ^
                    get_bit(y, (table2[i] + k) % y.num_bits) ));
        }
        if (tmp1 ^ tmp2)
            set_bit(dst, k);
        else
```

```
            clear_bit(dst, k);

        tmp1 = tmp2 = 0;
    }
}
```

9.2.4 Squaring

Squaring is one of the most efficient algorithm in normal basis. Since squaring is a simple circular shift left, the complexity of the square operation is $O(n)$ where n is the amount of WORD needed to represent a field element. Here is an example of squaring a field element in normal basis.

```
void circ_shiftL(Bits &src){
    WORD bit = src.bits[src.num_words - 1] &
                    indexMask[(src.num_bits - 1) % WORD_B];
    shiftL(src);
    if (bit)
        src.bits[0] += 1L;
}
```

9.2.5 Exponentiation

Exponentiation is a combination of squares and multiplies. Therefore, it's worst case is because of multiplication with was $O(m^2)$. Here is the exponentiation algorithm for reference.

```
void NormalBasis::expon(Bits &dst, Bits &x, BigInt pow){

    BigInt r = 0;
```

```
    int count = 0;
    set_allHigh(dst);

    Bits tmp;
    allocate_bits(tmp, dst.num_bits);

    // get the exponent in reverse
    while (pow != 0){
        r = r * 2;
        if (pow & 1)
            r = r + 1;

        pow = pow / 2;
        count = count + 1;
    }

    while (count){
        circ_shiftL(dst);

        if (r & 1){
            mult(tmp, x, dst);
            copyBits(dst, tmp);
        }
        r = r / 2;
        count--;
    }
}
```

10 Program Usage

My goal of this section is to explain how to use the program written in order to sign and verify messages. All code was written in C++ on a linux machine. The user will need to have libmhash and libgmp installed in order to use this program.

10.1 Generating Parameters

I felt it was important to separate parameter generation from the rest of the program. Since it would be inefficient for a server to regenerate parameters ever time its wants to sign a message, it would be helpful to save the parameters into a text file for a later date. Before a user can attempt to sign a message he/she must first generate parameters. This is done by executing the **genParams** executable. By default this program will output a "params" file with a list of the given parameters for field size 131. If you wish to change the field size, this can easily be done by passing the executable the command line argument "-fs x" where x is the desired field size. Since polynomial and normal basis generation depends on the field size, not all field sizes will work. For instance, it is essential that polynomial and normal basis algorithms know the factors of the order of the field. To help in finding the factors of the order, a data file called "faclookup" exists in the working directory. If the field is not listed there, the basis will not be formed. If the user insists on using that field, he or she is responsible for adding the factors into the database file. In addition to the field size having to be listed in the data base, it must also be of Type I or Type II normal basis, otherwise the user will not be able to sign or verify using normal basis. It is important to pay attention to the output of "genParams" to make sure all parameters are generated successfully.

10.2 Signing a Message

Signing a message is easily done with the command **sign**. The executable must receive at least 1 parameter, being the message to sign. By default, the sign program will use modular arithmetic to sign the message. If this is not what the user wants, he or she can change this to polynomial or normal basis by passing the "-poly" or "-norm" switch, respectively. Signing a message will result in a file created containing the signature. The name of the file created is the name of the input message file plus a ".sig" extension added to the end.

10.3 Verifying a Message

Verifying a message is very simple. All that need to be done is, "verify message" where message is the name of the file that was originally signed. There is no need to pass the verify executable the signature file, it will automatically look for it and terminate if not found.

11 Area's of Improvment

Much of my research told me that using a Galois Field $GF(2^m)$ represented with a polynomial basis would outperform standard modular arithmetic in $GF(p)$. Since my results did not show this, I spent a great deal of time trying to improve my representation of polynomials. There is a really interesting alternative for representing a finite field $GF(2^m)$. According to [7] we can represent field elements as polynomials with coefficients in the smaller field $GF(2^{16})$. Calculations in this smaller field are carried out using pre-calculated lookup tables. The nice thing about this representation is that the field $GF(2^{16})$ can be represented in a single computer WORD. It is well known that a field can be considered as a vector space over one of its subfields. The proper subfields of $GF(2^n)$ are the fields $GF(2^r)$, with

$r|n$ and $0 < r < n$. [7] My implementation of polynomials left me with a large number of bitwise operations. Some operations had me testing single bits and shifting words around. This is inefficient since most computers are designed for WORD operations.

References

[1] Alfred J. Menezes, Paul C. van Oorschot, S. A. V. *Handbook of Applied Cryptography.* CRC Press, 1996.

[2] Arash Reyhani-Masoleh, M. A. H. *Fast Normal Basis Multiplication Using General Purpose Processors*, April 2001.

[3] B. Sunar, C. K. *An Efficient Optimal Normal Basis Type II Multiplier.*

[4] Darrel Hankerson, Julio Lopez Hernandez, A. M. *Software Implementation of Elliptic Curve Cryptography Over Binary Fields.*

[5] Don Johnson, Alfred Menezes, S. V. The elliptic curve digital signature algorithm (ecdsa). *IJIS* (2001).

[6] E. Savas, C. K. K. Efficient methods for composite field arithmetic. Tech. rep., Oregon State University, December 1999.

[7] Erik De Win, Antoon Bosselaers, S. V. A fast software implementation for arithmetic operation in $GF(2^n)$. Tech. rep., Katholieke Universiteit Leuven.

[8] Gao, S. Digital signatures. `http://www.math.clemson.edu/faculty/Gao/cryptomod/node5.html`, October 1999.

[9] Gordon, D. M. *A survey of fast exponentiation methods.* Center for Communications Research, December 1997.

[10] Joachim Von Zur Gathen, M. G. Constructing normal bases in finite fields. Tech. rep., University of Toronto, June 1989.

[11] Leung, I. Multiplication. `http://cse.cuhk.edu.hk/~khleung/thesis/node34.html`, June 2001.

[12] Lopez, Julio Dahab, R. High-speed software multiplication in $\mathbb{F}_{2^m}$. Tech. rep., State University of Campinas, May 2000.

[13] Neal Koblitz, Alfred Menezes, S. V. The state of elliptic curve cryptography. *Designs, Codes and Cryptography 19* (2000), 173–193.

[14] Peng Ning, Y. L. Y. Efficient software implementation for finite field multiplication in normal basis. Tech. rep., North Carolina State University.

[15] Publication, F. I. P. S. Digital signature standard, May 1994.

[16] R. Lidl, H. N. *Finite Fields*. Cambridge University Press, 1996.

[17] Richard Schroeppel, Hilarie Orman, S. O. Fast key exchange with elliptic curve systems. Tech. rep., The University of Arizona, March 1995.

[18] Solinas, J. A. Efficient arithmetic on koblitz curves. *Designs, Codes and Cryptography 19* (2000), 195–249.

[19] Stinson, D. R. *Cryprography Theory and Practice*. Chapman and Hall/CRC, 2002.

[20] Tozer, C. *Digital Signature Guidelines*. American Bar Association, 1996.

[21] Wu, H. *On Complexity of Squaring Using Polynomial Basis in* $GF(2^m)$.

[22] Wu, H. *On Computation of Polynomial Modular Reduction*, June 2000.

[23] X9.62, A. Public key cryptography for the financial services industry: the elliptic curve digital signature algorithm (ecdsa). 1999.

[24] Xiaoyun Wang, Yiqun Lisa Yin, H. Y. Collision search attacks on sha1. Tech. Rep. 13, February 2005.

[25] Yongfei Han, Peng-Chor Leong, P.-C. T. Fast algorithms for elliptic curve cryptosystems over binary finite field. *Lecture Notes in Computer Science 1716* (1999), 75–85.

[26] Zhi Li, John Higgins, M. C. Performance of finite field arithmetic in an elliptic curve cryptosystem. Tech. rep., Brigham Young University.