Masters Project: Efficient Encryption on Limited Devices

Roderic Campbell
Department of Computer Science
Rochester Institute of Technology
Rochester, NY, USA
rmc8917@cs.rit.edu

July 3, 2006

```
---------------------------------------
Chair: Prof. Alan Kaminsky          Date

---------------------------------------
Reader: Prof. Hans-Peter Bischof    Date

---------------------------------------
Observer: Prof. Leonid Reznik      Date
```

Abstract

Encryption algorithms have been used since the dawn of time to ensure secure communication over insecure communication channels. Once a secret encryption key is established and as long as the key remains secret, two parties can communicate freely over open channels. The question of how to obtain such a secret key is a large dilemma. Many methods of obtaining such keys have been tried from the most basic form of an one-on-one encounter, to more advanced techniques like Diffie-Hellman. This paper attempts to compare some modern key exchange protocols and determine which would be the most appropriate, in terms of computational efficiency, for a small personal computing device.

Contents

1 Introduction

Over the past decade, there has been a notable increase in the widespread use of personal digital assistants, mobile phones with advanced functionality and consumer devices which contain limited, yet non-trivial processors. Such devices have had, and will continue to have a significant effect on the flow of social and business communication. At the same time, we have seen personal computers handling more and more highly sensitive data such as bank and business transactions. The next logical step is to see the limited devices handling such transactions on the go in a secure fashion. Human nature and the pace of business demands that all of these interactions take place effectively and efficiently. Thus follows the need for an efficient, lightwieght way of securely establishing a secret key for communication between two parties.

Key exchange protocols have existed since men have kept secrets from other men. Stories of Julius Caesar and his armies encrypting messages are legendary to those who know the history of espianage. The appropriately named Caesar Cipher is a simple algorithm which, given a simple key, maps out an alternate alphabet by shifting the values of each letter in a message any number of letters to the left. If a key is 1, then any b in your message becomes an a, any c becomes a b, any d becomes a c and so on. This will result in a scrambled mess of a message resulting in an encrypted or ciphertext message. The receiving party simply shifts each letter of the encrypted message to the right 1 letter resulting in the decrypted or plaintext message. The key can be any number between 1 and 26.

Ignoring the fact that there are only 26 possible keys, this simple encryption algorithm depends on each party knowing the secret key. In the old days, this would most likely be done by a trusted courier or a worst case scenario of a personal conversation between to two communicating parties.

Modifications of this algorithm allowed for more complex keys which could consist of words or larger numbers as keys. Given this technique, keys could be a previously established word, such as a last name, or a country, or a favorite food. This was still very easy to break and still requires a previous agreement on a key establishing method.

Later advancements in encryption lead to more complex mathematical structures requiring hundreds or thousands or millions of calculations which realistically, requires efficient computational machines such as computers. Algorithms used today, such as RSA or Diffie-Hellman, depend on modular exponentiation of very large numbers which would take an enormous amount of time if done by hand. These algorithms can be used in such a way that two parties can end up with the same data without revealing the data in an open communication channel. These are called Public Key Crypto-Systems.

Public key crypto-systems are based on the concept of an one-way function. To understand one-way functions, we must first grasp the concept of a two-way function. An example of a two-way function is addition. With the equation $A + B = C$, and given A and B, it is easy to calculate C. Likewise, if we are given B and C, we could easily compute A by using the inverse of addition *subtraction*. This gives us the equation $C - B = A$. A one-way function on the other hand, is one where it is not computationally feasible to come up with one of the arguments, given the remaining arguments and the solution. An example of an one-way function is modular exponentiation. A function of the form $y = x^n (mod\ p)$ is a one-way function. Given x, y, and p, we would be unable to *efficiently* determine n if we were using large enough values.

This computationally intensive process is the basis and starting point for this paper. The implementation for my Masters Project consists of a full implementation of a Arbitrary Length Precision Integer library, as well as implementations of Diffie-Hellman(DH), Elliptic Curve Encryption (ECC) and XTR which stands for Efficient and Compact Subgroup Trace Representation. Each of these encryption algorithms require very large numbers to obtain ample security levels.

DH is a tried and true method developed over 20 years ago and relies on the concept above of an one-way function called modular exponentiation of large numbers. ECC relies on an algebra based on curves of the form $y^2 = x^3 + ax + b$. Similar levels of security can be obtained by using numbers that are several times smaller than the numbers used in DH. This will be explained in detail later in the paper. XTR claims to achieve similar security levels to ECC using even smaller numbers.

My implementation attempts to measure the validity of the claims of each

of the algebras described in the paper. This can by done by running side by side comparisons of each algebra running Diffie-Hellman and computing a public key set. This is done on both a desktop platform as well as a small device. At the end conclusion is drawn about the suitability of these three methods for use on small devices.

This paper will describe each of the algorithms in depth, as well as describe the results obtained from the tests that were run on the code. The rest of this paper describes the implementation of my Master's Project.

2 Algorithm Background

In this section, I will describe in a bit of detail, how each of the three algorithms work.

2.1 Diffie-Hellman

The Security of DH relies on the difficulty in solving the Diffie-Hellman Problem, which is a variation of the Discrete Logarithm Problem. The Discrete Logarithm Problem is to find a such that $x = g^a (mod\ p)$, given x, g, and p where p is a large prime number, g a generator for a large subgroup of Z_p^*. The Diffie-Hellman problem is to find $g^{ab} (mod\ p)$ given $x = g^a (mod\ p)$ and $y = g^b (mod\ p)$, g and p (but not a and b). Since there is no efficient algorithm known which can solve this problem, then the Diffie-Hellman Algorithm can be used as an encryption algorithm given a large enough prime number.

The DH implementation is as follows:

1. Alice and Bob agree to use prime number p and a generator g from Z_p^*.
2. Alice chooses a secret integer a and computes $g^a (mod\ p)$ and sends this over to Bob.
3. Bob chooses a secret integer b and computes $g^b (mod\ p)$ and sends this over to Alice.

4. Alice computes $(g^a(mod\ p))^b(mod\ p)$ and has the secret key.

5. Bob computes $(g^b(mod\ p))^a(mod\ p)$ and has the secret key.

Since modulo arithmetic is associative $(g^a(mod\ p))^b(mod\ p)$ and $(g^b(mod\ p))^a(mod\ p)$ are identical. Yet neither a nor b have been sent over an insecure communication channel.

In this case Bob is a small device and is not suitable for the generation of the initial primes. At the end of all calculations, Alice and Bob are able to use a more efficient block cipher encryption algorithm to encrypt a conversation key.

It must be noted that the use of DH usually depends on extremely large prime numbers. The numbers must be large to ensure sufficient security. This is where the problem of DH comes in. While it is very secure, it is also very costly.

2.2 Elliptic Curve Cryptography

The next implemented algorithm is Elliptic Curve Cryptography (ECC). It is believed that ECC gives remarkably similar levels of security at a fraction of the cost of DH. With a further understanding of ECC, small devices may utilize the technology to exchange encryption keys. These keys may be used to encrypt bank transactions, credit card numbers, and other sensitive information.

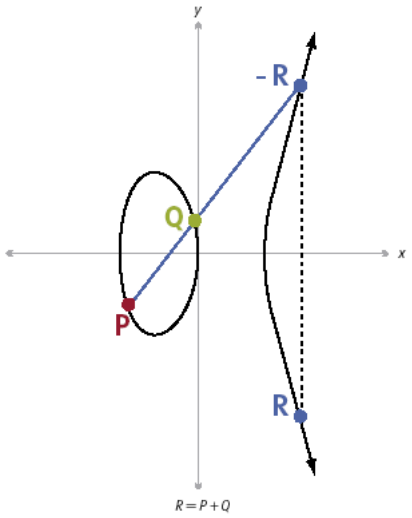

Figure 1: Elliptic curve Addition [3]

As a high level overview of Elliptic curves refer to the figure above. The encryption technique is based off of an abstract concept of elliptic curve addition. Suppose you have a curve E as shown in the figure. Given a point P and a point Q, addition is rougly defined as drawing a straight line through the two points and finding the reflection, a third point on the curve, called $-R$. The inversion of this point, R, is the sum of P and Q.

Elliptic curves are functions of the form $y^2 = x^3 + ax + b$ The cryptography of Elliptic curves relies on the widely believed difficulty of solving the discrete logarithm problem for a group of an elliptic curve over some finite field. [2] states that elliptic curves need to be non-supersingular form. This provides us with a cryptographically strong use. Non-supersingular curves are such that they are defined by the equation $y^2 = x^3 + ax + b$. The following example from [2] describes how curves over a finite field works.

Let E be the curve $y^2 = x^3 + 10x + 5$ over the field $GF(13)$. Then the points on E are $\{O, (1,4), (1,9), (3,6), (3,7), (8,5), (8,8), (10,0), (11,4), (11,9)\}$.

With an understanding of elliptic curves, we can define the algebra over the curves. Addition being the simplist form is mathematically described in [2]. The addition of elliptic curves as used in cryptography is analogous to

multiplication in the group Z_p^* as used in integer DH schemes.

Multiplication in elliptic curves is again described in [2] and can be thought of as repeated addition on elliptic curves over a finite field. Elliptic curve multiplication used in cryptography is analagous to exponentiation in the group Z_p^* in integer DH schemes.

Encryption of a value is conceptually simple. The idea is that you take a random point on the curve, and add it to itself some number of times, your plaintext, using a specialized algebra created specifically for Elliptic curves.

As stated before, multiplication over Elliptic curves is simply repeated additions of a point to itself. Using this concept, we are able to encode an integer onto the curve by multiplying the point by the integer we wish to encode.

One concept that needs to be cleared up is the actual use of ECC as an encryption technique. So far, all that has been discussed is the underlying concepts that make up ECC. For this implementation, the actual encryption algorithm that sits on top of the ECC algebra is in fact the same DH that is used in the previous section.

The new DH with the ECC algebra looks something like this:

1. Alice and Bob agree to use a curve and point c and p.
2. Alice chooses a secret integer a and computes $c * a$ and sends this new point over to Bob.
3. Bob chooses a secret integer b and computes $c * b$ and sends this new point over to Alice.
4. Alice computes $(c * b) * a$ and has the secret point.
5. Bob computes $(c * a) * b$ and has the secret point.

The analogy between the normal discrete logarithm problem and the elliptic curve logarithm problem is focused around the basic operation of multiplication for normal discrete logarithm and addition of points for elliptic curves. The main operation being exponentiation in discrete logarithm and scalar

multiplication in elliptic curves. This scalar multiplication is a result of calculations which are not modular exponentiation. This is the main reason that elliptic curve cryptography boasts such significant gains over normal Diffie-Hellman.

2.3 XTR

Efficient Compact Subgroup Trace Representation [5] estimates that encryption using XTR is 21% faster than ECC, and that decryption can be up to 45% faster. This evidence, should it prove to be valid, provides powerful arguments for the widespread investigation of XTR. This significant drop in memory usage and running time could lead us to the next key distribution standard.

[5] states that XTR key selection is much faster than DH and orders of magnitude easier and faster than ECC. For these reasons, [5] insists that XTR will find its home in small devices such as smart cards and wireless devices. This complements the goals and direction of this implementation.

The general description and a simplified background of XTR is not that is is an encryption technique. Much like ECC, the best way to think about it is as an alternate algebra with a form of modular exponentiation which is much faster than standard modular exponentiation. It is not actually exponentiation, it is just an alternate way to exploit the discrete logarithm problem. It gives us a new one way function.

The overall concept of XTR is based on [1] what [5] calls the Optimal Normal Basis Representation. The paper describes numbers as a tuple of numbers. A number x in GF(p^2) is represented as (x1,x2). In this paper, saying x <=> (x1,x2) means *the optimal normal basis representation of x is (x1,x2).* Likewise, the number t in GF(p) is represented as t <=> (p-t,p-t), which is equivalent to (-t,-t) (mod p). For example, 3 <=> (p-3,p-3).

The algebra is defined as follows, where all calculations are done modulo p:

1. Addition and subtraction. Let x <=> (x1,x2) and y <=> (y1,y2). Then x+y <=> (x1+y1,x2+y2) and x-y <=> (x1-y1,x2-y2).

2. Multiplication. Let x <=> (x1,x2) and y <=> (y1,y2). First compute
   t = x1 * y1
   u = x2 * y2
   v = (x1 + x2) * (y1 + y2)
   w = t + u - v
   Then x*y <=> (u+w,t+w).

3. Squaring. Let x <=> (x1,x2). First compute
   t = x2 * (x2 - x1 - x1)
   u = x1 * (x1 - x2 - x2)
   Then x^2 <=> (t,u). Doing it this way, we only have to do two multiprecision multiplications.

4. Exponentiation to the power p. Let x <=> (x1,x2), then x^p <=> (x2,x1)

5. Computing $xz - yz^p$. Let x <=> (x1,x2), y <=> (y1,y2), and z <=> (z1,z2). First compute
   t = z1 * (y1 - x2 - y2) + z2 * (x2 - x1 + y2)
   u = z1 * (x1 - x2 + y1) + z2 * (y2 - x1 - y1)
   Then $xz - yz^p$ <=> (t,u).

With that being said, there are a few things that need to be noted. First is the implementation of exponentiation to the power of p. Notice how it is simply a value swap within the tuple and essentially is free. Second is the relatively small number of calculations that are required to do most operations. For instance, as stated before, squaring only takes 2 multiplications. This should prove to be very effecient for the types of operations that will be used in the encryption algorithms.

The actual XTR exponentiation is quite different than standard exponentiation. XTR exponentiation is an attempt to find the group $\{c_{n-1}, c_n, c_{n+1}\}$ where $c_n = h_0^n + h_1^n + h_2^n (mod\ p^2)$, where h_0, h_1, and h_2 are the three roots of the cubic polynomial $X^3 - cX^2 + c^pX - 1$.

The algorithm is as follows:

1. If n = 0, then:
$c_{n-1} = c^p$,

$c_n = 3$,
$c_{n+1} = c$.
Done.

2. Else if n = 1, then:
$c_{n-1} = 3$,
$c_n = c$,
$c_{n+1} = c^2 - c^p - c^p$.
Done.

3. Else if n = 2, then:
$c_{n-1} = c$,
$c_n = c^2 - c^p - c^p$,
$c_{n+1} = c * c_n - c^p * c + 3$ [Corollary 2.3.5.ii] [Use $xz - yz^p$ formula]
Done.

4. Else: Do steps 5-13.

5. If n is even, then m = n-1, else m = n.

6. $\overline{m} = (m-1)/2$.

7. Let the bits of $\overline{m}$ be numbered from 0 (least significant) to r (most significant); that is:
$\overline{m}_0$ = least significant bit of $\overline{m}$
$\overline{m}_1$ = next least significant bit of $\overline{m}$
. . .
$\overline{m}_{r-1}$ = next most significant bit of $\overline{m}$
$\overline{m}_r$ = most significant bit of $\overline{m}$

[Comment: $\overline{m}_r$ is always 1; that is, r is the index of the leftmost 1 bit of $\overline{m}$.]

8. Initialize the three output quantities: $c_{n-1} = c^2 - c^p - c^p$,
$c_n = c * c_{n-1} - c^p * c + 3$, [Use $xz - yz^p$ formula]
$c_{n+1} = c * c_n - c^p * c_{n-1} + c$. [Corollary 2.3.5.ii] [Use $xz - yz^p$ formula]

9. For j = r-1, r-2, ..., 1, 0: Do steps 10-11.

[Begin loop body]

10. If $\overline{m}_j = 0$, then: Compute three temporary quantities from the current three output quantities: $c_{2n-2} = c_{n-1}^2 - c_{n-1}^p - c_{n-1}^p$ [Corollary 2.3.5.i]
$c_{2n-1} = c_{n-1} * c_n - c^p * c_n^p + c_{n+1}^p$ [Corollary 2.3.5.iii] [Use $xz - yz^p$ formula]
$c_{2n} = c_n^2 - c_n^p - c_n^p$ [Corollary 2.3.5.i]
Replace the output quantities with the temporary quantities:
$c_{n-1} = c_{2n-2}$
$c_n = c_{2n-1}$
$c_{n+1} = c_{2n}$

11. Else if $\overline{m}_j = 1$, then: Compute three temporary quantities from the current three output quantities: $c_{2n} = c_n^2 - c_n^p - c_n^p$ [Corollary 2.3.5.i]
$c_{2n+1} = c_{n+1} * c_n - c * c_n^p + c_{n-1}^p$ [Corollary 2.3.5.iv] [Use $xz - yz^p$ formula]
$c_{2n+2} = c_{n+1}^2 - c_{n+1}^p - c_{n+1}^p$ [Corollary 2.3.5.i]
Replace the output quantities with the temporary quantities:
$c_{n-1} = c_{2n}$
$c_n = c_{2n+1}$
$c_{n+1} = c_{2n+2}$

[End loop body]

12. If n is even, then:
Compute one temporary quantity from the current three output quantities: $c_{n+2} = c * c_{n+1} - c^p * c_n + c_{n-1}$ [Corollary 2.3.5.ii] [Use $xz - yz^p$ formula]
Replace the output quantities as follows:
$c_{n-1} = c_n$
$c_n = c_{n+1}$
$c_{n+1} = c_{n+2}$

13. Done.

Additionally we need to define how to find $Tr(g)$. This is quite simple and is explained in [5] as algorithm 3.2.2.

1. Pick $c \in GF(p^2) \backslash GF(p)$ at random and compute c_{p+1} using the above algorithm.
2. If $c_{p+1} \in GF(p)$ then return to Step 1.
3. Compute $c_{(p^2-p+1)/q}$ using the above algorithm.
4. If $c_{(p^2-p+1)/q} = 3$, then return to Step 1.

5. Let $Tr(g) = c_{(p^2-p+1)/q}$.

Using this new found algebra as an encryption technique is very similar to the ECC technique. We simply take the discrete logarithm problem embedded within XTR to exploit the key exchange protocol of DH.

XTR's DH is decribed in [5] as follows:

Suppose that Alice and Bob who both have access to the XTR public key data $p, q, Tr(g)$ want to agree on a shared secret key K. This can be done using the following XTR version of the Diffie-Hellman protocol:

1. Alice selects at random $a \in Z$, $1 < a < q - 2$, uses Algorithm 2.3.7 to compute $S_a(Tr(g)) = (Tr(g^{a-1}), Tr(g^a), Tr(g^{a+1})) \in GF(p^2)^3$, and sends $Tr(g^a) \in GF(p^2)$ to Bob.

2. Bob receives $Tr(g^a)$ from Alice, selects at random $b \in Z, 1 < b < q - 2$, uses Algorithm 2.3.7 to compute $S_b(Tr(g)) = (Tr(g^{b-1}), Tr(g^b), Tr(g^{b+1})) \in GF(p^2)^3$. and sends $Tr(g^b) \in GF(p^2)$ to Alice.

3. Alice receives $Tr(g^b)$ from Bob, uses Algorithm 2.3.7 to compute $S_a(Tr(g)^b) = (Tr(g^{(a-1)b}, Tr(g^{ab}), Tr(g^{(a+1)b})) \in GF(p^2)^3$. and determines K based on $Tr(g^{ab}) \in GF(p^2)$.

4. Bob uses Algorithm 2.3.7 to compute $S_b(Tr(g)^a) = (Tr(g^{a(b-1)}), Tr(g^{ab}), Tr(g^{a(b+1)})) \in GF(p^2)^3$, and determines K based on $Tr(g^{ab}) \in GF(p^2)$.

The results and findings of this implementation are described at a later point in the paper.

2.4 Secret Key Creation

In a full encryption scheme we would see the secret keys that are derived from the previously mentioned algorithms used in a hash function to create the actual secret key. This hashed secret key is then used as they secret key in a a block cipher for traffic data. The implementation described in this paper, however, did not implement or measure the performance of the traffic encryption. This falls out of the scope of the project which was implementing and measuring parameter generation and public key exchange protocols only.

3 Functional Specification

3.1 Desktop Environment

The Desktop environment consists of a 2.4 Ghz Pentium 4 processor with 1 GB PC-2700 DDR RAM. The JDK version on the system is 1.4.2.

3.2 Limited Device

The limited device consists of a Samsung VGA 1000 Java enabled mobile phone. Java 2 Platform, Micro Edition with the Connected Limited Device Configuration(CLDC), and the Mobile Information Device Profile make up the Java Platform which is used[4]. The device itself contains 1024KB of memory for Java applications.

3.3 Software Implementation

The implementation consists of the following tasks:

1. An Implementation of Diffie-Hellman using integer arithmetic, elliptic curve arithmetic, and XTR arithmetic parameter generation schemes in Java in a Desktop computing environment as well as the same arithmetic on a limited computing environment.
2. Implemented underlying algorithms such as arbitrary precision integer arithmetic in Java in the limited computing environment described above.
3. Implemented public key generation and key exchange for the above in Java in a desktop computing environment.
4. A system of metrics to analyze the following:
   (a) Running time of 2048-bit DH in comparison to ECC and XTR, each with parameters which provide an equivalent level of security.

4 Software Design

The Software Design can be clearly described using the following figure.

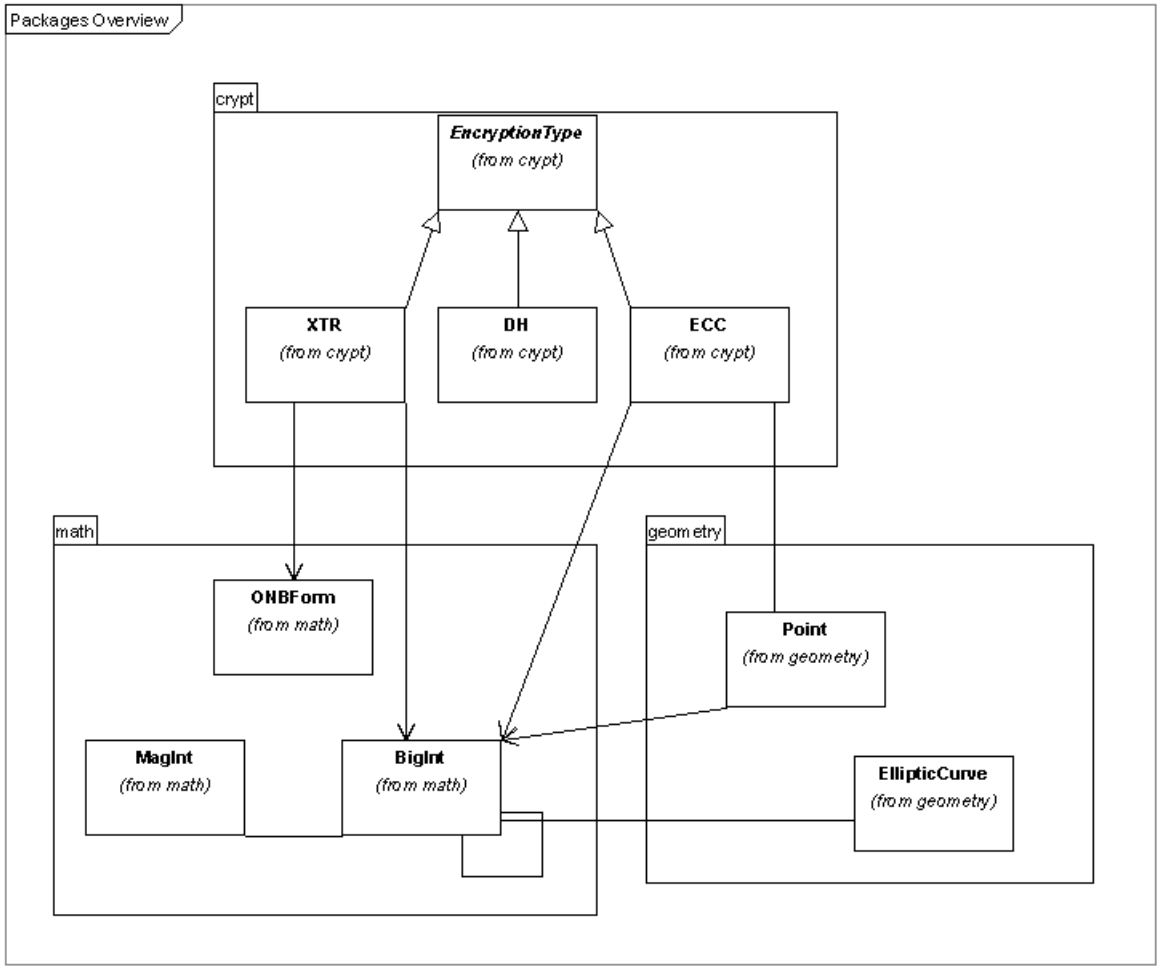

Figure 2: Software Design

The individual packages are described in the following sections with methods describing functionality that is specific to a given algorithm. Where needed the desicion making process is explained as well.

4.1 Math Library

The Math library is all contained within the class called BigInt. The BigInt is modeled after Java's BigInteger in that it is an Arbitrary Length Precision Integer. The underlying structure of the BigInt is an array of unsigned Java primitive integers and will be refered to as the BigInt's magnitude. The least significant digit is the zeroeth element of the array. Each BigInt contains the magnitude and a single integer to represent the sign. A 1 indicates that the BigInt is positive, a 0 represents a BigInt with a value of 0, and a sign of -1 indicates a negative BigInt.

The following subsections will describe in detail how select methods were implemented. This will ensure the reader has a solid understanding of exactly how the BigInt was implemented as well as describe specific details about the algorithms behind the code.

4.1.1 negate()

Possibly the simplist method to implement on a BigInt is the negation method. As stated before, there is an attribute which defines the BigInt's sign. The negation of a BigInt simply multiplies the sign by -1. This allows for the positive, negative and zero cases to be valid.

The method takes in no arguments and returns a copy of the BigInt for which it was called on with the exception that its sign is multiplied by -1.

4.1.2 add(), subtract() and multiply()

The addition method was the first method implemented due to its simple nature. Addition takes in a BigInt object and adds it to *this* and returns a new BigInt which represents the sum of the two. It is really nothing more than elementary addition that one would learn in elementary school. [6] describes these in great deatail. Given two arbitrarily long numbers, an *augend* and an *addend*, we find the sum by breaking the number up into individual digits and adding them together, starting with the least significant digit. If the sum of this single precision addition is greater than the base of

the digits, we add 1 to a carry bit and move to the next least significant digit. Doing this simple addition again, we note the carry bit, if it is set, we add 1 more to the sum of the single precision digits. We do this until all digits have been added to their counterpart. If the carry bit is set at the end of this, we simply append a 1 to the front of the number which becomes the new most significant digit.

In the actual implementation there are a few factors that need to be considered. In the beginning of the method, there is an error checking section which takes into account the zero cases which could give the addition method slower than necessary results. Also considered is the case where each, or either of the BigInts passed in are negative. This of course is simply a call to a negate method.

It was also stated earlier that the implementation was an array of Java primitive ints. A contradiction arises when it was stated that an addition could have a carry bit. The solution to this is that when we do addition, we actually take the int value that is in an element of the magnitude and we replace it with the equivalent long value. Doing this for the augend and the addend gives us ample magnitude to note the carry bit. If the sum of the numbers is larger than 32 bits, we mask out the upper 32 bits and set the carry bit.

Subtraction is a very simple extension of addition. To perform a subtraction, we simply take the addition of a negated value. This is a very simple modification and requires minimal coding. It takes as its argument a BigInt and returns the difference between *this* and the BigInt being passed in.

Both subtraction and addition are described in depth in section 14.2.2 of [6].

Multiplication is a very similar method. However there are some notable differences. As in addition, multiplication is the same multiplication that was taught in elementary school. We simply multiply the first digit in the second factor and multiply it by each individual digit of the first factor. Once this is done, we move to the next digit in the second factor and so on until all digits in one factor are multiplied by all digits in the other factor. We simply take the sum of these to get our result. Similar rules apply to multiplication regarding zero cases and ones cases and are accounted for in my implementation. Multiplication is described in depth in section 14.2.3 of [6].

Multiplication takes in a BigInt as its only argument. It returns the product of *this* and the BigInt being passed in.

4.1.3 divide()

The division algorithm used in this implementation can be found in [6]. This implementation uses a technique which is intended to reduce the harsh costs absorbed when using a division method. The gist of it is that the method attempts to digit shift the numerator until it only goes into the denominator 1 time. It is not appropriate for me to get into the method in this paper, but suffice it to say, division is still the worst of the basic arithmetic algorithms.

Of note should be the method called division2Over1(BigInt). This method is called on a BigInt which is 2 digits long and is passed in a BigInt which is 1 digit long. This is used in conjuction with Note 14.22 of [6] and a set of methods called normalize(int) and unnormalize(int) to exploit the speed of division of small BigInts.

Division takes in a BigInt and returns the quotient of *this* and the BigInt passed in with no remainder. A separate method called *divideAndRemainder()* also takes in a BigInt but returns an array of BigInts which includes the quotient and the remainder. The algorithm for division is discussed in more detail in section 14.2.5 of [6].

4.1.4 binaryExtended(), double(), half()

Many times in DH and ECC we need to find the inverse of a number modulo another number. Doing this requires that we use the Euclidean Algorithm. The Euclidean Algorithm in its normal form is not ideal for the number of times we are using this method, not to mention the magnitude of numbers that we are dealing with in this project.

The method called binaryExtended takes in 2 BigInts x and y. It computes integers a and b such that $ax + by = d$, where $d = gcd(x, y)$. This is different from normal extended Euclidean in that it eliminates costly multiple precision divisions at the expense of more iterations. The returned values

are returned as an array of BigInts. Binary Extended GCD Algorithm is discussed in more detail in section 14.4.3 of [6].

The Binary Extended Euclidean Algorithm [6] gives us a bit of a short-cut that will save a significant amount of time. This algorithm saves costly divisons at the expense of increased iterations. As described before, division is the one of the more computationally intense methods. Any opportunity to avoid division is highly desireable. The way [6] is able to get rid of the divisions is by replacing them with halving given an even number.

Binary halving and doubling is very simple. Both take no arguments and return double or half of *this*. The actual method in its simplist form is one of the quickest calculations that a machine can execute. Doubling is simply a left shift of bits, where division by 2 is simply a right shift of bits. Given the nature of the methods I am using, I believe that this is the best use of division in the BigInt class.

The method for half and doubling can be described simply as doubling, halving can be implied as the contrary. For each digit in the BigInt, we simply need to bit shift the digit to the left and note the carry bit. On the next digit, do the same bit shift operation and add 1 if the carry bit is set and note the next carry bit. Repeat these steps to the last digit and add 1 to the length of the number if the final digit has a carry bit set.

4.1.5 modExp(), montgomeryMultiplication()

As stated before, one of the most computationally intensive methods that can be used is division. And the most fundamental method that can be used in DH is Modular Exponentiation. Using Modular Exponentiation as one normally thinks of Modular Exponentiation will give us very large BigInt calls to division. We must remember that for our tests, we will be using BigInts which are 4096 bits long.

This brings us to section 14.46 of [6]'s implementation for Montgomery Multiplication. MongtomeryMultiplication() takes in a modulous, and x value, a y value, and the value $-m^{-1}(mod\ b)$ and returns $xyR^{-1}(mod\ m)$. This is a method which, through division and modular arithmetic of the base of our digits, is able to return $xyR^{-1}(mod\ p)$. Modular Arithmetic of our base is

as simple as taking the least significant digit of our BigInt. Division of our base is as easy as digit shifting to the right. This can be compared to base 10 division which is taught to elementary students learning math.

Using Montgomery Multiplication effectively allows us to remove the actual mod method call which would normally be needed for Modular Exponentiation. This gain in time is considered essential for encryption algorithms.

4.1.6 getRandomNBitsLong()

In many instances in the cryptographic schemes that we are talking about, it is necessary to find a random number which follows certain specific boundaries. My random number generator is very simple and requires little explanation. For the most part, I simply take a java.math.Random instance and use it to populate my digits of a BigInt. The signature for getRandomNBitsLong(size, java.math.Random) takes in a parameter for the size in bits of the new BigInt as well as a Pseudo Random Number Generator from Java's library.

4.2 EncryptionInterface

The EncryptionInterface is a definition of methods that any algorithm must implement in order to participate in Key Generation protocols. The EncryptionInterface has several key methods. The method createGToTheAModP() takes a size for Alices random multiplier. While the method returns void, it sets an internal variable which is equivalent to $g^a (mod\ p)$.

The method keyGen() takes a random seed, size for prime p and a size for generator g. This is the step which is done by the desktop machine as this contains the most time consuming computations. In our situation, Alice is the desktop machine.

The method keyGen1() takes a randomizer seed and a size for Bob's multiplier b. In our situation, this is done by Bob who generates his secret multiplier and performs $g^b (mod\ p)$.

The method keyGen2() is performed by both Bob and Alice and takes an object which will be a BigInt, a Point on a curve or a number in Optimal Normal basis Form for normal DH, Elliptic Curve DH and XTR DH respectively. The method also takes a string to decide if Alice or Bob are performing this method. After both parties have completed this step, they will have $g^{ab}(mod\ p)$ and thus have swapped enough information to have a common key without sharing any sensitive information.

4.3 DH Implementation

The basic algorithm for DH is quite simple and easy to understand. The math behind it deals with the understanding of the properties of modular exponentiation.

As described in an earlier section, the values of p and g represent the public and private keys. Using these gives us the ciphertext from the plaintext and vice versa.

Using the Modular Arithmetic algorithm's and supporting methods, it is quite easy to see that DH is a very simple algorithm by definition. As described earlier, this is known as the Discrete Logarithm problem and is an one-way function.

4.4 Elliptic Curve Implementation

The elliptic curve library will consist of the communication protocol to exchange the appropriate public information. When the public information is established, each of the parties will utilize methods in the math library to come up with random integers and calculations for the private key information.

The next few subsections deal with the classes that make up the Elliptic curve arithmetic.

4.4.1 Point, add(), multiply()

The point class is quite simple and can be described as simply a wrapper class around a BigInt array representing a point in two-dimensional space with a little more to accomodate Elliptic curves. Each point contains an X and a Y coordinate as well as curve that is associated with it.

The most significant methods associated with this class deal with point multiplication and point addition. These methods are the reason that the point needs to know what curve it belongs to. Addition of two points is described in [2] section A.10.1. The subtraction of a point Q = (x, y) from a point P = (a, b) is simply addition of the point -Q = (x, -y). Multiplication is described in section A.10.3 of [2] .

4.4.2 EllipticCurve, isPointOnCurve(), generateRandomPoint()

The elliptic curves are defined by the formula $y^2 = x^3 + xa + b$ are non-supersingular curves and are believed to be cryptographically strong. The choice is given to us in step 9 to make $a = b = c$ which will make things easier on us for now. So the Elliptic curve in this situation is made up of a point, a BigInt representing c and a modulus prime p.

The functionality of the class comes from its ability to generate a random point. In depth description of the method can be found at [2] section A.11.1. Given a point in 2d space, the elliptic curve class can determine if the point is on the curve by simply filling out both sides of the non-supersingular curve equation.

For the actual encryption of the data, ECC is using a modified version of Diffie-Hellman. Alice in this case will initiate the transmission by creating a public curve and a public starting point. She then will create a random BigInt A and multiply the public point P by A. The resulting point PA is sent over to Bob who is able to take that new point and multiply it by his random BigInt B to get the result BPA. Alice takes Bob's product BP and multiply it by her BigInt A to get ABP. The end result is that each side has a shared secret key but it was not shared in public.

4.5 XTR and ONBForm

As stated before, XTR is an alternate algebra which is defined in my library as ONBForm, or Optimal Normal Basis Form. This ONBForm is the basis for the algebra and boasts free exponentiation time.

The significant methods in the class ONBForm are as follows: add(), cor235iii(), isInGFP(), multiply(), square(), and subtract(). Each of these methods define the functionality as described in Section 2.

4.6 Test Bed

The test bed consists of a semi-automated means of executing the set of tasks described in the metrics portion of the deliverables list in the next section. The measurements of running time for the desktop and running time for the mobile device as appropriate are recorded using the standard System.getCurrentTimeMillis() method and subtracting a pre-operation measurement from a post-operation measurement.

The test bed is driven from the limited device and uses simple communication protocols to send appropriate values to the limited device. As an example: take a DH algorithm.

First a user must initiate the DH test from the Desktop. This will begin calculating the public keys p, g. Since the desktop is powerful enough, it is acceptable for it to calculate the values that Bob would calculate in order to complete the calculations that are depended upon Bob. The values for p and g are stored to a file and the desktop then completes Alice's role

Getting the results to the limited device is a bit difficult. Due to some network issues, the original plan of transferring the data to the phone would not work out. The toolkits provided by Sprint and Java did not provide ample technical support. Debugging network transmissions proved very difficult. The original plan was to send the public parameters through a socket connection exactly as key exchange would happen. Again, the SDK mixed with the limitd functionality of Sprint's network was not ample for figuring this out.The end solution was to hard code the data into the classes and redeploy

the code.

Using the limited device, the user must select which iteration, algorithm, size and number of iterations allowed by the build in user interface.

5 Experimental Results

The results of the tests run can be found in the next division of this section. All times are given in milliseconds. The tests were divided into each algorithm and then each bit length for each algorithm. Each algorithm has 2 bit lengths that the tests were run in. Normal DH was run at 4096 and 2048 bits. ECC was run at 783 and 585 bits. XTR was run at 683 and 341 bits. These numbers have been determined to give 4096 and 2048 bit security levels respectively (section 4.4 of [5]).

Initial observations show us that the key generation process for ECC takes 7 times longer than XTR and DH was approximately 105 times longer than ECC. The fact that ECC is faster than DH and XTR is faster than ECC was to be expected.

As we look to generation of $g^a(mod\ b)$ we see something a bit unexpected. First we see that XTR is more than 20 times faster than ECC. The surprise is that ECC is actually twice as costly as DH. Some investigation suggests that this is because ECC has many more smaller calculations which, in my implementation causes many more objects to be created. One of the greatest lessons that I have learned from this project is that object creation is highly costly when using the Java language. See the future enhancements section for more information on what could be done to fix this. It should be noted that this discovery does not contradict the work and studies done on ECC. Even if ECC turns out to be slower than DH, the papers claim that ECC uses smaller digits. They don't all directly say that the calculations are faster, though it could be assumed.

There is a java.math.BigInteger iteration that was run to see just how fast Java could perform. It seems that BigInteger is in fact much faster than the library that I have implemented. As you can see from figures, my library took about 25 times longer than the BigInteger implementation. If we had

a J2ME port of BigInteger, this could be an ideal solution for the problem that this paper is addressing.

2048/585/341 in milliseconds

	Key Exchange: Phone	Key Exchange Desktop	parameter Generation Desktop
DH	DNF	9479.7	8963307.9
ECC	DNF	21357.9	84585.9
XTR	8688974.8	822.75	12601.4

4096/783/683 in milliseconds

	Key Exchange: Phone	Key Exchange Desktop	parameter Generation Desktop
DH	DNF	1 iteration(18795344)	DNF
ECC	DNF	44093.7	260343.9
XTR	DNF	4606.25	135298.2

Figure 3: Average Results for 2048 and 4096 bit security levels

The relative comparisons of the time trials, shown above, give us a great understanding of how the algorithms compare to each other. We must also look at isolated times and notice the trends. First let us take a look at DH. We see that the 2048 bit DH took on average 149 minutes to complete the Key Generation stage. This does not include any computation on the part of the phone. This means that two parties would have to wait more than 2.5 hours before any actual communications can occur. This is obviously unacceptable. As we look at 4096 bit DH, we see that only 1 key generation iteration took more than 5 hours to complete. This is not even close to acceptable. As a result we have decided that this implementation is unacceptable at this time. The further calculations were stopped and given DNF as "Did Not Finish".

5.1 Data

$iteration$	$KeyGen$	$g^a (mod\ p)$	$g^{ab} (mod\ p)$
0	715031	9922	9641
1	10559188	10000	9765
2	11417485	9734	9469
3	19982532	9984	9734
4	8485344	10125	9938
5	12560422	9719	9468
6	7686453	9828	9532
7	4608000	10078	3828
8	8689406	9797	9563
9	4929218	9860	9609

Figure 4: 2048 bit Diffie-Hellman

$iteration$	$KeyGen$	$g^a (mod\ p)$	$g^{ab} (mod\ p)$
0	18795344	74422	72312
1	>172800000	DNF	DNF
2	DNF	DNF	DNF
3	DNF	DNF	DNF
4	DNF	DNF	DNF
5	DNF	DNF	DNF
6	DNF	DNF	DNF
7	DNF	DNF	DNF
8	DNF	DNF	DNF
9	DNF	DNF	DNF

Figure 5: 4096 bit Diffie-Hellman

$iteration$	$KeyGen$	$g^a(mod\ p)$	$g^{ab}(mod\ p)$
0	25953	406	390
1	95375	391	390
2	74703	391	390
3	35015	391	391
4	11703	406	406
5	36891	390	391
6	7219	406	390
7	136750	391	391
8	16000	390	407
9	20688	406	640

Figure 6: 2048 bit Diffie-Hellman using BigInteger

$iteration$	$KeyGen$	$g^a(mod\ p)$	$g^{ab}(mod\ p)$
0	107828	20968	21016
1	10328	20531	20641
2	35922	20578	22454
3	105343	20235	20375
4	127812	21500	21625
5	136360	20562	20672
6	33516	20734	20766
7	153984	20610	20844
8	110422	23547	22109
9	24344	20563	26828

Figure 7: 585 bit ECC

$iteration$	$KeyGen$	$g^a(mod\ p)$	$g^{ab}(mod\ p)$
0	286921	43047	43078
1	19422	43469	43907
2	171813	46468	48485
3	10047	46359	43968
4	9266	43094	43485
5	622891	45234	42906
6	394750	43000	43359
7	687766	43281	43719
8	22657	43218	43485
9	377906	45219	43093

Figure 8: 783 bit ECC

$iteration$	$KeyGen$	$g^a(mod\ p)$	$g^{ab}(mod\ p)$
0	28734	813	812
1	17094	734	735
2	22234	844	859
3	5250	844	860
4	3750	891	891
5	2812	703	703
6	16468	828	844
7	8641	844	703
8	13765	813	875
9	7266	922	937

Figure 9: 341 bit XTR

$iteration$	$KeyGen$	$g^a (mod\ p)$	$g^{ab} (mod\ p)$
0	18703	4688	4688
1	331031	4609	4656
2	145593	4391	4437
3	99843	4532	4562
4	23828	4750	4828
5	89578	4453	4500
6	422297	4828	4546
7	37938	4687	4610
8	56531	4453	4563
9	127640	4641	4703

Figure 10: 683 bit XTR

$iteration$	$g^b (mod\ p)$
0	8931633
1	8121933
2	9005609
3	9166692
4	9643583
5	7772234
6	8948961
7	7853797
8	8517770
9	8927536

Figure 11: Bob doing 341 bit XTR

iteration	$g^b (mod\ p)$
0	53182641
1	DNF
2	DNF
3	DNF
4	DNF
5	DNF
6	DNF
7	DNF
8	DNF
9	DNF

Figure 12: Bob doing 683 bit XTR

5.2 Recommendation

The purpose of this paper and project was to come up with a final recommendation for what would be appropriate for small devices at this time.

I feel that it is safe to say that assuming XTR is as secure as DH and having a faster Math library, I would use XTR over ECC and DH. XTR was very difficult to figure out based on the original paper but was very easy to implement once and understanding was reached.

5.3 Lessons Learned

1. As stated before, the performance of my library was not up to par with the performance of BigInteger. Maybe take BigInteger and move it over, it seemed to be faster in some cases. Perhaps replace only parts of BigInteger that were slow.

2. In the proposal I suggested that the implementation would be the easiest part of the project. This was a gross underestimation of the amount of code that I would be writing. Reviewing my schedule versus the actual time to complete, we see that the code took over one year to

complete. The largest delay was the distance factor I was working with. In retrospect, I should have started the project much earlier so that I would be able to have a good head start on the implementation by the time the summer of 2004 started. This could have given me the momentum to finish implementation and leave the easy part of the paper to do on my own remotely.

6 Future Enhancements

Future enhancements of this project could include several things. First of all, the use of BigInteger for the actual math library would probably be a good first step. As the data shows, BigInteger was significantly faster than the library that I developed. Doing this would require copying the source for BigInteger into the developers source library and math package. The developer would need to remove all errors that occur. This may be a trial and error ordeal which may take awhile.

A more extreme approach to fixing the timing issues would be to not use Java at all. As mentioned earlier, I suspect that most of the timing problems came from the creation of objects in Java. Using C would not give the developer this problem. The issues that arise with this solution is getting the code over to the small device. The phone that I was using was Java enabled so I could easily push Java programs onto it. If a developer were to use C, they would have to come up with a way to get the programs on to the phone. With the phone used for this project, that would be very difficult. As phones become more accessible, this may be an option.

Still another approach would be to use a PDA of some sort. PDA's are a more mature environment for running programs and are a bit more sofisticated than cell phones. This may lead to faster processors which would make these tasks much faster. Some future work which would not take much effort would be to run the tests on such a device.

7 Conclusion

For information on what specifically was implemented, please see section 3.3. It describes:

1. The underlying Math library
2. DH using 3 different arithmetics (integer, elliptic curve, and XTR)
3. PK Generation on the desktop
4. Metrics system

All of this was written in Java. BigInteger was used only as a test vehicle. All of the above was new code.

The implementation described in this paper attempted to run time trials on alternative solutions to the problem of efficiently exchanging keys over insecure communication channels. Discussed were two alternatives to the discrete logarithm problem as a solution to the lack of efficiency. ECC and XTR both provided alternative algebras which gave us a faster yet equally secure implementation. The implementation showed a significant increase in bit-size efficiency when using ECC, and an even greater increase in bit-size efficiency when using XTR. The time trials, however, revealed that ECC was in fact faster than XTR at the individual encryption steps. Possible causes were discussed in previous sections. Using this data and other research, we should be closer to an ideal situation for exchanging keys in a timely manner. While the tests here were not entirely complete as defined, they do reveal a trend that will no doubt be helpful in future encryption schemes.

A User's Manual

A.1 Running the Desktop

The final project comes in the form of a zip file. Unzip the file to an appropriate directory. To run the tests, go to the LimitedDeviceSecurity

folder. To compile the source, run *javac -cp . tests/*.java*. This will put all of the appropriate class files in the classes directory. These tests are used on the server side for timing purposes. To run these files, *java -cp . test/RunGenericDesktop*. This will print the usage output for the main in the RunTests class. The arguments that are passed in determine which tests and how many tests are run. For instance, the argument string "0 1234 2048 10" will run DH with a bit size of 2048 10 times with a random seed of 1234 and will output the results.

A.2 Running the Phone

Installing the program on the phone is more of a problem than one would have expected. Sprint phones do not have a way to get the programs onto the phone directly from the computer. The protocol defined is called the Over The Air Protocol(OTA). SprintPCS provides poor documentation on getting programs to the phone. Getting this to work requires a web server. SprintPCS did not mention this. You put the Java Archive file(JAR) as well as a descriptor file (JAD) onto the server. You must then send a text message to the phone with no subject and the URL of the JAD file as the body of the message. The web server must be configured to define how to handle both JAD and JAR files. This presented quite a problem for quite some time since it was not described anywhere how to actually do this. The actual lines of code that need to be put into the http.conf file for an apache server are as follows:

```
AddType application/java-archive .jar
AddType text/vnd.sun.j2me.app-descriptor .jad
```

It is difficult if not impossible to find this information formally written up.

Other problems that may be found when uploading a program. In order to reduce network traffec Sprint's network caches all of the network traffic for an undetermined amount of time. It may cause problems if you download a program then find that there was an error which can be fixed in a few short minutes. You will not be able to get the new file back to the phone. I have had situations where I would wait for 2-3 hours and still get the cached copy

of the file. The solution that I found for this issue was that I could send the text message to the phone with *?r=N* attached after the .jad where N is a random number. This tricked the network into thinking the file was different while not having any effect on the link location.

A.2.1 Creating the Jar

Using Netbeans 4.1 with the mobility pack installed will allow for very simple creation of an appropriate jar file. When code is ready to be deployed, you simply right-click on the project in the *Project* view of Netbeans and select *Properties*. This will bring up an options panel. Under *Deployment options* you must select a location for the jar file to be placed. When this is done, hit ok. Then again right-click the *Project* which will be deployed and select *deploy* from the popup menu.

The process that begins is the compile, build and move of the jar file which will be pushed to the Limited Device. The .jad file is also created at this point. The .jad file referes to the Java Archive Descriptor. This file tells a browser where it can find the archive file and some stats regarding size of the archive file.

The jar and jad files are now ready. Using the method described above can now be used to get the jar file onto the Limited Device.

References

[1] Alan kaminsky, personal communication, may 5, 2005.

[2] Ieee std 1363-2000. ieee standard specifications for public-key cryptography. January 30 2000.

[3] An intro to elliptical curve cryptography. http://www.deviceforge.com/articles/AT4234154468.html, July 2004. Retrieved March 20, 2006.

[4] S. M. Inc. Java 2 platform, micro edition - frequently asked questions. http://java.sun.com/j2me/faq.html, 2004. Retrieved June 18, 2004.

[5] A. K. Lenstra and E. R. Verheul. The XTR public key system. *CRYPTO 2000, Proceedings of the 20th Annual International Cryptology Conference*, 1880:1–19, 2000.

[6] A. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 2001.