5-1-2009

Extremely low-overhead security for wireless sensor networks: Algorithms and implementation

Michael Schab

Extremely Low-overhead Security for Wireless Sensor Networks: Algorithms and Implementation

By

Michael William Schab

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Computer Engineering

Supervised by

Dr. Marcin Lukowiak
Department of Computer Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, NY
May, 2009

Approved By:

Dr. Marcin Lukowiak
Primary Advisor – R.I.T. Dept. of Computer Engineering

Dr. Shanchieh Jay Yang
Secondary Advisor – R.I.T. Dept. of Computer Engineering

Dr. Manuel Lopez
Secondary Advisor – R.I.T. Dept. of Mathematics

Thesis Release Permission Form

Rochester Institute of Technology
Kate Gleason College of Engineering

Title: Extremely Low-overhead Security for Wireless Sensor Networks: Algorithms and Implementation

I, Michael William Schab, hereby grant permission to the Wallace Memorial Library to reproduce my thesis in whole or part.

Michael William Schab

Date

Dedication

This work would not have been possible without the support of my loving wife Jennifer, the never quit attitude of my brother Jeffrey and my parents who believed in and encouraged me throughout this long journey.

Acknowledgements

I would like to thank my RIT committee members, Dr. Lukowiak, Dr. Lopez and Dr. Yang for their advice and guidance in making this thesis a success. Additional gratitude is extended to Dr. Hu for the initial concept of this thesis.

Abstract

Recent advances in the development of compact microprocessors have brought forth new applications in the field of data acquisition and wireless communication. One of these applications is a compact sensor mote that has the ability to perform both data acquisition and transmission within a Wireless Sensor Network (WSN). Wireless sensor communication is susceptible to the same data security threats as traditional wireless networks. In an environment where sensors are broadcasting battlefield intelligence or patient biometrics, data confidentiality must be enforced utilizing some form of encryption.

Unlike traditional wireless networks, where the communicating devices have unrestricted access to power and memory, a wireless sensor has very limited resources. A wireless sensor consists of a battery designed to last an extended amount of time, therefore it is critical that the computation and transmission overhead involved in enforcing data security be optimized to preserve battery life.

The research presented in this thesis details a nesC/TinyOS implementation of the NTRUEncrypt PKCS executing on a Crossbow MICAz mote. Algorithm details regarding message encryption and decryption are analyzed along with optimization techniques that improve execution time and reduce memory size. A summary of performance metrics including execution time, power consumption and code size relative to the comparable ECIES-160 PKCS is also provided.

Table of Contents

List of Figures

List of Tables

Glossary

ECC	**Elliptic Curve Cryptography**
ECDH	**Elliptic Curve Diffie Hellman**
ECDSA	**Elliptic Curve Digital Signature Algorithm**
ECIES	**Elliptic Curve Integrated Encryption Scheme**
PKCS	**Public Key Crypto-System**
WSN	**Wireless Sensor Network**.

1 Introduction

Security within a network, either wired or wireless, typically involves the use of cryptography. Cryptography is the process that allows for secure communication over insecure channels [1]. A channel is a transport mechanism that facilitates the exchange of information from one location to another using a physical wire or through the air. In order for two parties to communicate securely, they must utilize a mathematically common element, know as a key, to transform and thereby disguise messages being transmitted be each other. The transformation of an intelligible message, called plaintext, into an unintelligible form, called ciphertext, is called encryption. The reverse process of converting the ciphertext back into a usable plaintext form is known as decryption. Whether the key is kept secret among the communicating parties, or a subset of the key is publically shared, defines the type of cryptosystem.

1.1 Private Key Cryptography

A cryptosystem in which the key is 'shared' and kept private between the communicating parties is known as a symmetric, or private key, cryptosystem. Since the key is used in both the encryption and decryption process, it must remain private otherwise the security between the two parties will be compromised. As long as a secure channel exists within the private key cryptosystem, private keys can be easily updated on a regular cadence to prevent an adversary from studying the cryptosystem and determining the private key [1].

1.2 Public Key Cryptography

Public key, or asymmetric cryptosystems are an evolutionary improvement over its private key counterpart. Introduced in 1976 by Diffie and Hellman [41], at the time when computer networking was at its infancy, Diffie and Hellman realized a need to establish secure communication over an insecure channel, i.e. a network connection. Unlike private key cryptosystem where the private key is shared between the communicating parties, public key cryptosystems do not share a common key. A party interested in establishing a secure conversation, obtains each others public keys, without the need of a private channel. The distribution of public keys can be done publically, without the means of a secure channel, thereby allowing anyone to communicate securely to the owner of the public key.

Though public key cryptosystems (PKCS) allow for quick and easy setup, their key size, number of bits, are typically larger than those of a private key cryptosystem of equivalent security level [46]. The public key cryptosystem, RSA [3] whose name was derived from the authors' initials, has a 1024-bit public key version with an equivalent symmetric security level of only 80-bits [47].

Since the PKCS keys are larger and more complex to create, the amount of computation power required for encryption and decryption is significantly greater than that of a private key cryptosystem. Figure 1.1 provides a high level overview of the process involved in sending an unsecure plaintext message from a sender, the conversion into an encrypted ciphertext, to the final decryption back to plaintext form that a recipient can understand. This is a one way function starting at the sender and ending at the receiver, therefore in order for a message to return back to the sender, the sender would have to provide their own public key to the receiver and the whole process would run in reverse.

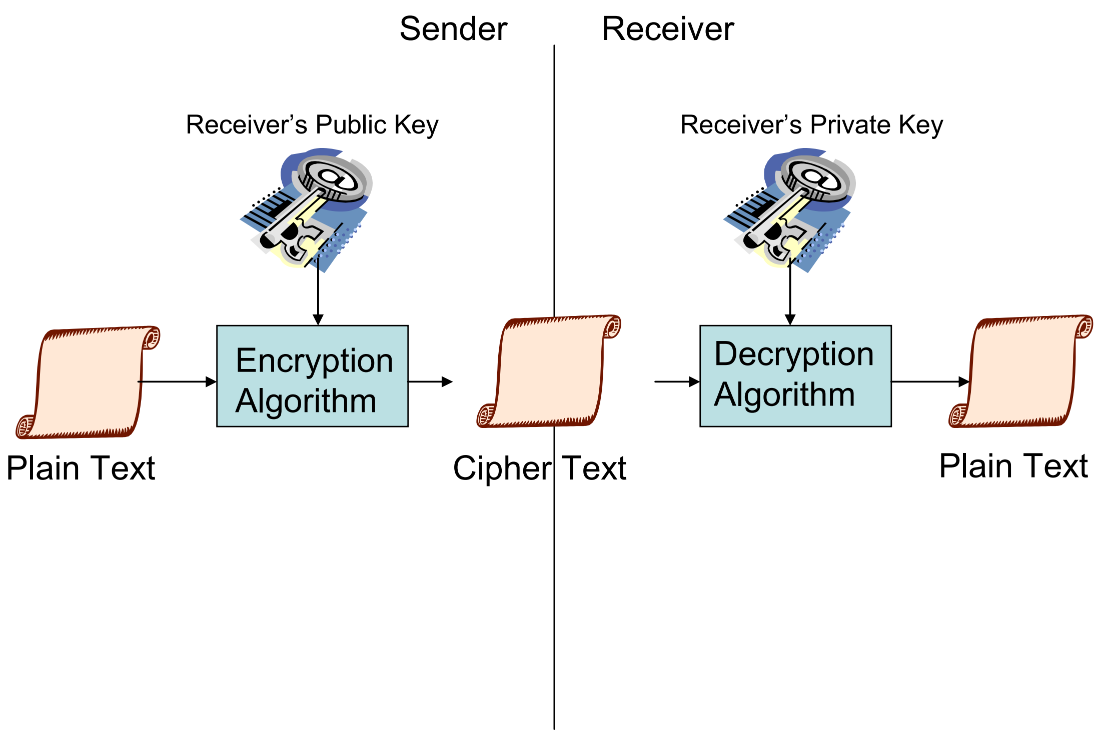


Figure 1.1: Public Key Encryption and Decryption

Advantages of Public Key Cryptosystems in Key Distribution

The small key size and low computational complexity of private key cryptosystems allow for fast execution times and low memory usage, but the inherent design of the cryptosystem does not facilitate updating the private key used between the two communicating parties.

Key distribution schemes allow for the secure deployment, or replacement, of the private keys being utilized in a cryptographic system. Secure distribution of these keys between the interested parties typically falls into one of three schemes: Key Distribution Center Scheme, Key Pre-distribution Scheme and Public Key Scheme [29].

The Key Distribution Center Scheme utilizes a central server in which each node interested in communicating on the network must access in order to obtain a private key. In a wireless environment, where nodes are placed in remote areas that rely on message hopping, access to a central server is not an option.

The Key Pre-distribution Scheme involves embedding keys within each node prior to deployment [29]. This can be a universal key or multiple keys stored within each sensor. The universal key, though not requiring much memory space, would easily compromise the security of the network if an adversary were to capture the node and obtain the common network key. Having each node contain multiple keys, a unique key pair per node, reduces the probably of an adversary determining the correct 'active' key, however, storing multiple keys per node increases memory size. In a wireless sensor network, such as a battle field, where node counts could be substantial, storing unique keys for per node communication is impractical [18].

Lastly, the public key scheme eliminates the issues involved with the aforementioned schemes [29]. Due to the asymmetric property of PKCS, sensor nodes do not need to contain any pre-distributed keys. The advantage of a public key cryptosystem over a private key cryptosystem is there are no private transactions required prior to establishing secure communication [3]. Not having to establish a private transaction means wireless sensors can be deployed at will and secure communication can be established immediately. If a sensor is compromised as a result of an adversary, new public keys can be easily distributed.

1.3 Security on Resource Constrained Devices

Wireless Sensor Network (WSN) applications range from sensors collecting battle field intelligence, to the monitoring of patient vitals such as heart rate and blood pressure. Transmission of confidential data, such as sensitive medical information must be protected from fraudulent activities such as alterations to treatment procedures or drug dosages [5].

Wireless devices within a WSN typically have limited resources including battery life, memory size and processing power. Maximizing battery life is extremely important in environments where sensors are deployed only once and never serviced again. Though transmission and reception of information usually requires the most energy in a WSN [34], the extra processing cycles imposed by an encryption scheme can actually consume more power than communication [9]. Every effort should be directed towards optimization of the cryptosystem code to reduce power consumption.

Data security in resource constrained devices, such as wireless sensors, has traditionally been solved using private key cryptosystems such as MiniSec [6] and TinySec [7]. These private-key based cryptosystems are popular due to their low energy consumption and fast execution times, but sacrifice security. An alternative cryptosystem that provides enhanced security at the expense of increased computational complexity is the asymmetric, or public key, cryptosystem.

While PKCS appear to be superior in regards to network compromises from an adversary, they do have some deficiencies. Several publications argue that a PKCS, though secure, are not realistic for use in wireless nodes due to their excessive computational overhead [29][30]. An excellent example is the famous RSA PKCS [3]. It is extremely secure, but is not a viable option for resource constrained devices due to its high computational requirements [19]. The Elliptic Curve Cryptography (ECC) standard is an alternative PKCS well suited for resource constrained devices that offers the equivalent security to RSA, but with faster execution times and reduced memory size. ECC [11] is based on the difficulty of the Discrete Logarithm Problem, but on an elliptic curve. Given two points P and Q, it’s believed computationally infeasible to find a number k such that Q = kP, see [13].

NTRUEncrypt is a relatively new PKCS that suggests faster execution times and requires less memory, for an equivalent security level, than ECC and RSA [24]. Conceived in 1996, NTRUEncrypt is a latticed based PCKS that features short, easily created keys with fast execution times and low memory requirements [2].

1.4 Related Work

Challa et al work comparing NTRUEncrypt to RSA clearly shows NTRUEncrypt to have significant performance gains over RSA [16]. Investigation into research benchmarking NTRUEncrypt, to the very comparable ECC cryptosystem, is close to non existent. The closest example by Driessen et al [15], provides an excellent comparison between the NTRUEncrypt based key signature algorithm, NTRUSign, and the comparable ECC equivalent Elliptic Curve Digital Signature Algorithm (ECDSA). Driessen et al research supported their statement of "NTRUSign is superior to the other signature schemes when comparing signature generation and verification time" [15]. Much of the performance gains of NTRUSign were from use of trinary polynomials and Karatsuba [40] variants to obtain a 9x performance increase over ECDSA [15].

A different approach, by Buchmann et al [14], that enhances the performance of NTRUEncrypt's fundamental operation of polynomial multiplication, involves finding bit patterns within polynomials [14]. By identifying repeating bit patterns, the number of additions required to compute the product of two polynomials can be reduced, thereby lowering execution time.

1.5 Thesis Objectives

This thesis focuses on the details involved in the software implementation of NTRUEncrypt on a resource constrained device including the comparison of NTRUEncrypt performance metrics relative to the equivalent ECC version referred to as Elliptic Curve Integrated Encryption Scheme (ECIES). Since ECIES is an accepted standard under IEEE 1363-2000 [42], much research detailing implementation optimizations along with execution times on various hardware platforms has been published [12][13][17]. Though the details of ECIES will not be discussed, an overview of ECC will be provided for completeness. The intent of this thesis is to provide an in depth understanding of the steps involved in the implementation of NTRUEncrypt and how its performance relates to ECIES.

The rationale to compare NTRUEncrypt with ECIES was based on several publications by Challa et al [16] and Wang et al [5] that showed ECC based PKCS to have greater performance than that of RSA [3]. Since ECC based PKCS are very popular in the resource constrained embedded microprocessor market, comparing ECIES to NTRUEncrypt on a resource constrained device was chosen.

1.6 Organization

The remainder of this thesis provides an overview of elliptic curves in Section 2 including the theory involved in the ECIES PKCS. Section 3 introduces NTRUEncrypt and provides a detailed explanation, including examples, of how to implement NTRUEncrypt. Section 4 provides data obtained from an actual implementation of both NTRUEncrypt and ECIES on a wireless sensor, including execution time, RAM and ROM size, and power consumption. Software implementation details, along with Karatsuba optimization techniques used by other researchers, of NTRUEncrypt are discussed. Section 5 concludes this thesis by providing a summary of findings along with suggestions for future work.

2 Elliptic Curves

2.1 Elliptic Curve Groups over Real Numbers

An elliptic curve is a smooth graph that does not contain any self-intersecting points along its curve. Elliptic curves can be used to define a group given the following form with *a, b, x* and *y* all being real numbers [8].

$$y^2 = x^3 + ax + b \tag{1}$$

The shape of an elliptic curve is controlled by the value selection of variables *a* and *b*. For an elliptic curve to be valid for use in cryptography, the curve must not contain any repeated factors and therefore must satisfy the following equation.

$$4a^3 + 27b^2 \neq 0 \qquad (2)$$

Over a number field, a cubic, i.e. a polynomial of degree three, can have at most three roots, where a root is defined as a point where the curve crosses the x-axis. For real numbers, the roots of a cubic fall into two categories, degenerate and non-degenerate. A degenerate case occurs if any two roots of the cubic coincide with one another, such as the case where the two curve cross at the x-y intersection as shown in Figure 2.1 below:

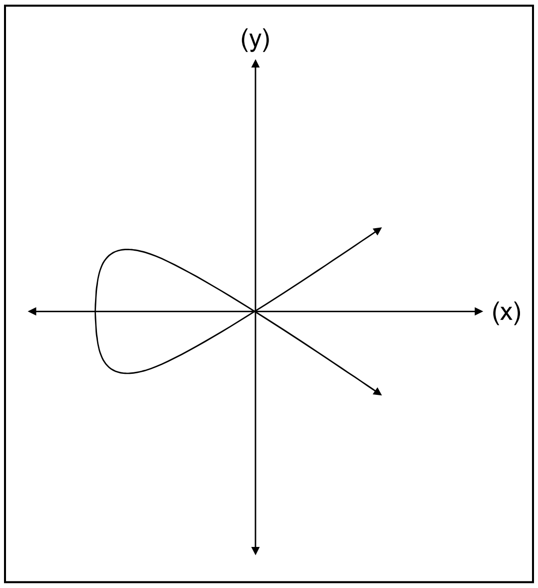


Figure 2.1: Degenerate Curve

The following examples cover several non-degenerate curve applications.

2.2 Point Addition for Points with Different x-Coordinates

The addition of two points on an elliptic curve can be represented both mathematically, as well as, graphically. The addition of two points, *F* and *G*, for points with different x-coordinates, is performed by drawing a straight line through points *F* and *G* until the line intersects the curve at a third point –*H*, because the slope of such line is finite. The next step is to take the reflection of point –*H* across the x-axis to obtain *H*. The resultant *H* is the summation of *F* and *G*, see [8].

Given:
$F = (x_F + y_F)$, $G = (x_G + y_G)$

Find (Addition of F and G):
$H = F + G$

Solve:

$$s = \frac{(y_F - y_G)}{(x_F - x_G)}$$

$$x_H = s^2 - x_F - x_G$$

$$y_H = s(x_F - x_G) - y_F$$

Answer:
$H = (x_H, y_H)$

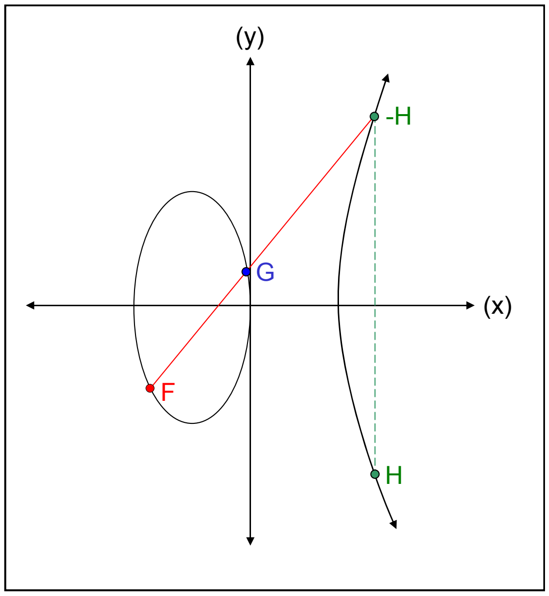


Figure 2.2: Elliptic Curve Point Addition When $F \neq -G$

2.3 Point Addition for Points with the Same x-Coordinates

In this case, the previous point addition technique is invalid. The drawing of a line through these two points results in a vertical line intersecting only two points on the elliptic curve, instead of three. In this case, the elliptic curve group defines a third, infinity point O, with the two points having additive inverses, see [8].

Given:
$F = (x_F + y_F)$, $G = -F = -(x_F + y_F)$

Find (Addition of F and $-F$):
$H = F + (-F)$

Answer:
$H = F + (-F) = O$

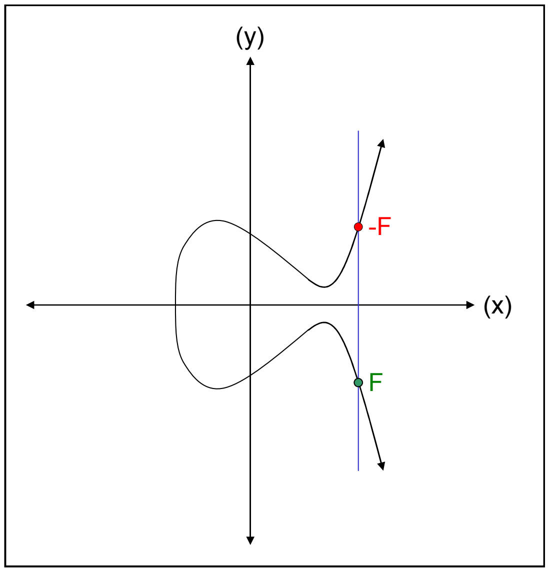


Figure 2.3: Elliptic Curve Point Addition when $F = -F$

2.4 Point Doubling ($F + F = 2F$, $F_y \neq 0$)

Doubling of a point involves adding a point to itself. The process involves drawing a line through point F, tangent to the elliptic curve. The line will intersect a second point, $-H$, on the elliptic curve if $F_y \neq 0$. The reflection of point $-H$ across the x-axis results in the product point H, see [8].

Note: If $F_y = 0$, then the result of doubling F is the infinity point O, see [8].

Given:
$F = (x_F + y_F)$

Find (2F):
$H = 2F = F + F$

Solve:
$$s = \frac{(3x_F^{\ 2} + a)}{(2y_F)}$$
Note: *a* represents one of the domain parameters for the elliptic curve.

$$x_H = s^2 - 2x_H$$
$$y_H = -y_F + s(x_F - x_H)$$

Answer:
$H = (x_H, y_H)$

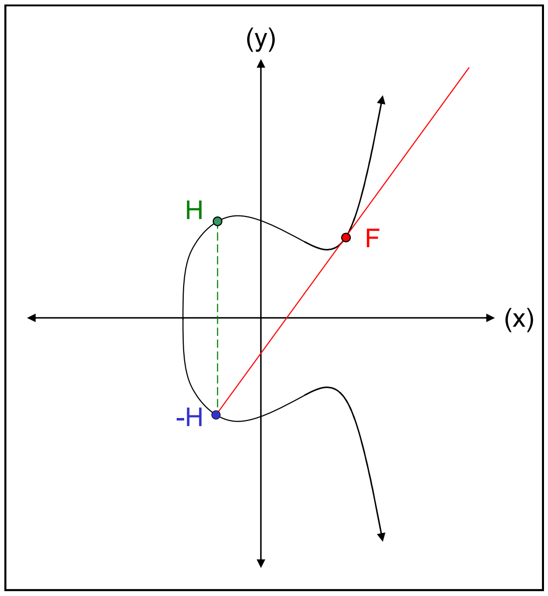


Figure 2.4: Elliptic Curve Point Doubling when $F + F = 2F,\ F_y \neq 0$

2.5 Point Doubling (F + F = 2F, F$_y$ = 0)

When attempting to double a point where the y-coordinate = 0, i.e. $F_y = 0$, the tangent line to the elliptic curve will never intersect a second point on the curve. The tangent line to the elliptic curve is actually vertical and results in the product, *2F*, equaling the infinity point *O*, see [8]. In this example, there are three possible points for *F* where $F_y = 0$.

Given:
$F = (x_F + y_F)$

Find (2F):
$H = 2F = F + F$

Answer:
$H = 2F = O$, since $F_y = O$

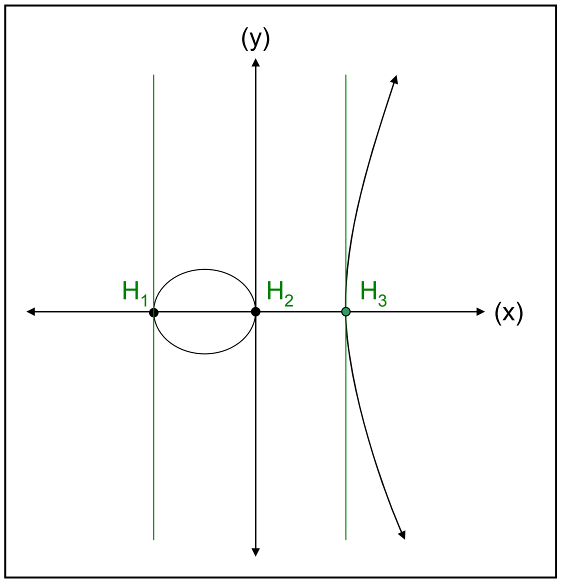


Figure 2.5: Elliptic Curve Point Doubling when $F + F = 2F, F_y = 0$

2.6 Point Multiplication

Point multiplication on an elliptic curve utilizes a scalar parameter k multiplied by a point F on the elliptic curve, such that $kF = H$. Using the previous techniques of point addition and doubling, point multiplication is possible. The following example illustrates the technique [8].

Given $k = 17$ and point F, find $H = kF$.

$$H = kF = 17F = 2(2(2(2F))) + F$$

Note: If $F_y = 0$ for point F, the following substitutions are made in the above equation based on the point doubling equation, Figure 2.5, when $F_y = 0$.

Given $2F = O$, then $2F + F = 3F = F$. Continuation of this pattern reveals:

$$H = kF = 4F = 3F + F = O,\ 5F = 4F + F = F, \ldots\text{etc}$$

In summary, if scalar k is even, $H = O$, otherwise $H = F$.

2.7 Elliptic curves Groups Over F_p

Cryptographic use of elliptic curves requires a finite field F_p instead of real numbers. Extending the definition of the elliptic curve from real numbers to a finite field of integers restricts the field size to *p* values through the use of modulo arithmetic. For an elliptic curve, $E(F_p)$ all computations in F_p are reduced modulo *p*, therefore an elliptic curve containing non negative integer variables *a, b, x* and *y* can be defined as follows [32]:

$$y^2 \equiv x^3 + ax + b \pmod{p} \tag{3}$$

Just as with elliptic curves over real numbers must not contain repeated factors, elliptic curves over a finite field F_p must also maintain this property.

$$4a^3 + 27b^2 \pmod{p} \neq 0 \tag{4}$$

Elliptic curves, F_p, consist of a finite number of points thereby making them suitable for cryptosystems. In order to utilize a F_p for cryptography, equation (4) must be satisfied for randomly selected values of *a*, *b,* and *p*.

The following illustrates this step given randomly chosen variables $a = 4$, $b = 3$ and $p = 5$:

$$y^2 \equiv x^3 + ax + b \pmod{p}$$

$$y^2 \equiv x^3 + 4x + 3 \pmod{5}$$

Selecting point (2, 4):

$$4^2 \pmod{5} \equiv 2^3 + 5(2) + 3 \pmod{5}$$

$$16 \pmod{5} \equiv 8 + 10 + 3 \pmod{5}$$

$$1 \equiv 1$$

Since point (2, 4) satisfies the equation, it is a valid point on the elliptic curve F_5 and can be used for cryptography.

The graphical representation presented in the last section is not feasible for an elliptic curve with a finite number of points. The point addition and doubling techniques are the same except a reduction modulo p is performed [8]. There is a point addition and point doubling algorithm described as follows.

2.8 Point Addition

Given:
$F = (x_F + y_F),\ G = (x_G + y_G)$

Find (Addition of F and G):
$H = F + G$

Solve:

$$s = \frac{(y_F - y_G)}{(x_F - x_G)} \ (\text{mod}\ p)$$

$$x_H = s^2 - x_F - x_G \ (\text{mod}\ p)$$

$$y_H = s(x_F - x_G) - y_F \ (\text{mod}\ p)$$

Answer:
$H = (x_H, y_H)$

Point Doubling

Given:
$F = (x_F + y_F)$

Find (2F):
$H = 2F = F + F$

Solve:

$$s = \frac{(3x_F{}^2 + a)}{(2y_F)} \ (\text{mod}\ p)$$

Note: a represents one of the domain parameters for the elliptic curve.

$$x_H = s^2 - 2x_H \ (\text{mod}\ p)$$

$$y_H = -y_F + s(x_F - x_H) \ (\text{mod}\ p)$$

Answer:
$H = (x_H, y_H)$

Figure 2.6: Elliptic curve Point Addition for F_p

3 Elliptic Curve Cryptography (ECC)

Elliptic Curve Cryptography (ECC) was introduced by Neal Koblitz and Victor Miller in 1985 [33] and is an accepted standard by IEEE under 1363-2000 & 1363a-2004 with security based on the difficulty of solving the discrete logarithm problem [8]. ECC utilizes a finite group composed of points (x, y) located on an elliptic curve with the encryption and decryption process based on the aforementioned point addition and multiplication.

3.1 Key Generation

ECC key generation starts by selecting a finite field elliptic curve $E(F_p)$ based on Equation (3). Given an elliptic curve E over a finite field F_p, let G be a point that has a prime order *n* within $E(F_p)$. These values can be used to generate cyclic subgroup $E(F_p)$, see [33]:

$$\{G\} = \{\infty, G, 2G, \ldots (n-1)G\}$$

3.1.1 Private Key(s)

The private key is an integer *k* that is selected at random from the interval [1, n-1].

3.1.2 Public Key

The public key, *Q*, is the private key *k*, multiplied by a random point *H* selected from the elliptic curve.

$$Q = kH \tag{5}$$

3.2 Encryption

Encryption in ECC starts with the desired plaintext message to send, *m*. The message, *m*, is converted into a point, *M*, in the finite field F_p and is encrypted by adding it to a randomly selected integer *k* multiplied by the recipient's public key *Q*, such that:

$$E_1 = M + kQ \tag{6}$$

In addition to E_1, the sender also calculates E_2 by multiplying the previously selected *k* with the random point *H*, such that:

$$E_2 = kH \tag{7}$$

The sender then transmits both point E_1 and E_2 to the recipient [33]. In order for the recipient to decode the ciphertext points, the two communicating parties must agree on a set of domain parameters, *T*, that are exchanged up front. The parameters contain a list of 6 items that relate the ciphertext to the original plaintext. The domain parameters are:

$$T = (p, a, b, G, n, h)$$

Where:

p	Size of field *F*
a	First value defining curve
b	Second value defining curve
G	Initial base point on curve
n	Order of point G
h	Cofactor

3.3 Decryption

To recover the original message, *m*, the recipient first must find point *M* on the elliptic curve given the following equation:

$$M = E_1 - kQ \tag{8}$$

Utilizing the domain parameters, *p, E, H* and *n*, the following substitutions are performed [33].

$$kQ = k(dG) = d(kG) = d(E_2) \tag{9}$$

Substitution of variables from (9) into (8) results in the following decryption equation for point *M*.

$$M = E_1 - d(E_2) \tag{10}$$

Since the domain parameters are public, the recipient can reconstruct the elliptic curve and locate the original message, *m*, given point *M*.

4 NTRUEncrypt Implementation

NTRUEncrypt was implemented in nesC and timing data was collected for each step of the cryptosystem including key generation, encryption and decryption execution times.

NesC [28] is a variant of the C Programming Language that is optimized to run on resource constrained devices.

Pseudo-code representing the actual nesC code developed in this thesis are represented as 'Algorithms' throughout the rest of this thesis unless otherwise noted.

4.1 Background

NTRUEncrypt, also known as the NTRU encryption algorithm, is a new PKCS relative to other cryptosystems and was just recently adopted into the IEEE P1363.1™/D12 [22] draft standard for PKCS in February of 2009. NTRUEncrypt was proposed by Hoffstein, Pipher and Silverman in 1996 and is based on ring theory [2]. Security of the cryptosystem relies on the difficulty of finding extremely short vectors within a lattice. It was developed in an effort to provide an efficient public key cryptosystem that required less system resources such as memory and CPU processing power, while still maintaining similar security to that of other PKCS. While the exact translation for the acronym NTRU is not exactly known, several rumored translations include "Number Theorists aRe Us" [1] and "N-Th Degree Truncated Polynomial Ring".

NTRUEncrypt is based on a 'Ring of Truncated Polynomials' represented by ring R below:

$$R = \frac{Z[X]}{(X^N - 1)} \tag{11}$$

The polynomials within the ring, *R*, consist of all truncated polynomials of degree N-1 having integer coefficients [31].

$$a(X) = \sum_{i=0}^{N-1} a_i X^i \in R = a_0 X^0 + a_1 X^1 + a_2 X^2 + \dots a_{N-1} X^{N-1} \tag{12}$$

The NTRU algorithm involves the addition and convolution multiplication of polynomials within *R*, see [31]. Addition of polynomials, denoted by the symbol '$\oplus$', is performed as follows utilizing traditional polynomial addition.

$$a(X) \oplus b(X) = \sum_{i=0}^{N-1}(a_i X^i + b_i X^i) \in R$$
$$= (a_0 X^0 + b_0 X^0) + (a_1 X^1 + b_1 X^1) + \ldots(a_{N-1} X^{N-1} + b_{N-1} X^{N-1}) \tag{13}$$

Since the resultant of any polynomial manipulation must remain within *R*, convolution multiplication of polynomials, or 'star multiplication', denoted by the symbol '$\otimes$', is performed the traditional way with the exception that the exponents must never exceed *N*. Constraining the polynomial to size *N* is accomplished by reducing and rotating exponents (i mod *N*).

$$a(X) \otimes b(X) = \sum_{\substack{n=0 \\ i+j \equiv n \bmod N}}^{N-1}(a_i X^i * b_i X^i) \in R = c_0 X^0 + c_1 X^1 + \ldots c_{N-2} X^{N-2} + c_{N-1} X^{N-1} \tag{14}$$

Constraining the polynomials to size *N* has an added benefit when implementing NTRU in software and/or hardware. Unlike traditional multiplication where two *N* sized polynomials multiplied together could potentially expand to 2*N* -1 in size, star multiplication limits the size to *N*, thereby requiring less memory.

The fundamentals of NTRUEncrypt are based on parameters *N*, *p*, and *q*. As mentioned above, *N* represents the number of degrees of the polynomial ring *R*. The parameters *p* and *q* represent the modulus values used throughout the encryption and decryption process. Both of these modulus values must maintain the following properties [2].

- The gcd(*p*, *q*) = 1, i.e. *p* and *q* must be relatively prime
- The value of *q* must be considerably larger than *p*

Utilizing this ring of polynomials and reducing modulo *p* and *q*, encryption and decryption can be performed [2].

4.2 Modulo Arithmetic

Modulo arithmetic is a fundamental reduction operation used in many steps of the NTRUEncrypt algorithm. Modulo reduction, referred to as 'mod', is the remainder value produced by the division of two numbers [44].

Given: $a = 23, b = 7$
Find: $c = 23 \bmod 7$

$$\frac{23}{7} = 3 \textit{ remainder } 2$$

$$23 \bmod 7 = 2$$

The general expression for modulo is:

$$a \bmod b$$

The concept of congruence can be also defined using modulo:

$$a \equiv c (\text{mod } b)$$

With congruency, given a constant *b*, the value *c* remains the same regardless of the value chosen for *a*. The following two values of 23 and 30 are considered congruent to each other.

$$23 \equiv 2(\text{mod } 7)$$

$$30 \equiv 2(\text{mod } 7)$$

If *a* and *b* have no common factors between them, then it is possible to find an inverse for *a*(mod *b*), such that [44]:

$$\mathrm{a} * \mathrm{c} \equiv 1 \ (\text{mod b})$$

Find the inverse of 2(mod 7):

$$11 * 2 \equiv 1 \ (\text{mod } 7)$$

The inverse of 2(mod 7) = 11 since 11 * 2 = 22 ≡ 1 (mod 7). The Extended Euclidean algorithm [22] may be used to find the inverse of any number [44].

4.2.1 Modulo of Negative Numbers

Modulo is defined as the difference between the largest integer multiple of the divisor that is less than the dividend. Modulo reduction of a positive number is straightforward:

$$23 \bmod 7 = 2$$

Modulo reduction of a negative number can be difficult to comprehend:

$$-23 \bmod 7 = 5$$

The confusion begins with the natural tendency to divide and transfer the 'sign' to the quotient. For the example of -23 mod 7, the largest integer divisor less than the dividend is -28, not -23. The number line in Figure 4.1 represents the graphical representation for both -23 mod 7 and 23 mod 7.

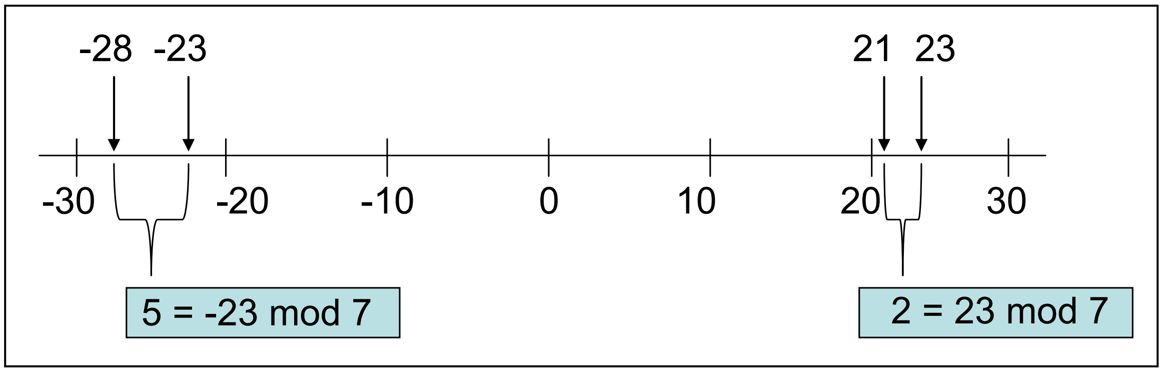


Figure 4.1: Number line representing positive and negative modulo

Understanding the concept of negative modulo is extremely important in the implementation of NTRUEncrypt since the private key polynomials contain negative coefficients. The incorrect usage of modulo reduction for negative numbers will result in unexpected polynomial coefficients throughout the NTRUEncrypt algorithm.

The modulo operator '%' in nesC, as well as other programming languages, simply transfers the 'sign' value of the dividend to the quotient, thereby resulting in an incorrect solution. As a result, a custom modulo function detailed in Algorithm 4.1, written in nesC, was used to correct this issue.

Algorithm 4.1: Mod(value, modValue)

```
Input:  value – Input to apply modulo
        modValue – Modulo value to apply

Output: Modulo reduced value

Init: retVal = value

1: If ( value = 0 )
2:     Return 0
3: If ( retVal + modVal <= 0 OR retVal – modVal >= 0)
4:     retVal = value % modVal
5: If ( retVal < 0 )
6:     retVal = retVal + modval
7: Return retVal
```

Minor optimizations to prevent the modulo '%' operation from occurring on values equal to zero or less than the 'modVal' where added in steps 1 and 3 of Algorithm 4.2.1.

4.3 Key Generation

Key generation in NTRU, just like all asymmetric cryptosystems, creates a private and public key pair. Generation of the private and public keys in NTRU begins with the selection of two polynomials *f* and *g* within ring *R* with coefficients being small relative to the large modulus *q*. Selection of the two polynomials *f* and *g* are constrained to the following criteria:

- Polynomial *f* must be invertible modulus of both *p* and *q*. More specifically [2]:

$$\begin{aligned} f_q \otimes f &\equiv 1 \,(\text{mod}\, q) \\ f_p \otimes f &\equiv 1 \,(\text{mod}\, p) \end{aligned} \tag{15}$$

- Polynomial *f* must contain d_f number of coefficients equal to '+1', d_f-1 number of coefficients equal to '-1' and the remaining $N\text{-}2d_f\text{-}1$ coefficients equal to '0'.

- Polynomial g must contain d_g number of coefficients equal to '+1', d_g number of coefficients equal to '-1' and the remaining $N\text{-}2d_g$ coefficients equal to '0'.
- To ensure security, generation of both polynomials f and g, need to be random generated by means of a Random Number Generator (RNG) or Index Generation Function (IGF) detailed in [22].

Note: The reason d_f contains an unequal number of '+1' and '-1', is due to the constraint of f to be invertible, since a polynomial $f(1) = 0$ can never be inverted. Polynomial g does not need to be invertible and therefore has an equal number of '+1's to '-1's [2].

Creation of the random polynomials used throughout the NTRUEncrypt PCKS must be generated in an unbiased manner to prevent an adversary from predicting future sequences. Section 9.2.1 of the P1363.1 draft standard details the requirements for generating random polynomials using either an index generation function (IGF) or a random number generator capable of producing unbiased outputs. For the purpose of this thesis, Algorithm 4.2 was written based on a random number generator. Since the MICAz mote contains a rolling counter that is set to zero upon reset, generating a truly random number given a constant seed was not feasible, though it was very useful for producing predicable outputs for debugging purposes. Implementation on a device that maintains time across resets would resolve this issue.

Algorithm 4.2: GenerateRandomPolynomial(*a(x), n, num_ones, num_neg_ones*)

```
Input:  n – Nth degree size of polynomial
        num_ones – Number of coefficients equal to '+1'
        num_neg_ones – Number of coefficients equal to '-1'

Output: a(x) = Polynomial with random coefficients equal to -1, 0, or +1

Init: pos = 0

1:  While ( num_ones OR num_neg_ones )
    {
2:      rVal = RandomNumberGenerator
3:      pos = rVal % n
4:      if (num_ones AND a(pos) = 0)
        {
5:          a(pos) = 1;
6:          num_ones = num_ones - 1
        }

7:      rVal = RandomNumberGenerator
8:      pos = rVal % n
9:      if (num_neg_ones AND a(pos) = 0)
        {
10:         a(pos) = -1
11:         num_neg_ones = num_neg_ones - 1
        }
    }
```

Since the only requirement to generate a random polynomial is an input count for the number of '+1' and '-1', Algorithm 4.2 loops through an *n* sized polynomial, inserting *num_ones* count of '+1' or *num_neg_ones* count of '-1' at random positions.

4.3.1 Private Key(s)

NTRUEncrypt requires the generation and storage of two private keys, f and f_p. The generation of the private key polynomial f utilizes Algorithm 4.2 with d_f number of '+1' and (d_f-1) number of '-1'.

The second private polynomial key, f_p, is the inverse calculation of f modulo p (f mod p). Calculation of this inverse, in the ring of truncated polynomials R, is performed using either the Extended Euclidean Algorithm (EEA) [22] or the Almost Inverse

Algorithm (AIA) [19]. Wilhelm's investigation into the performance of both algorithms suggests the AIA to have better performance and therefore was utilized in this thesis [23].

4.3.2 Almost Inverse Algorithm

By definition, a polynomial *b* is invertible, modulo *p*, if the resultant inverted polynomial *B* maintains the following property [31].

$$b \otimes B = 1 \pmod{p} \; (\bmod\, x^N - 1) \tag{16}$$

The work in presented by Silverman et al [20] presents two AIA implementations, one for modulo $p = 2$ and another for modulo $p = 3$. The combined AIA, Algorithm 4.3, was written to support both modulo reductions thereby reducing code space. The specific changes required to add $p = 2$ support to the base $p = 3$ algorithm are shown as **bolded** steps throughout Algorithm 4.3.

Algorithm 4.3: AIA(a(x), b(x), n, c)

```
Input:  a(x) – Polynomial to invert
        n – N^th degree size of polynomial
        c – Modulo value to apply to coefficients

Output: b(x) ≡ a(x)^-1 – Inverse polynomial of a(x) in (Z/cZ)[x] / x^n – 1; NoInverse; or
        ERROR

Init: k = 0, b(x) = 1, c(x) = 0, f(x) = a(x), g(x) = x^n – 1

Error Check:   If (c != 2 OR c != 3)
                    Return ERROR

1:      Loop:
        {
2:      While ( f(0) = 0 AND NumDegrees( f(x) != 0 ) )
        {
3:            $f(x) = \frac{f(x)}{x}$
4:            $c(x) = c(x) \otimes x$
5:            $k = k + 1$
        }
6a:     If ( NumDegrees( f(x) ) = 0 )
        {
6b:           If ( f(x) = 1 OR ( f(x) = -1 AND c = 3 ) )
7a:                Return $f(0)x^{(n-k)} \otimes b(x) \pmod{x^n-1}$
              Else
7b:                Return NoInverse;
        }
8:      If ( NumDegrees( f(x) ) < NumDegrees( g(x) ) )
        {
9:            Swap f(x) & g(x)
10:           Swap b(x) & c(x)
        }
10a:    If ( c = 3 AND ( f(0) = g(0) ) )
        {
11:           $f(x) = f(x) - g(x) \pmod{c}$
12:           $b(x) = b(x) - c(x) \pmod{c}$
        }
        ELSE
        {
13:           $f(x) = f(x) + g(x) \pmod{c}$
14:           $b(x) = b(x) + c(x) \pmod{c}$
        }
```

To demonstrate the property of inversion, an example detailing the process involved in finding the inverse of polynomial *b* using the AIA, Algorithm 4.3, is shown.

<u>Given:</u> $N = 5$, $p = 3$, $b = 1 + x^2 - x^4$
<u>Find:</u> $B = b^{-1}$

Tables 4.1-4.5, illustrate the number of 'rounds' required to find the inverse of $b = 1 + x^2 - x^4$. A 'round' is characterized by the variable *k* and is incremented whenever the conditional $f(0) = 0$ is satisfied for polynomial *f(x)* (Step 2 of Algorithm 4.3). Each column is labeled with the variable name referenced in the AIA Algorithm 4.3, with the column labeled 'Step' corresponding to the particular operation step in the algorithm. The first entry in Table 4.1 is the initial setup as shown in the 'Init' section of Algorithm 4.3.

k	b(x)	c(x)	f(x)	g(x)	Step
0	1	0	$1 + x^2 - x^4$	$-1 + x^5$	
			$-1 + x^5$	$1 + x^2 - x^4$	9
	0	1			10
			$x^2 - x^4 + x^5$		13
	1				14
			$x - x^3 + x^4$		3
		x			4
1					5
1	1	x	$x - x^3 + x^4$	$1 + x^2 - x^4$	

Table 4.1: k = 1 Round of AIA

k	b(x)	c(x)	f(x)	g(x)	Step
1	1	x	$x - x^3 + x^4$	$1 + x^2 - x^4$	
			$1 - x^2 + x^3$		3
		x^2			4
2					5
2	1	x^2	$1 - x^2 + x^3$	$1 + x^2 - x^4$	

Table 4.2: k = 2 Round of AIA

k	b(x)	c(x)	f(x)	g(x)	Step
2	1	x^2	$1 - x^2 + x^3$	$1 + x^2 - x^4$	
			$1 + x^2 - x^4$	$1 - x^2 + x^3$	9
	x^2	1			10
			$2x^2 - x^3 - x^4$		11
	$-1 + x^2$				12
			$2x - x^2 - x^3$		3
		x			4
3					5
3	$-1 + x^2$	x	$2x - x^2 - x^3$	$1 - x^2 + x^3$	

Table 4.3: k = 3 Round of AIA

k	b(x)	c(x)	f(x)	g(x)	Step
3	$-1 + x^2$	x	$2x - x^2 - x^3$	$1 - x^2 + x^3$	
			$2 - x - x^2$		3
		x^2			4
4					5
4	$-1 + x^2$	x^2	$2 - x - x^2$	$1 - x^2 + x^3$	

Table 4.4: k = 4 Round of AIA

k	b(x)	c(x)	f(x)	g(x)	Step
4	$-1 + x^2$	x^2	$2 - x - x^2$	$1 - x^2 + x^3$	
			$1 - x^2 + x^3$	$2 - x - x^2$	9
	x^2	$-1 + x^2$			10
			~~3~~ $- x -2x^2 + x^3$		13
	$-1 + X^2$				14
			$-1 - 2x + x^2$		3
		$-x + x^3$			4
5					5
			$1 -$ ~~$3x$~~		13
	$-1 -x + 2x^2 + x^3$				14
5	**$-1 -x + 2x^2 + x^3$**	$-x + x^3$	**1**	$2 - x - x^2$	

Table 4.5: k = 5 Round of AIA

The effects of modulo reduction throughout the AIA can be observed in Step 13 of Table 4.5. After the 'Addition' Step 13, the coefficient 3 was reduced to zero thereby decreasing the polynomial by one degree.

Algorithm 4.3 continues to increment variable *k* until the number of degrees of *f*(*x*) = 0 (Step 6a). Step 6b is evaluated next and is responsible for determining whether or not the polynomial is actually invertible. If *f*(*x*) != ± 1, Algorithm 4.3 is aborted and returns a value indicating no inverse exists (Step 7b). If an inverse is found, i.e. *f*(*x*) = ± 1, Step 7a is executed to reveal the inverted polynomial *B* as follows:

$$B = f(0)x^{(n-k)} \otimes b(x)\ (\operatorname{mod} x^n - 1) \quad \text{(Step 7a of Algorithm 4.3)}$$

$$B = (1)x^{(5-5)} \otimes (-1 - x + 2x^2 + x^3)\ (\operatorname{mod} x^5 - 1)$$

$$B = (1)(1) \otimes (-1 - x + 2x^2 + x^3)\ (\operatorname{mod} x^5 - 1)$$

$$\boldsymbol{B = -1 - x + 2x^2 + x^3} \qquad \text{(Inverse polynomial)}$$

As a final check for inversion, Equation (16) is used to verify polynomial *B* is the inverse polynomial of *b*.

$$b \otimes B = 1\ (\operatorname{mod} p)\ (\operatorname{mod} x^N - 1)$$

$$(1 + x^2 - x^4) \otimes (-1 - x + 2x^2 + x^3) = 1\ (\operatorname{mod} 3)\ (\operatorname{mod} x^5 - 1)$$

Using Equation (14) the convolution of polynomials within ring *R* is computed.

$$(-1 - x + 2x^2 + x^3 - x^2 - x^3 + 2x^4 + x^5 + x^4 + x^5 - 2x^6 - x^7) = 1\ (\operatorname{mod} 3)\ (\operatorname{mod} x^5 - 1)$$

$$-1 - x + x^2 + 3x^4 + 2x^5 - 2x^6 - x^7 = 1\ (\operatorname{mod} 3)\ (\operatorname{mod} x^5 - 1)$$

To keep the polynomial constrained to ring *R*, exponents ≥ *N* = 5, are rotated.

$$-1 - x + x^2 + 3x^4 + 2x^{5-5} - 2x^{6-5} - x^{7-5} = 1\ (\operatorname{mod} 3)\ (\operatorname{mod} x^5 - 1)$$

$$-1 - x + x^2 + 3x^4 + 2x^0 - 2x^1 - x^2 = 1\ (\operatorname{mod} 3)\ (\operatorname{mod} x^5 - 1)$$

$$-1 - x + x^2 + 3x^4 + 2 - 2x \quad - x^2 = 1 \,(\mathrm{mod}\, 3)\; (\mathrm{mod}\, x^5 - 1)$$
$$1 - 3x + 3x^4 = 1 \,(\mathrm{mod}\, 3)\; (\mathrm{mod}\, x^5 - 1)$$

A final modulo reduction of p = 3 proves B = b^{-1}

$$1 - \cancel{3x} + \cancel{3x4} = 1 \,(\mathrm{mod}\ 3)\; (\mathrm{mod}\ x^5 - 1)$$
$$\mathbf{1 = 1\,(mod\,3)\;(mod}\ \boldsymbol{x}^5 \mathbf{- 1)}$$

Therefore:

$$\mathrm{B} = \mathrm{b}^{-1}$$
$$\mathbf{(-1 - \boldsymbol{x} + 2\boldsymbol{x}^2 + \boldsymbol{x}^3) = (1 + \boldsymbol{x}^2 - \boldsymbol{x}^4)^{-1}}$$

4.3.3 Public Key

The NTRU public key calculation requires computation of another inverse polynomial f_q. The calculation is very similar to the private key calculation of f_p, with the only difference being the inverse calculation uses the larger modulo value q. Since q must be much larger than p, using Algorithm 4.3 exclusively to find the inverse f_q is not possible due to its modulo constraint of either p = 2 or p = 3. Closer examination of the large value q used in NTRUEncrypt, reveals that it is always a number base $\log_2$. This property of q derives a second version, Algorithm 4.4, of the AIA [15] that builds upon the original AIA Algorithm 4.3.

To illustrate how f_q is calculated, a value of q = 32 is chosen. The value 32 can be represented as a multiple of base 2 numbers such that:

$$32 = 2(2)(2)(2)(2) = 2^5$$

Since the number 32 is equivalent to $\log_2(32) = 5$, Algorithm 4.3 is utilized in the first step of calculating f_q by calculating polynomial f module p = 2, f_2. Following the exact same steps as the example in Tables 4.1-4.5, except with a modulo value of p = 2 instead of p = 3, f_2 is found. Using the resultant f_2 inverse polynomial and $\log_2(32)$ calls to

Algorithm 4.3, f_{32} is calculated using Algorithm 4.4. Closer examination of Algorithm 4.4, step 3, shows the original f_2, represented as a(x), being convolution multiplied by itself $\log_2(32)$ times to ultimately calculated f_{32}.

Algorithm 4.4: AIA_Q(a(x), b(x), n, c, q)

```
Input:  a(x) – Polynomial to invert
        n – Nth degree size of polynomial
        c – Modulo value to apply to coefficients
        q – Modulo value base 2

Output: $b(x) \equiv a(x)^{-1}$ – Inverse polynomial of $a(x)$ (mod $c$)

Init: $q = 2$
1:      While ( $q < c$ )
        {
2:          $q = q*2$
3:          $b(x) = b(x) \otimes (2 - a(x) \otimes b(x)) \pmod{q}$
        }
```

Since not all polynomials will have an inverse, a new random small polynomial f will need to be chosen and the inverse calculations, f_p and f_q, performed again until successful. Generating and testing for inversion is a time consuming process especially if an inverse is not found and the process needs to be repeated. An optimization that will be discussed in the next section eliminates the need to calculate the inverse polynomial f_p, thereby reducing key generation time and the storage space required for f_p.

Using polynomial g and inverse polynomial f_q from above, calculation of the public key, h, is found:

$$h = pf_q \otimes g \pmod{q} \tag{17}$$

Algorithm 4.5: CalcPublicKey(fq(x), g(x), h(x), p, q, n)

```
Input:  fq(x) – Secret polynomial key $f_q$
        g(x) – Secret polynomial $g$
        $p$ - NTRUEncrypt '$p$' parameter
        $q$ – NTRUEncrypt '$q$' parameter
        $n$ – Degree size of polynomial

Output: h(x) – Public key polynomial $h = pf_q \otimes g(\text{mod}\, q)$

1:  $h(x) = f_q(x) \otimes g(x)$ (mod $q$)
2:  For ($i$ = 0 to n)
    {
3:      $h(i) = (h(i) * p)$ (mod $q$)
    }
```

The first operation to compute the public key, *h(x)*, is to perform the star multiplication of $f_q(x) \otimes g(x)$ as shown in Step 1 of Algorithm 4.5 using Equation (14). Step 2 iterates through each position of *h(x)*, multiplying by *p* modulo reduction *q* as shown in Step 3.

4.4 Encryption

The encryption process converts a plaintext message *m* into a suitable ciphertext *e* that is broadcast to the recipient. Encrypting a message involves translating the binary plaintext message *m* into a polynomial message construct *M* with coefficients ranging from $-\frac{(p-1)}{2}$ to $\frac{(p-1)}{2}$ using Table 4.6:

Input: Plaintext message *m* binary bit pattern	**Output: Message construct *M* trinary pattern**
{0,0,0}	{0,0}
{0,0,1}	{0,1}
{0,1,0}	{0,-1}
{0,1,1}	{1,0}
{1,0,0}	{1,1}
{1,0,1}	{1,-1}
{1,1,0}	{-1,0}
{1,1,1}	{-1,1}

Table 4.6: Binary to trinary conversion

The transformation from binary to trinary increases the message density thereby allowing a larger binary plaintext message *m* to be stored in message construct *M* at a 3-to-2 ratio.

In order to provide plaintext awareness [2], a blinding polynomial *r* is generated with the same criteria as *g* per the following requirements:

- Polynomial *r* must contain d_r number of coefficients equal to '+1', d_r number of coefficients equal to '-1' and the remaining N-$2d_r$ coefficients equal to '0'.
- To ensure security, generation of polynomial *r* needs to be random generated by means of a Random Number Generator (RNG) or Index Generation Function (IGF) detailed in [22].

Once a valid blinding polynomial *r* is selected, it is multiplied with the public key *h*. The final step is to add message *m* mod *q* to the product of $r \otimes h$ to produce the encrypted ciphertext message *e*.

$$e = r \otimes h + m \ (\text{mod } q) \tag{18}$$

Algorithm 4.6 was written to perform encryption which performs one polynomial star multiplication and a polynomial addition.

```
Algorithm 4.6: Encrypt(r(x), h(x), m(x), e(x) p, q, n)

Input:  r(x) – Blinding polynomial key r
        h(x) – Public key polynomial h
        m(x) – Message polynomial to encrypt
        p - NTRUEncrypt 'p' parameter
        q – NTRUEncrypt 'q' parameter
        n – Degree size of polynomial

Output: e(x) – Ciphertext polynomial $e = r \otimes h + m \pmod{q}$

1:  $e(x) = r(x) \otimes h(x) \pmod{q}$
2:  $e(x) = e(x) + m \pmod{q}$
```

4.5 Decryption

Decryption reverses the encryption process to obtain the original message *m*. The first step in the decryption process is to obtain the intermediate polynomial *a,* from the ciphertext *e,* using the private polynomial keys f_p and f [2].

$$a = f \otimes e \pmod{q} \tag{19}$$

To ensure a high probability of decryption success, the coefficients of polynomial *a* must be adjusted so that all coefficients range from $-\frac{q}{2}$ to $\frac{q}{2}$ instead of 0 to q. Details regarding why this step is required is discussed in the next section under Decryption Failures.

Once the coefficients are 'balanced', polynomial *a* is star multiplied by the inverse private key polynomial f_p and reduced modulo *p*.

$$b = f_p \otimes a \pmod{p} \tag{20}$$

Next, the original message construct *M* is derived by an additional module *p* reduction.

$$M = b \pmod{p} \tag{21}$$

A subtle step, that is not very well documented, is needed to correctly recover message construct *M*. Various coefficients throughout the algorithm are negative and therefore there exist instances where the multiplication of two negatives coefficients or modulo reduction of a negative coefficient results in a positive coefficient. Since the decrypted message construct *M* must maintain a trinary form, its coefficient need to be 'balanced' around zero the same way polynomial *a* was. Algorithm 4.7 was written to rotate a polynomial into a trinary form.

Algorithm 4.7: ConvertToTrinary(a(x), p, n)

```
Input:   $a(x)$ – Polynomial to convert
         $p$ - NTRUEncrypt '$p$' parameter
         $n$ – Degree size of polynomial

Output:  a($x$) – Polynomial with trinary coefficients (-1, 0, 1)

Init: $maxCoef = 2$

1:  for ( $i = 0$ to $n$ )
2:  {
3:      $a(i) = \text{Mod}( a(i), p )$
4:      if ( $a(i) >= maxCoef$ )
5:          $a(i) = a(i) - p$
6:      if ( $a(i) <= ( -1 * maxCoef )$ )
7:          $a(i) = a(i) + p$
8:  }
```

The final step in the decryption process is to convert the message construct *M* from a trinary form back to the original binary plaintext message *m* using Table 4.7.

Input: Message construct *M* trinary pattern	**Output: Plaintext message *m* binary bit pattern**
{0,0}	{0,0,0}
{0,1}	{0,0,1}
{0,-1}	{0,1,0}
{1,0}	{0,1,1}
{1,1}	{1,0,0}
{1,-1}	{1,0,1}
{-1,0}	{1,1,0}
{-1,1}	{1,1,1}

Table 4.7: Trinary to binary conversion

4.6 Decryption Failures

A decryption failure is when the plaintext message output of the decryption step does not match the original input message. NTRUEncrypt is unique to other PCKS in that, with standard parameters, ciphertext can fail to decrypt [48].

Given a polynomial of the form:

$$a(X) = a_0X^0 + a_1X^1 + a_2X^2 + \dots a_{N-1}X^{N-1} \pmod q$$

The minimum and maximum coefficients are defined as:

$$Min(a(X)) = \min\{a_0, a_1, \dots, a_{N-1}\}, \qquad Max(a(X)) = \max\{a_0, a_1, \dots, a_{N-1}\}$$

The width of a polynomial is the polynomial's range:

$$Width(a(X)) = Max(a(X)) - Min(a(X)$$

To understand decryption failures, Equation 19 utilizes the following substitutions:

$$a = f \otimes e \pmod q$$

$$\textit{Substitution for: } e = r \otimes h + m \pmod q$$

$$a = f \otimes r \otimes h + f \otimes m \pmod q$$

$$\textit{Substitution for: } h = pf_q \otimes g (\text{mod } q)$$

$$a = f \otimes r \otimes pf_q \otimes g + f \otimes m \ (\text{mod } q)$$

$$\textit{Reduce: } f \otimes f_q \equiv 1 \ (\text{mod } q)$$

$$a = r \otimes p \otimes g + f \otimes m \ (\text{mod } q)$$

Since the coefficients of *r*, *g*, *f* and *m* are small, relative to *q*, their products will have a small *Width* [50]. The objective is to find a modulo *q* interval that results in a successful decryption, such that:

$$a(1) = r(1) \otimes p(1) \otimes g(1) + f(1) \otimes m(1) \ (\text{mod } q)$$

If an incorrect modulo *q* interval is chosen, a decryption failure will occur since the modulo q is incorrectly 'zeroing' out the coefficients of the polynomial. A successful decryption occurs when the degree-one-or-higher terms of the inverse polynomial f_p cancel out, instead of the module q 'zeroing' out the terms.

A 'gap' decryption failure occurs when $Width \geq q$, while a 'wrap' decryption failure occurs if $Width < q$. With either failure, the resulting message *m* will be incorrect by some multiple of *q* mod *p* [50]. The probability of a decryption failure can be significantly reduced if the modulo *q* interval is adjusted such that the polynomial coefficients are centered on zero and range from $-\frac{q}{2}$ to $\frac{q}{2}$ instead of 0 to *q*. Algorithm 4.8 was written to 'balance' a polynomial by forcing all coefficients outside into the range of $-\frac{q}{2}$ to $\frac{q}{2}$.

Algorithm 4.8: BalancePoly(f(x), n, q)

```
Input:  f(x) – Polynomial to balance
        n – Degree size of polynomial
        q – NTRUEncrypt 'q' parameter

Output: f(x) – Balance polynomial around $\frac{q}{2}$

Init: maxCoef = $\frac{q}{2}$
1:  For (i = 0 to n)
    {
2:      If( f(i) > maxCoef )
3:          f(i) = f(i) – q
4:      If( f(i) < -maxCoef )
5:          f(i) = f(i) + q
    }
```

4.7 Why Decryption Works

For a cryptosystem to be useful, it must consistently be able to reproduce the original plaintext message, from ciphertext, without any errors. The steps involved in the encryption and decryption process constrain the various polynomials to modulo p or module q space. The encryption process combines several small modulo p constrained polynomials together to form a larger modulo q constrained polynomial [37]. Since the initial polynomials are constrained to a small modulo p, modulo reduction by q has no affect on the polynomial coefficients. However, the steps involved in decryption start with a polynomial constrained to the large modulo q space and work backwards reducing the polynomials to the smaller modulo p space.

Starting from the encrypted ciphertext e and working backwards, Figure 4.1 illustrates the necessary variable substitutions to obtain the original plaintext c.

$$\begin{tikzcd}
a = f \otimes e \,(\text{mod } q) \arrow[dr, "\text{Equation (19)}"'] & & h = pf_q \otimes g (\text{mod } q) \arrow[dl, "\text{Equation (17)}"] & \\
& \boxed{e = r \otimes h + m (\text{mod } q)} \arrow[d, "\text{Equation (18)}"] & & \\
& a = pf_q \otimes r \otimes f \; + m \,(\text{mod } q) \arrow[d] \arrow[d, shift left=2] & & \text{[Substitution 1]} \\
& f_q \otimes f \equiv 1 \,(\text{mod } q) \; \text{Equation (15)} \arrow[dd, "\text{Reduce (mod q)}"] & & \\
c = f_p \otimes a \,(\text{mod } p) \arrow[dr, "\text{Equation (20)}"'] & & & \\
& a = p \otimes r + m \arrow[d] & & \text{[Substitution 2]} \\
& c = f_p p \otimes r \,(\text{mod } p) + m \arrow[dr, "\text{Reduce (mod p)}"] & & \text{[Substitution 3]} \\
& & \boxed{c = m \,(\text{mod } p)} &
\end{tikzcd}$$

Figure 4.2: Flow chart from encrypted ciphertext *e* to plaintext *c*

Starting with the original ciphertext equation (18), substitutions for variables *e* and *h* using Equations (19) and (17) respectively results in an expanded definition of variable *a* [Substitution 1] in Figure 4.2. The multiplicative inverse identity of Equation (15) and the substitution of the plaintext definition of Equation (20) reduces the definition of variable *a* to three variables *p*, *r* and *m* [Substitution 2]. The modulo *q* reduction of this step does not affect any coefficients as long as the rules pertaining to *q* being large relative to *p* are enforced. The final [Substitution 3], performs a modulo reduction *p* which cancels out all terms except for the desired plaintext message *m*.

5 NTRUEncrypt Optimizations

The core mathematics involved in almost every step of the NTRU algorithm involves some form of polynomial multiplication. Techniques that reduce the execution time required for polynomial multiplication directly improves the efficiency of NTRU.

5.1 Star Multiplication

Recall standard star multiplication of two polynomials, *a(x)* and *b(x)*, contained in ring *R* and both being *N*-degrees in size, is performed utilizing aforementioned Equation (14). The pseudo-code version used throughout the NTRUEncrypt algorithm is detailed in Algorithm 5.1.

Algorithm 5.1: Star_Multiply(a(x), b(x), c(x), n, d, m)

```
Input:  a(x) – First polynomial
        b(x) – Second polynomial
        n – Size of polynomials a(x) and b(x)
        d – Size of output polynomial h(x)
        m – Modulo value to apply to coefficients

Output: $c(x) = a(x) \otimes b(x)$ in $(Z/(c)Z)[x] / (x^n – 1)$

Init: $c(x) = 0$, $val = 0$, $exp = 0$, $i = 0$, $j = 0$

1:      For ( $i = 0$ to $n$ )
        {
2:          If ( $a(i)$ != 0 )
            {
3:              For ( $j = 0$ to $n$ )
                {
4:                  If ( $b(j)$ != 0 )
                    {
5:                      $exp = (i + j) \pmod{d}$
6:                      $val = c[exp] + a(i)b(i)$
7:                      $c[exp] = \mathrm{Mod}(val, m)$
                    }
                }
            }
        }
```

The convolution of two polynomials using traditional multiplication in code utilizes two nested loops as seen in Steps 1 and 3 of Algorithm 5.1. Step 5 calculates and stores the rotation amount, *exp*, of the exponents within ring *R*. Steps 6 and 7 perform the multiplication of the input polynomials, reduction modulo *m*, and adds the resulting product to the correct polynomial coefficient. One very simple optimization to Algorithm 5.1 is shown in Steps 2 and 4. These two 'If' checks prevent any further execution of the polynomial multiplication in Steps 5-7 if the coefficient of any polynomial equals zero. Though this optimization has no affect on non-zero coefficients, 'Zero Check' reduces the execution time required for generating the NTRUEncrypt private key polynomials since approximately 1/3 of the secret polynomial coefficients for *f*, *g* and *r* are chosen to equal zero per Section 4.3. Algorithm 5.1 requires very little RAM to execute, but has a growth rate of $O(n^2)$

5.2 Karatsuba

The Karatsuba Algorithm (KA) was introduced in 1963 by Anatolii Alexeevitch Karatsuba [40] as a technique to improve the time required to multiply two polynomials. The technique reduces the number of coefficient multiplications involved in standard multiplication techniques, but at the expenses of utilizing extra additions [39].

5.3 Karatsuba Multiplication

In contrast to traditional multiplication, KA requires more RAM to execute, but decreases execution times by decomposing each input polynomial into several smaller polynomials.

Weimerskirch et al [39], demonstrates the steps required to multiply two, one degree, polynomials based on KA as follows given polynomials *f(x)* and *g(x)* in one pass:

$$f(x) = f_0 + f_1 x, \qquad g(x) = g_0 + g_1 x$$

Components of polynomials *f(x)* and *g(x)* are extracted and stored in three intermediate variables, fg_0, fg_1 and r_{fg} as follows:

$$fg_0 = f_0 g_0, \quad fg_1 = f_1 g_1, \quad r_{fg} = (f_0 + f_1)(g_0 + g_1)$$

The resulting product of *h(x) = f(x)g(x)* is performed as follows:

$$h(x) = fg_1(x^2) + (r_{fg} - fg_0 - fg_1)x + fg_0$$

In comparison to traditional multiplication, which requires $n^2 = 2^2 = 4$ multiplications and $(n - 1)^2 = (2 - 1)^2 = 1$ additions, the KA method requires 4 additions and 3 multiplications. KA therefore saves 1 multiplication, but required 3 additional additions [39].

5.4 Recursive Karatsuba (KM)

Polynomial multiplication using KA for only one iteration, as seen above, can be extended to incorporate recursion. The recursive KA method, referred to as KM in this thesis, is nothing more than 'splitting' the polynomials f(x) and g(x) into two halves and applying the aforementioned one-pass KA algorithm repeatedly until the polynomial size, n, is equal to 1.

The technique to apply KM for multiplying any two, even, n-size polynomials was modified slightly to accommodate odd values of *n* as suggested by Silverman [43]. The major change involved using the smaller of the two polynomial sizes after the split rather than just *n*/2. Application of KM is outlines in Algorithm 5.2.

Algorithm 5.2: KM_STAR(F(x), G(x), H(x), n, c, cutOff)

```
Input:  F(x) – First polynomial
        G(x) – Second polynomial
        H(x) – Product of F(x) and G(x) in $(Z/(c)Z)[x] / (x^n – 1)$
        n – Size of polynomials
        c – Modulo value to apply to coefficients
        cutOff – Minimum value of n for KM to execute

Output: $H(x) = F(x) \otimes G(x)$ in $(Z/(c)Z)[x] / x^n – 1$

Init: ls = n / 2, hs =  n – ls, count = 0,

1:  count = count + 1
2:  If ( n = 1 )
    {
        If ( F(0) = 0 OR G(0) = 0 )
3a:         Return 0;
        Else
3b:         Return H(x) = Mod( F(x)*G(x), c) in $(Z/(c)Z)[x] / (x^n – 1)$
    }
4:  If ( n < cutOff )
5:      Star_Multiply(F(x), G(x), H(x), n, c)
6:  $F_L = F_0x^0 ... F_{ls}x^{ls}$
7:  $F_H = F_{ls+1}x^{ls+1} ... F_{hs}x^{hs}$
8:  $G_L = G_0x^0 ... G_{ls}x^{ls}$
9:  $G_H = G_{ls+1}x^{ls+1} ... G_{hs}x^{hs}$
10: $FG_L = F_LG_L$ = KM_STAR($F_L, G_L, FG_L, ls, c$)
11: $FG_H = F_HG_H$ = KM_STAR($F_H, G_H, FG_H, ls, c$)
12: $R_{fg} = (F_L + F_H)(G_L + G_H)$
        = KM_STAR($(F_L + F_H), (G_L + G_H), R_{fg}, hs, c$) in $(Z/(c)Z)[x] / (x^n – 1)$
13: $H(x)$ = Mod$( (FG_H(x^{2ls}) + (R_{fg} – FG_L – FG_H)(x^{ls}) + FG_L))$ in $(Z/(c)Z)[x] / (x^n – 1)$
```

Slight changes to the recursive version of KM presented by Weimerskirch et al [39] included constraining polynomials *f(x)* and *g(x)* to the ring, *R*, of truncated polynomials in steps 3b, 5, 12, and 13 or whenever multiplication of polynomials occurred. Also, a coefficient reduction modulo *c* is performed for any operation on a coefficient.

Given: $f(x) = -1 – x$, $g(x) = -1 – x$, $n = 2$, $c = 3$, threshold = 1

Find: $h(x) = f(x)g(x) \pmod c$ in $(Z/(c)Z)[x] / (x^n – 1)$ using KM

```
Init: $ls = n / 2 = 2 / 2 = 1$,
      $hs = n - ls = 2 - 1 = 1$

6:  $F_L = -1$
7:  $F_H = -1$
8:  $G_L = -1$
9:  $G_H = -1$
10: $FG_L = F_L G_L = (-1)(-1) = 1 \pmod 3$
11: $FG_H = F_H G_H = (-1)(-1) = 1 \pmod 3$
12: $R_{fg} = (F_L + F_H)(G_L + G_H)$
        $= (-1 - 1)(-1 - 1)$
        $= 4 \pmod 3 = 1 \pmod 3$
13: $H(x) = (FG_H(x^{2ls}) + (R_{fg} - FG_L - FG_H)(x^{ls}) + FG_L)$
        $= (1(x^2) + (1 - 1 - 1)(x^1) + 1) \pmod 3$
        $= x^2 - x + 1 \pmod 3$ in $(Z/(c)Z)[x] / (x^n - 1)$
        $= x^{2-n} + 2x + 1 \pmod 3$ in $(Z/(c)Z)[x] / (x^n - 1)$
        $= x^{2-2} + 2x + 1 \pmod 3$ in $(Z/(c)Z)[x] / (x^n - 1)$
        $= 1 + 2x + 1 \pmod 3$ in $(Z/(c)Z)[x] / (x^n - 1)$
        $= 2 + 2x \pmod 3$ in $(Z/(c)Z)[x] / (x^n - 1)$
16: $H(x) = (2 + 2x) \pmod 3$ in $(Z/(c)Z)[x] / (x^n - 1)$
```

Figure 5.1: KM Example

5.5 KM Performance

KM splits the input polynomial into two segments for each iteration of the algorithm. Figure 5.2 details the polynomial sizes (N) for each iteration for an initial $N = 107$. For Iteration 1 of KM, the original $N = 107$ is divided into two smaller polynomials of $N = 53$ and $N = 54$. Each iteration of the KM 'splits' the previous N in half, thereby forming a binary tree. As the binary tree grows, the number of polynomials increases while their respective N-size decreases until each branch of the tree is reduced to an $N = 1$ as shown in Step 2 of Algorithm 5.2. Depending on the initial N-size, the 'splitting' involved in KM quickly composes a large binary tree requiring significant memory.

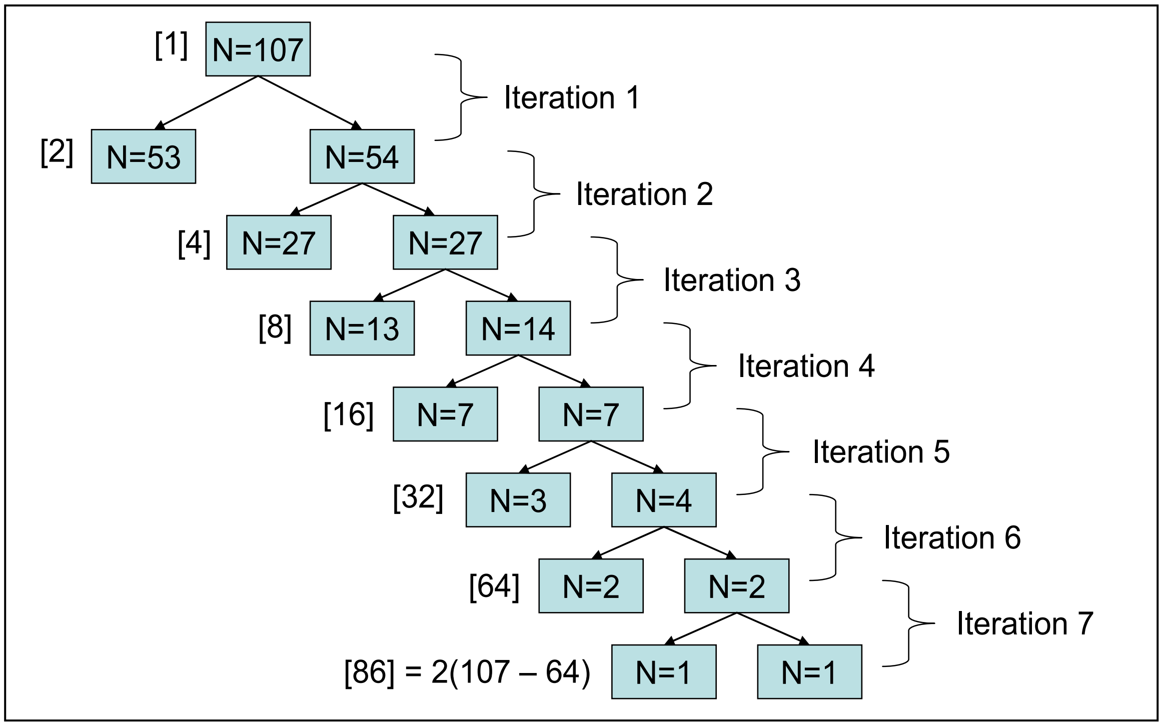


Figure 5.2: KM polynomial Count [] and Size (N) for each iteration for N=107

Benchmarking of KM relative to traditional multiplication was performed on a MICAz mote [25] with nesC [28] implementations of both. The test was composed of performing convolution, within ring R, two polynomials, *f(x)* and *g(x)*, with coefficients equal to '+1' and a *N*-size equal to 107. The initial tests using polynomials *f(x)* and *g(x)* showed KM to have poorer performance in terms of longer execution times than traditional multiplication. Driessen et al also noticed this performance decrease and attributed the drop in performance to the overhead of the recursion having to build another function stack [15]. To understand the affects of stack maintenance on performance for KM, the value of '*CutOff*' , as shown in Step 4 of Algorithm 5.2, was modified. The 'Cutoff' variable determines whether to continue the recursive calls to KM or to switch over to using the star multiply Algorithm 5.1. A 'Cutoff' value larger than the initial input polynomials N-size will disable KM and always execute the star multiplication, Algorithm 5.1. Selecting a 'Cutoff' equal to '1' will have the opposite effect, thereby always calling the KM, Algorithm 5.2, and never executing the Algorithm 5.1. Selecting an intermediate 'CutOff' value will utilize both Algorithms 5.1 and 5.2.

To demonstrate the effect the 'CutOff' value has on execution time, Table 5.1 summarizes the execution time required to perform the convolution of *f(x)* and *g(x)* within ring R for $N = 107$.

CutOff Value	Number of Iterations	Execution Time (ms)
108	**0**	**622**
55	1	543
28	2	452
15	3	398
8	**4**	**385**
5	5	441
3	6	576
1	**7**	**727**

Table 5.1: KM execution time versus CutOff value

As previously mentioned, the first entry executes star multiplication, Algorithm 5.1, since the 'CutOff' value is larger than the polynomial *N*-size of 107. Utilizing Algorithm 5.1 exclusively resulted in a 622ms execution time. Decreasing the 'CutOff' value below the input polynomial *N*-size starts to enable the recursive iterations of the KM. Selecting a 'CutOff' value equal to 1 disables all Algorithm 5.1 executions and relies on complete KM usage. Since recursion is being executed for every N-size polynomial in KM, the overhead of stack maintenance results in the worst performance time of 727ms. With a careful selection of a 'CutOff' value, equal to 8, between the two aforementioned extremes, an overall improved execution time of 385ms was obtained using only 4 out of a possible 7 iterations of KM. Any further decrease in the 'CutOff' value increases execution time due to the excessive stack overhead involved in recursion.

5.6 Private Key Polynomial f_p

During the key generation phase in Section 4.3, polynomial f was randomly generated until the following properties of the polynomial were met [45]:

- f is invertible mod p

- f is invertible mod q

While the random generation of f is fairly quick, performing the AIA (Algorithm 4.3) is very time consuming, especially if an inverse for f is not found and the AIA needs to execute again. An optimization used in the commercial release of NTRUEncrypt that eliminates the need to find the inverse polynomial f_p utilizes the following property of f [45]:

$$f = 1 + p * F \tag{22}$$

Since the AIA success criteria for a polynomial to be invertible is $f(0) = \pm 1$ (mod p), as shown in step 6b, one option is to generate a random polynomial F, star multiply by p and add 1. The star multiplication of F and p ensures a modulo reduction of p exists along with satisfying the $f(0) = \pm 1$ (mod p) criteria of the AIA. Generating a known invertible polynomial f benefits the NTRU algorithm in two regards:

- Polynomial f is guaranteed to be invertible mod p and therefore eliminates the AIA check for inverse.
- The second polynomial star multiplication of f_p and a in the decryption formula (20) is no longer required, thereby reducing the decryption execution time.

Algorithm 5.3 was used to generate a guaranteed invertible f_p

Algorithm 5.3: CreateInvertible(f(x), g(x), n, m)

```
Input:   $f(x)$ – Polynomial to invert
         $n$ – Size of polynomials a(x) and b(x)
         $m$ – Modulo value to apply to coefficients

Output:  $g(x)$ – Invertible polynomial

1: $g(x) = f(x)$
2: for ( $i = 0$ to $n$ )
   {
3:     $g(x) = g(x) * m$
   }
4: $g(0) = \text{Mod}( g(0) + 1, m )$
```

6 Evaluation and Performance

The intent of this thesis was to perform a like-for-like comparison of NTRUEncrypt against ECIES. To ensure the comparison would be as fair as possible, a version of each cryptosystem was chosen that had equivalent symmetric bit security levels. The work by Kouzmenko concluded that NTRUEncrypt-251 and ECC-163 both have an equivalent asymmetric security level of 80-bits [35]. Since the NTRUEncrypt-251 was based on a prime field, the prime field based ECIES-160 was chosen over the equivalent security level, non-prime field based, ECIES-163 [38].

6.1 Hardware

The MICAz mote, from Crossbow Technologies [25] in Figure 6.1, was used in the characterization and benchmarking of ECIES-160 versus NTRUEncrypt-251. Programming and PC communication with the mote utilizes the MIB520 USB Gateway in Figure 6.2.

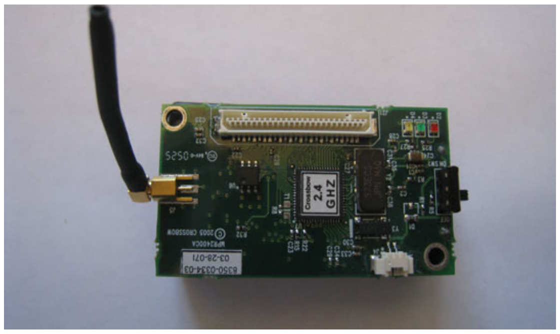


Figure 6.1:Crossbow MICAz

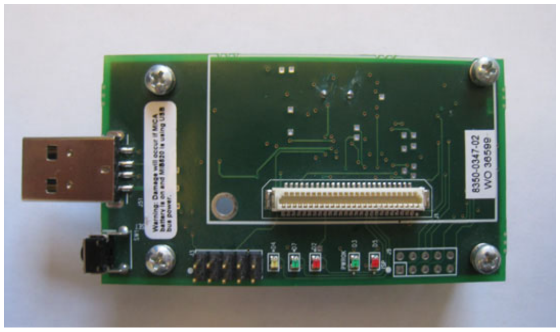


Figure 6.2: Crossbow MIB520 Gateway

The MICAz is an Atmel based 8-bit ATMega128L microcontroller [36] that contains a 2.4GHz IEEE 802.15.4 compliant RF transceiver and 4KB of RAM. A listing of the main specifications of the MICAz mote is summarized in Table 6.1.

Processor Performance	**MPR2400CA (ATmega128L)**
Clock	0 – 8 MHz
Program Flash Memory	128K bytes
Measurement (Serial) Flash	512K bytes
Configuration EEPROM	4K bytes
Current Draw	8mA – Active Mode
	< 15 µA – Sleep Mode
RF Transceiver	
Transmit Data Rate	250kbs
Frequency Band	2400 MHz to 2483.5 MHz
Outdoor Range	75m – 100m
Indoor Range	10m – 30m
Current Draw	19.7mA – Receive Mode
	17.4mA – Transmit, 0dBm
Electromechanical	
Battery	2X AA Batteries
External Power	2.7V – 3.3V

Table 6.1: MICAz (MPR2400CA) Hardware Specification

The MICAz mote contains expansions connectors that allow auxiliary inputs to be acquired such as light, temperature and acceleration, though for this thesis the base MICAz mote is the only required hardware.

6.2 Software

The MICAz motes run on TinyOS, which is open source operating system specifically designed for embedded sensor network devices [27]. It features various component libraries that include sensor drivers along with network protocols. Since TinyOS is open source, it is very easy to extend the base functionality of the pre-distributed drivers to suit the unique needs of the developer.

The MoteWorks version 2.0.F [51] of TinyOS 1.x was used exclusively through this thesis as the primary operating system for the MICAz. Though any version of

TinyOS 1.x would have sufficed, the MoteWork's released included the custom nesC header file, SOdebug.h, which provided a very nice interface to output code debug information to the PC using the MICAz's USB port. Without this powerful interface, debugging of the code in TinyOS 1.x would have been significantly more difficult.

The now deprecated TinyOS 1.x version was used instead of the latest TinyOS 2.1 version because the only operating system TinyECC supports is TinyOS 1.x. The use of TinyOS 2.1 would have eliminated the need for the MoteWorks version since TinyOS 2.1 includes improved interfaces for debugging code via the USB serial port.

6.3 Test Setup

A standard PC with an USB port was used to program and collect data from the MICAz mote. TinyOS version 1.1, supplied within the MoteWorks release, was installed on the PC and used in the nesC code development of NTRUEncrypt and collection of timing data for ECIES.

6.4 ECIES-160 Results

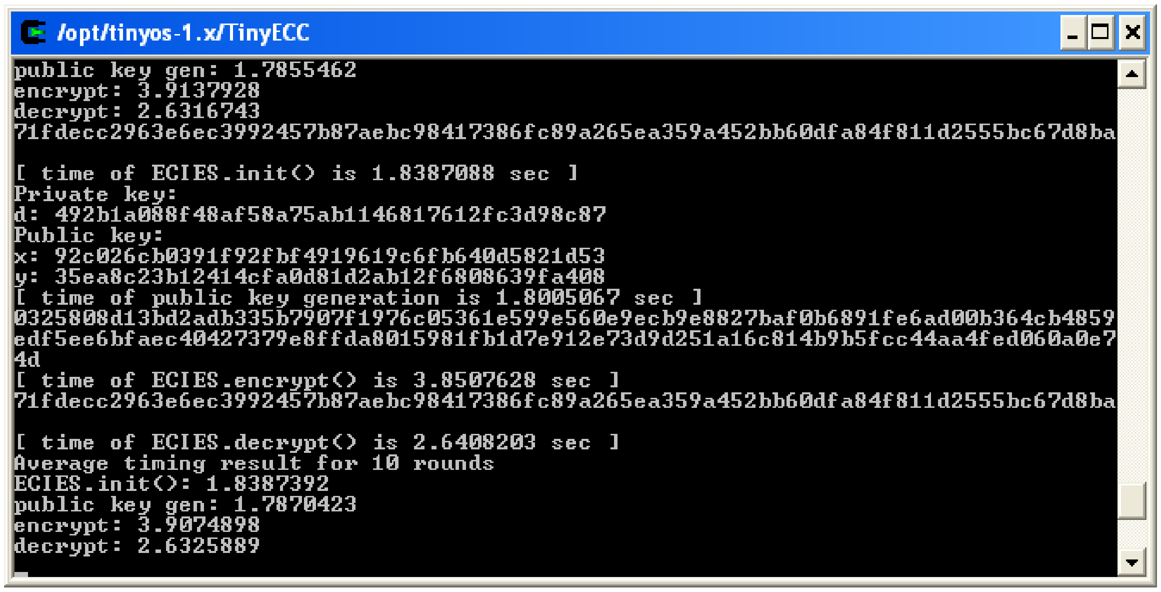
TinyECC [26] is a publically available software package that operates on TinyOS [27] and provides an ECC based public key cryptosystem for WSNs. It contains full implementations of the following ECC schemes:

- ECDH – Elliptic Curve Diffie-Hellman Key Agreement
- ECDSA – Elliptic Curve Digital Signature Algorithm
- ECIES – Elliptic Curve Integrated Encryption Scheme

ECIES was chosen as the ECC version to compare NTRUEncrypt throughout this thesis for its basic properties of encryption and decryption. ECIES differs from ECDSA in that ECDSA only provides signature generation and verification without an encryption of the message and would be better suited to compare against NTRUSign [36].

There are a number of optimization switches for ECIES that can be enabled at compilation time that will reduce execution times for initialization, encryption and decryption. To provide a fair comparison of execution times for the various operations between ECIES and NTRUEncrypt, all optimization switches were enabled which provided “the most computationally efficient configuration” [17]. A full listing of the switches along with details of each setting can be found in the TinyECC paper [17].

The TinyECC implementation of ECIES-160 by Liu et al [26], using the recommended elliptic curve domain parameters specified in secp160r1 [10], was compiled and install on a MICAz mote in an effort to recreate and compare execution times. With all optimizations enabled and the results average over 10 iterations, Figure 6.3 shows the actual results obtained after compiling and executing the ‘testECIESM.nc’ file released under TinyECC.



```
public key gen: 1.7855462
encrypt: 3.9137928
decrypt: 2.6316743
71fdecc2963e6ec3992457b87aebc98417386fc89a265ea359a452bb60dfa84f811d2555bc67d8ba

[ time of ECIES.init() is 1.8387088 sec ]
Private key:
d: 492b1a088f48af58a75ab1146817612fc3d98c87
Public key:
x: 92c026cb0391f92fbf4919619c6fb640d5821d53
y: 35ea8c23b12414cfa0d81d2ab12f6808639fa408
[ time of public key generation is 1.8005067 sec ]
0325808d13bd2adb335b7907f1976c05361e599e560e9ecb9e8827baf0b6891fe6ad00b364cb4859
edf5ee6bfaec40427379e8ffda8015981fb1d7e912e73d9d251a16c814b9b5fcc44aa4fed060a0e7
4d
[ time of ECIES.encrypt() is 3.8507628 sec ]
71fdecc2963e6ec3992457b87aebc98417386fc89a265ea359a452bb60dfa84f811d2555bc67d8ba

[ time of ECIES.decrypt() is 2.6408203 sec ]
Average timing result for 10 rounds
ECIES.init(): 1.8387392
public key gen: 1.7870423
encrypt: 3.9074898
decrypt: 2.6325889
```

Figure 6.3: Actual ECIES-160 Execution Times

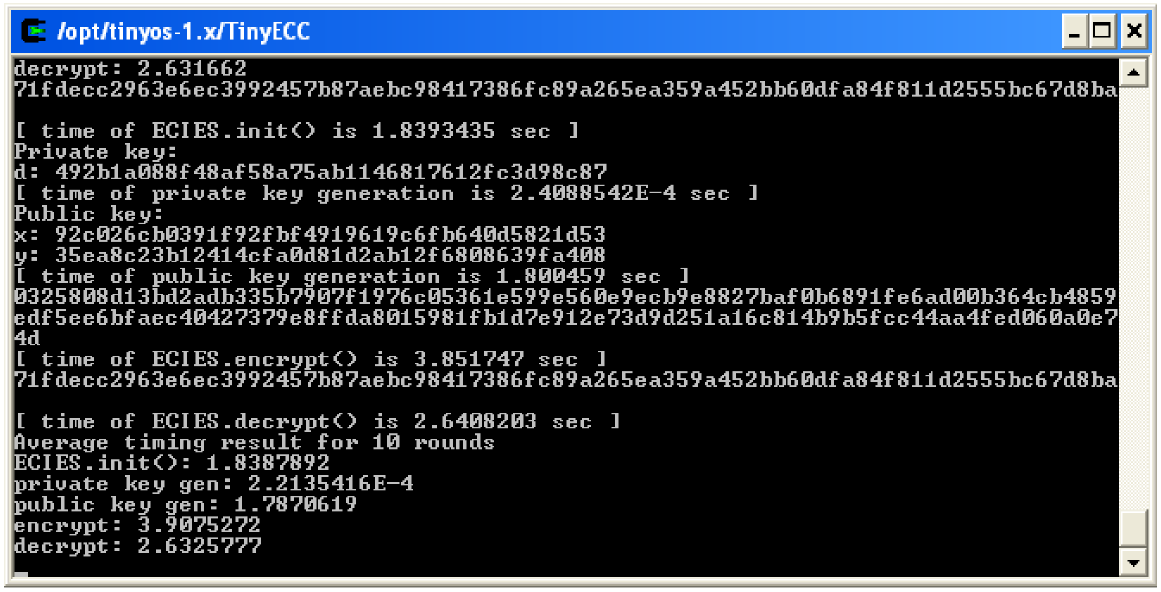
Table 6.2 summarizing the results obtained in this thesis along with those observed by Liu et al [17].

ECIES Operation	Liu et al (ms)	This Work (ms)
Initialization	1834.74	1838.74
Encryption	3907.34	3907.49
Decryption	2632.66	2632.59

Table 6.2 Liu et al vs. my work timing results for ECIES-160

The Initialization time for ECIES includes the time required to setup the Barrett Reduction and Sliding Window optimizations used in ECIES. Barrett Reduction is an alternative method for performing modulo reduction by using pre-computed modulo reciprocals along with multiplication operations instead of division [52]. The Sliding Window optimization speeds up scalar multiplications by storing pre-computed product combinations with a window width of *w* bits [17]. By performing the multiplication and storing the product ahead of time for a window width *w*, only one addition operation every *w* bits is needed for a scalar multiplication.

The output in Figure 6.3 shows initialization, public key generation, encryption and decryption times, but does not show the execution time for the private key. To understand the amount of time required to generate the private key in ECIES-160, the TinyECC code for the ECIESM.nc and show_ecies.java classes were modified to output the private key generation time as shown in Figure 6.4.



```
decrypt: 2.631662
71fdecc2963e6ec3992457b87aebc98417386fc89a265ea359a452bb60dfa84f811d2555bc67d8ba

[ time of ECIES.init() is 1.8393435 sec ]
Private key:
d: 492b1a088f48af58a75ab1146817612fc3d98c87
[ time of private key generation is 2.4088542E-4 sec ]
Public key:
x: 92c026cb0391f92fbf4919619c6fb640d5821d53
y: 35ea8c23b12414cfa0d81d2ab12f6808639fa408
[ time of public key generation is 1.800459 sec ]
0325808d13bd2adb335b7907f1976c05361e599e560e9ecb9e8827baf0b6891fe6ad00b364cb4859
edf5ee6bfaec40427379e8ffda8015981fb1d7e912e73d9d251a16c814b9b5fcc44aa4fed060a0e7
4d
[ time of ECIES.encrypt() is 3.851747 sec ]
71fdecc2963e6ec3992457b87aebc98417386fc89a265ea359a452bb60dfa84f811d2555bc67d8ba

[ time of ECIES.decrypt() is 2.6408203 sec ]
Average timing result for 10 rounds
ECIES.init(): 1.8387892
private key gen: 2.2135416E-4
public key gen: 1.7870619
encrypt: 3.9075272
decrypt: 2.6325777
```

Figure 6.4: Actual ECIES-160 Execution Times Including Private Key Generation

The results indicate a negligible amount of time is required to compute the private key in ECIES, i.e. 0.2ms, as shown as 'private key gen' in Figure 6.4 The time is small since it involves only selecting a random integer within the order of the base point for the elliptic curve as detailed in section 3.1.1.

The recreation of Liu et al's results provides confidence that the timing results obtained from this thesis's NTRUEncrypt implementation will be comparable to that of the ECIES-160 TinyECC version.

6.5 NTRUEncrypt-251 Implementation and Results

NTRUEncrypt's security is based on the values selected for parameters N, p and q. The initial suggested values from both NTRU.com [31] and CEES [49] are shown in Table 6.3.

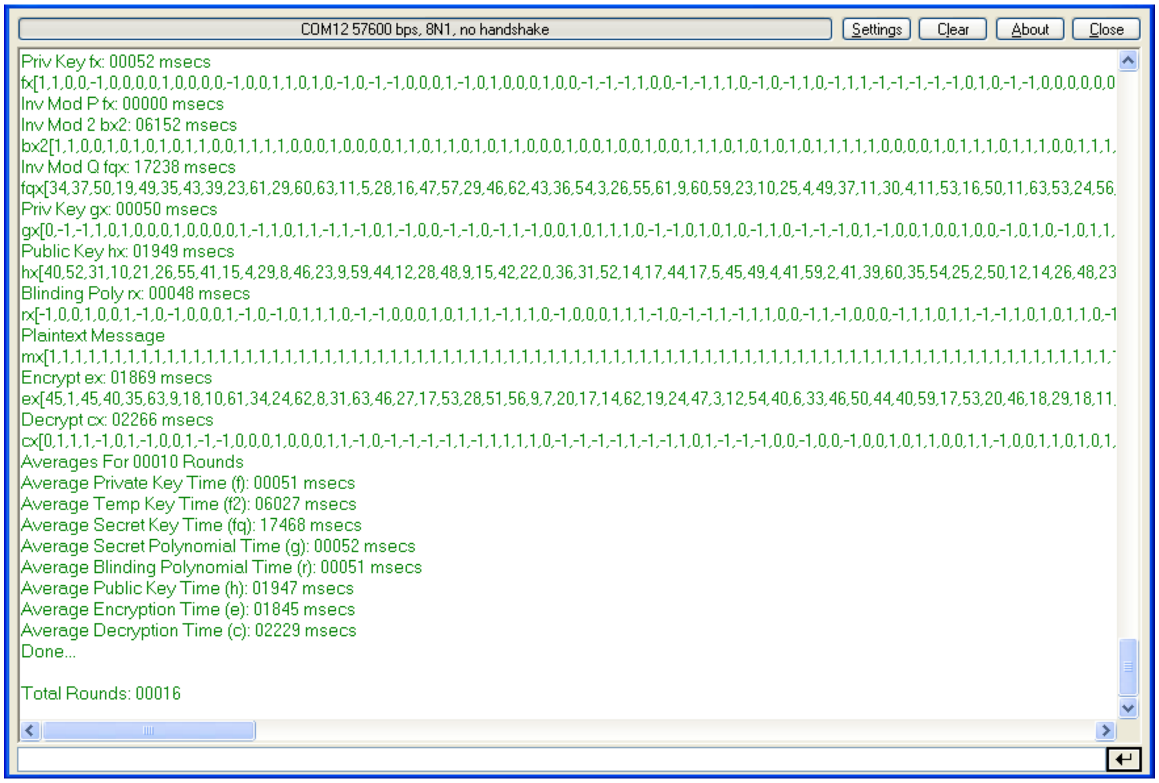
Security Level	N	p	Q	df	dg	dr
Low	107	3	64	15	12	5
Moderate	167	3	128	61	20	18
Standard	251	3	128	72	72	72

Table 6.3: NTRUEncrypt parameter selection based on security

As years have passed, recent attacks such as meet-in-the-middle (MITM) and lattice reduction [21] against NTRUEncrypt have revealed short falls in the original parameter sets in Table 6.3. As a result, newly suggested parameters recently, release in 2009, indicate the selection of yet larger values of N, p and q. For comparison purposes, Hirschhorn et al now suggests for 112-bit security with a compromise between speed and code space, that $N = 541$, $p = 3$ and $q = 2048$ [21]. These values will continue to be assumed stable until the next successful attack proves them ineffective.

The complete implementation of NTRUEncrypt-251 was written in nesC utilizing all of the pseudo-code algorithms mentioned in the previous sections. When applicable, the code utilized 1-byte, or 8-bit, sized registers to minimize memory usage. This constraint limits the register to a signed size ranging from -127 to +127. Therefore, the NTRUEncrypt-251 requirement of modulo $q = 128$ reduction of coefficients was not possible. In order to store polynomial coefficients that can be reduced by $q = 128$, the storage size for coefficients in the microprocessor code were increased to 2-bytes, or 16-bits. Since the MICAz mote contains 4kB of 8-bit data registers, increasing the coefficient storage size to 16-bits doubled the required amount of RAM. Even with techniques such as using globally defined arrays and maximizing the reuse of these arrays, the MICAz mote failed to provide enough RAM to implement the complete NTRUEncrypt-251 cryptosystem. The only way to successfully execute NTRUEncrypt-251 on the MICAz mote was to reduce the value of q to 64, and used polynomial arrays with 8-bit coefficients.

Even with the reduced coefficient widths, the collection of timing data from NTRUEncrypt-251 implementation proved to be challenge. The first attempt executed the complete cryptosystem with the Karatsuba and $f = 1 + p \otimes F$ optimizations disabled. This resulted in the MICAz continuously crashing due to lack of memory. The only option remaining was to enable the $f = 1 + p \otimes F$ substitution, since enabling Karatsuba would definitely require more memory. The $f = 1 + p \otimes F$ substitution eliminated the need to store the inverse f_p array, thereby reducing the required RAM by 251 bytes. This small reduction in memory was just enough to capture timing for a 10 round average of the complete cryptosystem as shown in Figure 6.5 and summarized in Table 6.4.



```
COM12 57600 bps, 8N1, no handshake    Settings  Clear  About  Close
Priv Key fx: 00052 msecs
fx[1,1,0,0,-1,0,0,0,0,1,0,0,0,0,-1,0,0,1,1,0,1,0,-1,0,-1,-1,0,0,0,1,-1,0,1,0,0,0,1,0,0,-1,-1,-1,1,0,0,-1,-1,1,1,0,-1,0,-1,1,0,-1,1,1,-1,-1,-1,-1,-1,0,1,0,-1,-1,0,0,0,0,0,0
Inv Mod P fx: 00000 msecs
Inv Mod 2 bx2: 06152 msecs
bx2[1,1,0,0,1,0,1,0,1,0,1,1,0,0,1,1,1,1,0,0,0,1,0,0,0,0,1,1,0,1,1,0,1,0,1,1,0,0,0,1,0,0,1,0,0,1,0,0,1,1,1,0,1,0,1,0,1,0,1,1,1,1,1,0,0,0,0,1,0,1,1,1,0,1,1,1,0,0,1,1,1,
Inv Mod Q fqx: 17238 msecs
fqx[34,37,50,19,49,35,43,39,23,61,29,60,63,11,5,28,16,47,57,29,46,62,43,36,54,3,26,55,61,9,60,59,23,10,25,4,49,37,11,30,4,11,53,16,50,11,63,53,24,56,
Priv Key gx: 00050 msecs
gx[0,-1,-1,1,0,1,0,0,0,1,0,0,0,0,1,-1,1,0,1,1,-1,1,-1,0,1,-1,0,0,-1,-1,0,-1,1,-1,0,0,1,0,1,1,1,0,-1,-1,0,1,0,1,0,-1,1,0,-1,-1,-1,0,1,-1,0,0,1,0,0,1,0,0,-1,0,1,0,-1,0,1,1,
Public Key hx: 01949 msecs
hx[40,52,31,10,21,26,55,41,15,4,29,8,46,23,9,59,44,12,28,48,9,15,42,22,0,36,31,52,14,17,44,17,5,45,49,4,41,59,2,41,39,60,35,54,25,2,50,12,14,26,48,23
Blinding Poly rx: 00048 msecs
rx[-1,0,0,1,0,0,1,-1,0,-1,0,0,0,1,-1,0,-1,0,1,1,1,0,-1,-1,0,0,0,1,0,1,1,1,-1,1,1,0,-1,0,0,0,1,1,1,-1,0,-1,-1,1,-1,1,1,0,0,-1,1,-1,0,0,0,-1,1,1,0,1,1,-1,-1,1,0,1,0,1,1,0,-1
Plaintext Message
mx[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
Encrypt ex: 01869 msecs
ex[45,1,45,40,35,63,9,18,10,61,34,24,62,8,31,63,46,27,17,53,28,51,56,9,7,20,17,14,62,19,24,47,3,12,54,40,6,33,46,50,44,40,59,17,53,20,46,18,29,18,11,
Decrypt cx: 02266 msecs
cx[0,1,1,1,-1,0,1,-1,0,0,1,-1,-1,0,0,0,1,0,0,0,1,1,-1,0,-1,-1,-1,-1,1,-1,1,1,1,1,0,-1,-1,-1,-1,1,-1,-1,1,0,1,-1,-1,-1,0,0,-1,0,0,-1,0,0,1,0,1,1,0,0,1,1,-1,0,0,1,1,0,1,0,1,
Averages For 00010 Rounds
Average Private Key Time (f): 00051 msecs
Average Temp Key Time (f2): 06027 msecs
Average Secret Key Time (fq): 17468 msecs
Average Secret Polynomial Time (g): 00052 msecs
Average Blinding Polynomial Time (r): 00051 msecs
Average Public Key Time (h): 01947 msecs
Average Encryption Time (e): 01845 msecs
Average Decryption Time (c): 02229 msecs
Done...

Total Rounds: 00016
```

Figure 6.5: NTRUEncrypt-251 Average Exec Time for 10 Rounds with CutOff = 252

Due to the selection of q = 64, instead of q = 128, decryption failures occurred as indicated by the fact decrypted message c does not equal the original plaintext message mx in Figure 6.5. Figure 6.5 also shows that it took 16 total attempts of finding polynomial inverses for f_q to obtain 10 successful inverses to complete an end-to-end execution of the NTRUEncrypt cryptosystem.

NTRUEncrypt-251 Operation	Variable Name	Execution Time (ms)
Encryption	e	1845
Decryption	c	**2229**
Private Key Generation	f	51
Temp Key Generation	f_2	6027
Secret Key Generation	f_q	17468
Encryption Total		**3895**
Secret Polynomial Generation	g	52
Blinding Polynomial Generation	r	51
Public Key Generation	h	1947

Table 6.4: NTRUEncrypt-251 Execution Times

The three highlighted rows were created for comparison to the categories of the ECIES-160 output in the previous section 6.4. The execution times for the categories of Initialization and Encryption Total is the sum total of the execution times of the indented rows below each category. Since the total encryption time for ECIES-160, shown in Table 6.2, includes the public key generation time, the category labeled 'Encryption Total' in Table 6.4 displays the total time for all steps of the encryption for NTRUEncrypt-251.

Attempts to improve the above executions times by enabling just one round of Karatsuba caused the MICAz to continuously reset due to insufficient RAM.

6.6 NTRUEncyrpt-251 vs. ECIES-160

Figure 6.6 compares the execution times for all stages of NTRUEncrypt-251 versus ECIES-160.

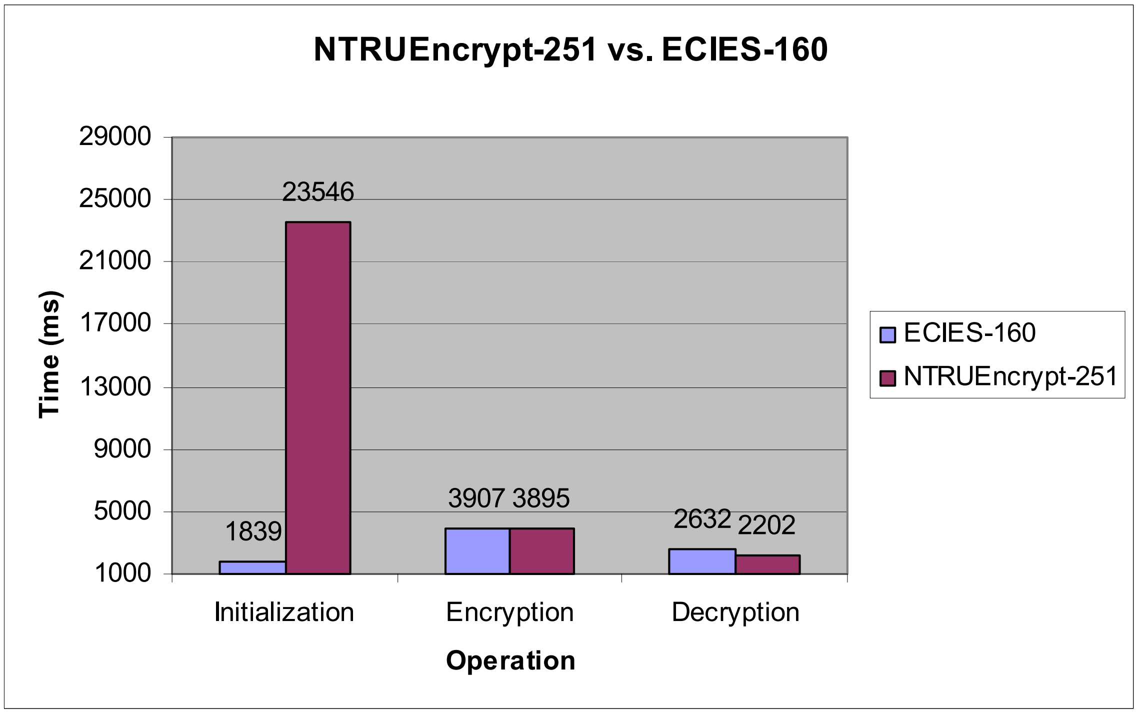
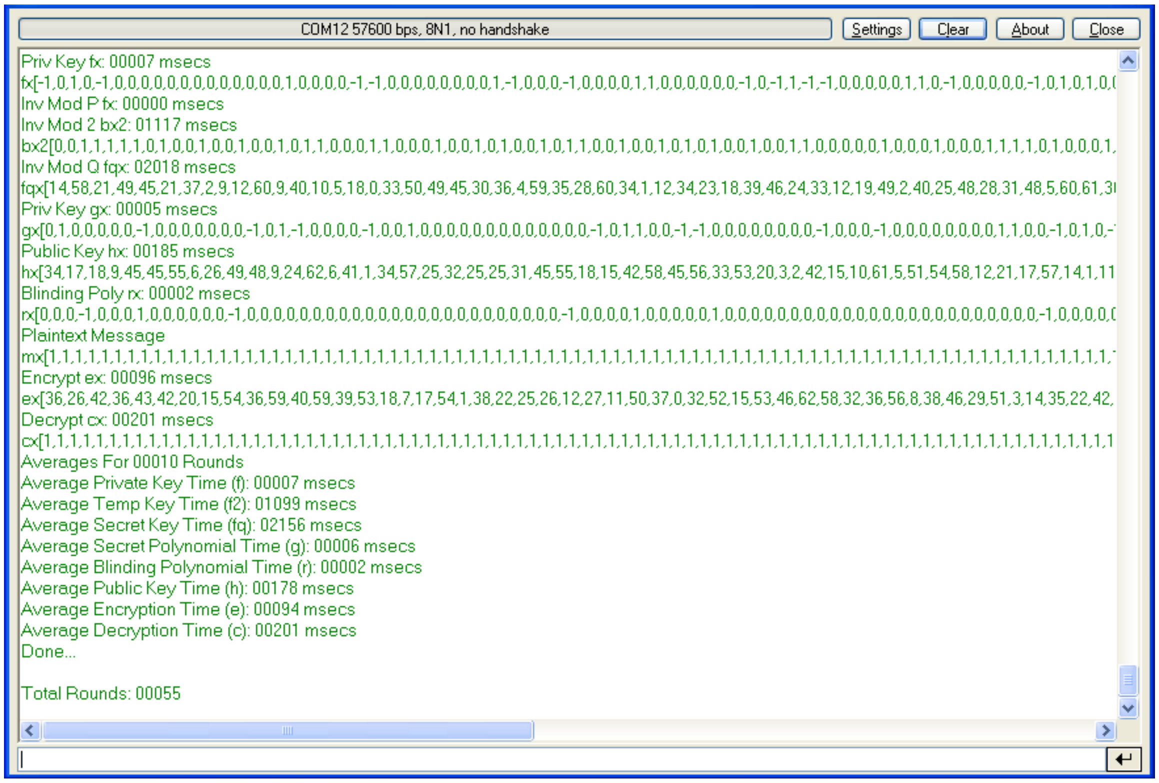


Figure 6.6: NTRUEncrypt-251 and ECIES-160 Execution Times

The Initialization times between the two cryptosystems revealed NTRUEncrypt-251 took approximately 13 times longer to initialize relative to ECIES-160, though the Encryption and Decryption times of the two cryptosystem were very comparable.

6.7 NTRUEncyrpt-107

Given the poor performance of NTRUEncrypt-251 on the MICAz mote, the 'low' security version of NTRUEncrypt, specified in Table 6.3, was evaluated to understand the effects Karatsuba would have on the overall execution time. Since NTRUEncrypt-107 requires less than half of the memory of NTRUEncrypt-251, execution times with all optimizations enabled were easily collected without any memory constraint issues. Figure 6.8 summarizes the various executions times at selected Karatsuba cutoff values with Figure 6.7 showing the details for the fastest execution time when CutOff = 28.



```
Priv Key fx: 00007 msecs
fx[-1,0,1,0,-1,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,-1,-1,0,0,0,0,0,0,0,0,1,-1,0,0,0,-1,0,0,0,0,1,1,0,0,0,0,0,0,-1,0,-1,1,-1,-1,0,0,0,0,0,1,1,0,-1,0,0,0,0,0,-1,0,1,0,1,0,
Inv Mod P fx: 00000 msecs
Inv Mod 2 bx2: 01117 msecs
bx2[0,0,1,1,1,1,1,0,1,0,0,1,0,0,1,0,0,1,0,1,1,0,0,0,1,1,0,0,0,1,0,0,1,0,1,0,0,1,0,1,1,0,0,1,0,0,1,0,1,0,1,0,0,1,0,0,1,1,0,0,0,0,0,1,0,0,0,1,0,0,0,1,1,1,1,0,1,0,0,0,1,
Inv Mod Q fqx: 02018 msecs
fqx[14,58,21,49,45,21,37,2,9,12,60,9,40,10,5,18,0,33,50,49,45,30,36,4,59,35,28,60,34,1,12,34,23,18,39,46,24,33,12,19,49,2,40,25,48,28,31,48,5,60,61,
Priv Key gx: 00005 msecs
gx[0,1,0,0,0,0,0,-1,0,0,0,0,0,0,0,-1,0,1,-1,0,0,0,0,-1,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,-1,0,1,1,0,0,-1,-1,0,0,0,0,0,0,0,0,-1,0,0,0,-1,0,0,0,0,0,0,0,0,1,1,0,0,-1,0,1,0,-
Public Key hx: 00185 msecs
hx[34,17,18,9,45,45,55,6,26,49,48,9,24,62,6,41,1,34,57,25,32,25,25,31,45,55,18,15,42,58,45,56,33,53,20,3,2,42,15,10,61,5,51,54,58,12,21,17,57,14,1,11
Blinding Poly rx: 00002 msecs
rx[0,0,0,-1,0,0,0,1,0,0,0,0,0,0,-1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,-1,0,0,0,0,1,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,-1,0,0,0,0,0,
Plaintext Message
mx[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
Encrypt ex: 00096 msecs
ex[36,26,42,36,43,42,20,15,54,36,59,40,59,39,53,18,7,17,54,1,38,22,25,26,12,27,11,50,37,0,32,52,15,53,46,62,58,32,36,56,8,38,46,29,51,3,14,35,22,42,
Decrypt cx: 00201 msecs
cx[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
Averages For 00010 Rounds
Average Private Key Time (f): 00007 msecs
Average Temp Key Time (f2): 01099 msecs
Average Secret Key Time (fq): 02156 msecs
Average Secret Polynomial Time (g): 00006 msecs
Average Blinding Polynomial Time (r): 00002 msecs
Average Public Key Time (h): 00178 msecs
Average Encryption Time (e): 00094 msecs
Average Decryption Time (c): 00201 msecs
Done...

Total Rounds: 00055
```

Figure 6.7: NTRUEncrypt-107 Average Exec Times for 10 Rounds with CutOff = 28

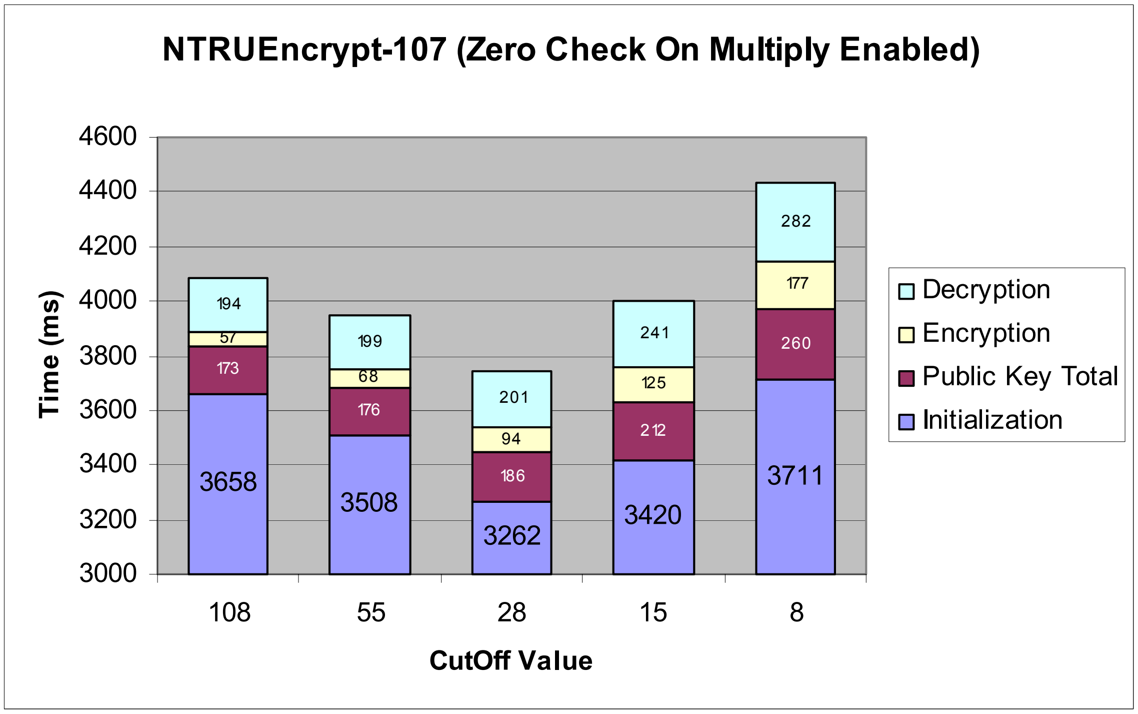


Figure 6.8: NTRUEncrypt-107 Execution Times with Zero Check Enabled.

The execution times for Encryption, Decryption and Public Key generation actually increase as more iterations of Karatsuba were introduced, while Initialization times decrease. The reason this occurs is due to less usage of the ‘zero check’ optimization within the Star Multiplication, Algorithm 5.1, as CutOff values are decreased. This very simple check for multiplication of zero coefficients is actually quicker than utilizing the Karatsuba method that does not include the check. Figure 6.9 shows the results for the execution times when the Zero Check was disabled for increasing levels of Karatsuba. The overall time for every cutoff value was worse than when the zero check enabled. The fastest execution time was 3921ms with Karatsuba (CutOff = 28) and the zero check in Star Multiply enabled compared to 5847ms with Karatsuba (CutOff = 8) and zero check disabled. This simple optimization of checking for zeros saved 1926ms overall.

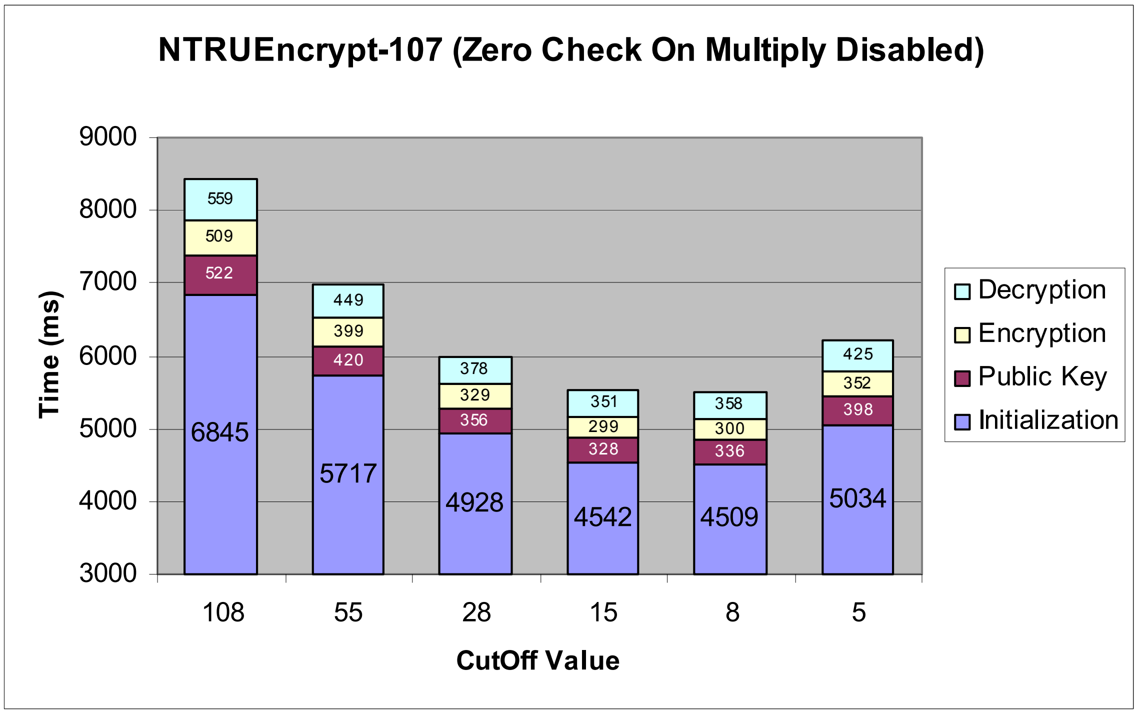


Figure 6.9: NTRUEncrypt-107 Execution Times with Zero Check Disabled.

Given additional memory on the MICAz, Karatsuba would have reduced the overall execution time for NTRUEncrypt-251 based on the execution time improvements Karatsuba provide NTRUEncrypt-107.

6.8 Energy Consumption

6.8.1 Microprocessor Power Consumption

The energy calculation equation presented in Liu et al [17] was used to calculate the power consumption used by the MICAz for each cryptosystem.

$$W = U \times I \times t \tag{21}$$

Where:

W – Power in millijoules (mJ)

U – Voltage in volts (v)

I – Current in milliamps (ma)

t – Time in milliseconds (ms)

Per the Crossbow data sheet for the MICAz mote, the current draw I = 8ma, at a voltage U = 3v [25]. An example calculation using the encryption time for ECIES-160 being 3907.49 ms, the amount of energy required to execute is:

$$W = U \times I \times t$$

$$W = 3v \times 8ma \times 3907.4ms$$

$$W = 93.8\ mJ$$

The amount of power required to execute NTRUEncrypt-251 versus ECIES-160 at various stages of the algorithms is shown in Table 6.5.

Operation	ECIES-160 Power Consumption (mJ)	NTRUEncrypt-251 Power Consumption (mJ)
Initialization	44.1	576.1
Encryption	93.8	93.5
Decryption	63.2	53.5

Table 6.5: Power Consumption for ECIES-160 and NTRUEncrypt-251

6.8.2 Transmit and Receive Power

Power calculations for the transmission and receipt of data, is based on the total number of data bits and the power required by the antenna to transmit a single bit. Power consumption of the ZigBee transceiver, CC2420, integrated into the MICAz mote consumes the following power to transmit and receive [9]:

Transmit Power @0dBm = 0.209 μJ/bit

Receive Power = 0.226 μJ/bit

The number of bits used in transmission and receiving, for each cryptosystem, are determined by their respective public key sizes in Table 6.6.

Cryptosystem	**Calculation**	**Public Key Size (bits)**
NTRUEncrypt-251	N*$\log_2(q)$ = 251* $\log_2(128)$	1757
ECIES-160	Num Public Keys * Size of Public Key (bits) = 2*160	320

Table 6.6: Public Key Size for NTRUEncrypt-251 and ECIES-160

Multiplication of the public key size by the power to transmit and receive one bit produces the transceiver power required to exchange a message on the MICAz mote as shown in Table 6.7.

Cryptosystem	**Operation Power (μJ)**	
	Transmit	**Receive**
NTRUEncrypt-251	1757*0.209 = 367.21	1757*0.226 = 397.08
ECIES-160	320*0.209 = 66.88	320*0.226 = 72.32

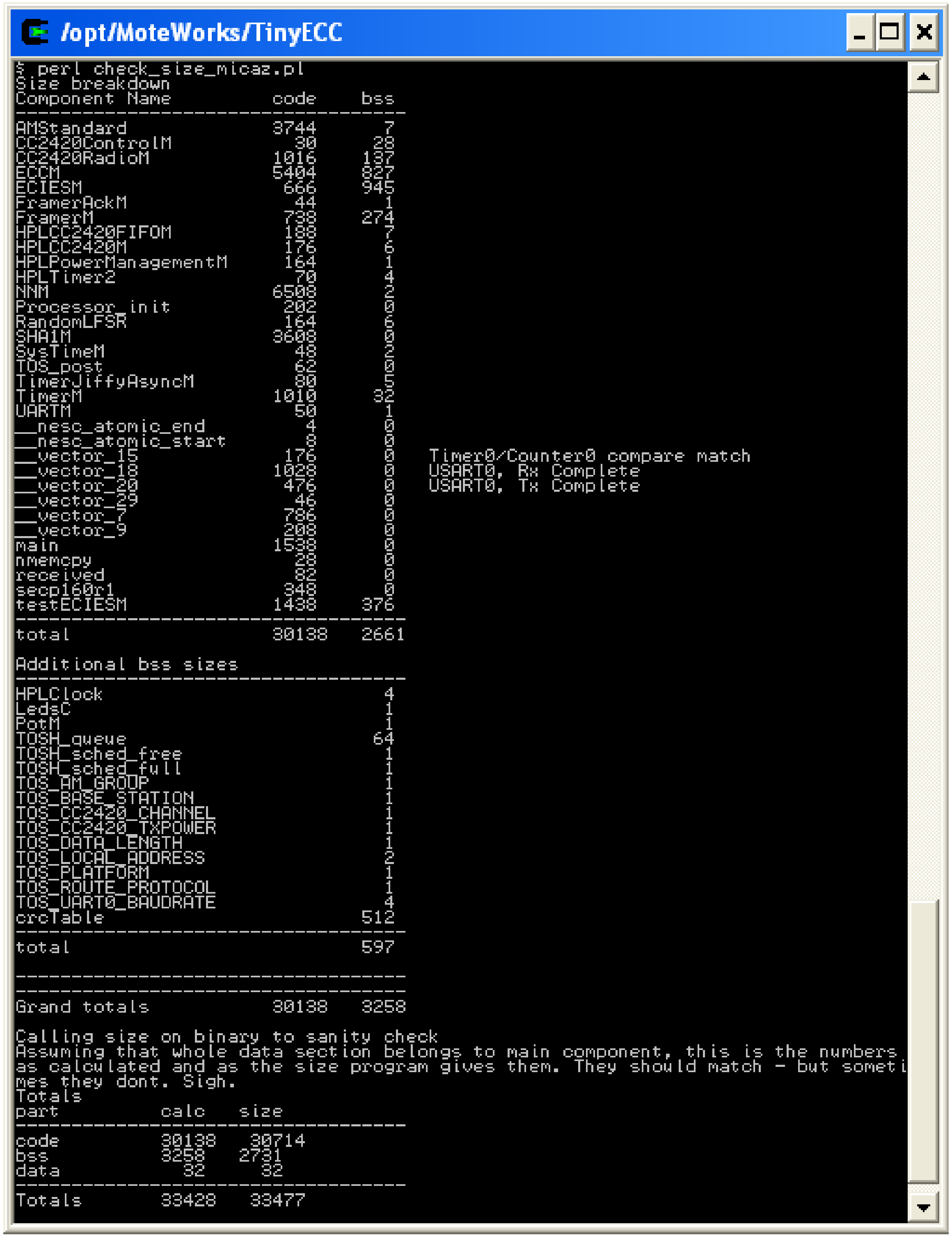
Table 6.7: Transmit and Receive Power Based on Cryptosystem

Since the power to transmit and receive is based on the number of bits in the public key, ECIES-160 requires less power than NTRUEncrypt-251.

Overall the execution time required by the cryptosystem to prepare the data to be transmitted requires a magnitude of 3 times more power than to transmit and receive the data. Optimization techniques that reduce the amount of execution time required by the processor will have the most impact on battery life.

6.9 Code Size

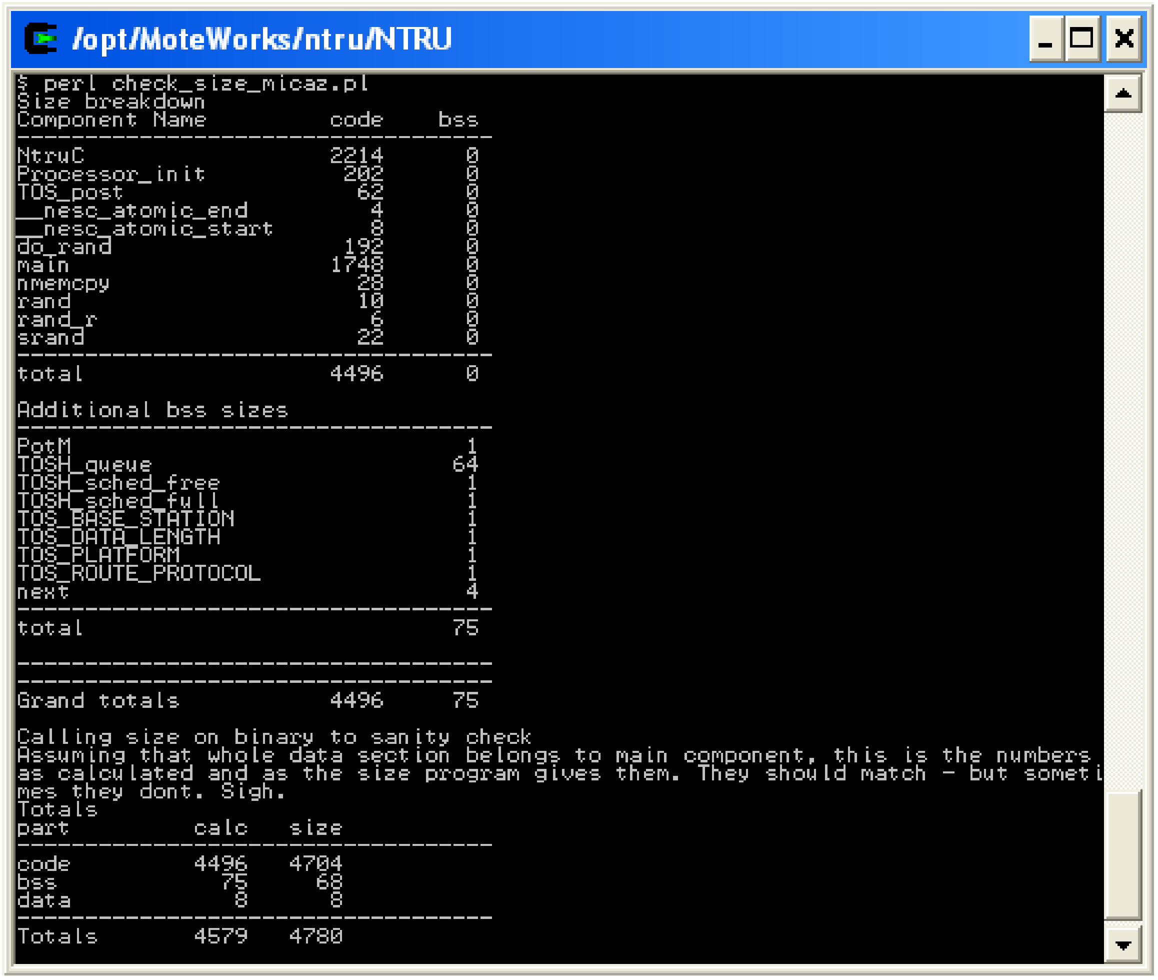
Calculation of the nesC code size for ECIES-160 and NTRUEncrypt-251 utilized the ‘check_size_micaz.pl’ PERL [4] script, under the TinyOS distribution, to obtain RAM and ROM sizes [17]. The script displays the code’s RAM and ROM size for each class referenced in the final binary executable file. Executing the script on the ECIES-160 and NTRUEncrypt-251 code obtained the outputs shown in Figure 6.10 and Figure 6.11 respectively.



```
$ perl check_size_micaz.pl
Size breakdown
Component Name         code    bss
-----------------------------------
AMStandard             3744      7
CC2420ControlM           30     28
CC2420RadioM           1016    137
ECCM                   5404    827
ECIESM                  666    945
FramerAckM               44      1
FramerM                 738    274
HPLCC2420FIFOM          188      7
HPLCC2420M              176      6
HPLPowerManagementM     164      1
HPLTimer2                70      4
NNM                    6508      2
Processor_init          202      0
RandomLFSR              164      6
SHA1M                  3608      0
SysTimeM                 48      2
TOS_post                 62      0
TimerJiffyAsyncM         80      5
TimerM                 1010     32
UARTM                    50      1
__nesc_atomic_end         4      0
__nesc_atomic_start       8      0
__vector_15             176      0   Timer0/Counter0 compare match
__vector_18            1028      0   USART0, Rx Complete
__vector_20             476      0   USART0, Tx Complete
__vector_29              46      0
__vector_7              786      0
__vector_9              208      0
main                   1538      0
nmemcpy                  28      0
received                 82      0
secp160r1               348      0
testECIESM             1438    376
-----------------------------------
total                  30138   2661

Additional bss sizes
-----------------------------------
HPLClock                         4
LedsC                            1
PotM                             1
TOSH_queue                      64
TOSH_sched_free                  1
TOSH_sched_full                  1
TOS_AM_GROUP                     1
TOS_BASE_STATION                 1
TOS_CC2420_CHANNEL               1
TOS_CC2420_TXPOWER               1
TOS_DATA_LENGTH                  1
TOS_LOCAL_ADDRESS                2
TOS_PLATFORM                     1
TOS_ROUTE_PROTOCOL               1
TOS_UART0_BAUDRATE               4
crcTable                       512
-----------------------------------
total                          597

-----------------------------------
-----------------------------------
Grand totals           30138   3258

Calling size on binary to sanity check
Assuming that whole data section belongs to main component, this is the numbers
as calculated and as the size program gives them. They should match - but someti
mes they dont. Sigh.
Totals
part         calc   size
-----------------------------------
code         30138   30714
bss          3258   2731
data           32     32
-----------------------------------
Totals       33428   33477
```

Figure 6.10: ECIES-160 Code Size

]



```
$ perl check_size_micaz.pl
Size breakdown
Component Name           code     bss
-----------------------------------
NtruC                    2214       0
Processor_init            202       0
TOS_post                   62       0
__nesc_atomic_end           4       0
__nesc_atomic_start         8       0
do_rand                   192       0
main                     1748       0
nmemcpy                    28       0
rand                       10       0
rand_r                      6       0
srand                      22       0
-----------------------------------
total                    4496       0

Additional bss sizes
-----------------------------------
PotM                                1
TOSH_queue                         64
TOSH_sched_free                     1
TOSH_sched_full                     1
TOS_BASE_STATION                    1
TOS_DATA_LENGTH                     1
TOS_PLATFORM                        1
TOS_ROUTE_PROTOCOL                  1
next                                4
-----------------------------------
total                              75

-----------------------------------
-----------------------------------
Grand totals             4496      75

Calling size on binary to sanity check
Assuming that whole data section belongs to main component, this is the numbers
as calculated and as the size program gives them. They should match - but someti
mes they dont. Sigh.
Totals
part           calc    size
-----------------------------------
code           4496    4704
bss              75      68
data              8       8
-----------------------------------
Totals         4579    4780
```

Figure 6.11: NTRUEncrypt-251 Code Size

The script isolates each individual class and displays that class's code size (ROM) and bss (RAM) sizes in two columns. The ROM calculation is the amount of program flash memory required to store the program code, including the initial values of global variables, within the MICAz's max limit of 128KB of ROM. The RAM calculation, also known as data, contains the size for constants and initialized static data. Combining the values of each row for each column produces the 'Totals' value in the last line of Figure 6.10 and Figure 6.11. To obtain code sizes for each cryptosystem, only the entries related to the actual cryptosystem code are combined where as the other entries for the MICAz's operating system are ignored. The NTRUEncrypt-251 was written in one nesC file, so the only entry needed from Figure 6.11 for the ROM and RAM sizes is the 'NtruC' entry. Code size for the ECIES-160 implementation uses many nesC classes, all of which need to be added together. Adding the ROM and RAM sizes together for ECCM, ECIESM, NNM, RandomLFSR, SHA1M, testECIESM and secp160r1 produced the ROM and RAM sizes for ECIES-160 in Table 6.8.

Cryptosystem	ROM (bytes)	RAM (bytes)
NTRUEncrypt-251	2214	0
ECIES-160	18136	2156

Table 6.8: NTRUEncrypt-251 and ECIES-160 ROM and RAM Sizes

The memory sizes published by Liu et al [17] were 19308 bytes for ROM and 1510 bytes for RAM. The memory size differences between Liu et al [17] and this thesis's work may be related to different classes being chosen in the total memory size calculations.

The RAM size of '0' in the NTRUEncrypt calculation indicates no constants or static data were used in the implementation. Overall, NTRUEncrypt-251 occupies less ROM and RAM than ECIES-160.

7 Conclusion and Future Work

The work presented in this thesis provided the design, implementation and evaluation of NTRUEncrypt on a resource constrained device. An overview of ECC provided a high level understanding of the PKCS. Performance characteristic including execution time, power consumption and code size for both NTRUEncrypt-251 and ECIES-160 were evaluated. The performance impacts of several optimizations to NTRUEncrypt were presented along with there tradeoffs.

Though NTRUEncrypt-251 was not able to outperform ECIES-160 due to system resource constraints, results suggest better performance of NTRUEncrypt-251 may be obtainable if more memory was available.

Opportunities exist that can significantly improve the performance of NTRUEncrypt on resource constrained devices. This thesis proved that Karatsuba can provide significant reductions in the overall execution time of the NTRUEncrypt, especially in the calculation of polynomial inverses. This thesis was unable to provide an optimized NTRUEncrypt-251 implement that could outperform ECIES-160 due to the 4kB memory constraint of the MICAz mote. Given several kB more memory, NTRUEncrypt-251 has the potential to outperform ECIES-160.

Future work of implementing NTRUEncrypt-251 on a Crossbow TelosB mote, which contains 10kB of RAM, should allow the enablement of optimizations that may bring NTRUEncrypt-251 performance equal to or better than ECIES-160.

The addition of a ‘Zero Check’ in the Karatsuba algorithm may enhance overall execution time of NTRUEncrypt similar to the results of the ‘Zero Check’ in the Star Multiply algorithm.

8 Bibliography

[1] W. Trappe and L. C. Washington, "Introduction to Cryptography with Coding Theory". Pearson Prentice Hall, 2006. pp. 385-390.

[2] J. Hoffstein, J. Pipher, and J. H. Silverman, "NTRU: A Ring-Based Public Key Cryptosystem" in Algorithmic Number Theory (ANTS III) June 1998, pp. 267-288 Vol. 1423 of Lecture Notes in Computer Science.

[3] R. L. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public Key Cryptosystems" in MIT Laboratory for Computer Science and Department of Mathematics, February 1978, pp. 120-126.

[4] PERL, "Practical Extraction and Report Language", http://www.perl.org/

[5] H. Wang and Q. Li "Efficient Implementation of Public Key Cryptosystems on Mote Sensors (Short Paper)" in Department of Computer Science College of William and Mary, November 2006, pp. 519-528.

[6] M. Luk, G. Mezzour and A. Perrig, "MiniSec: A Secure Sensor Network Communication Architecture" in CyLab, Carnegie Mellon University, April 2007, pp. 479-488.

[7] C. Karlof, N. Sastry and D. Wagner, "TinySec: A Link Layer Security Architecture for Wireless Sensor Networks" in UC Berkley, November 2004, pp. 162-175.

[8] Certicom, "An Elliptic Curve Cryptography (ECC) Primer", http://www.certicom.com/images/pdfs/WP-ECCprimer.pdf

[9] Kr. Piotrowski, P. Langendoefer and S. Peter, "How Public Key Cryptography Influences Wireless Sensor Node Lifetime" in IHP, October 2006, pp.169-176.

[10] Certicom Research, "SEC 2: Recommended Elliptic Curve Domain Parameters", September 20, 2000, Version 1.0. - http://www.secg.org/download/aid-386/sec2_final.pdf

[11] V. S. Miller, "Use of Elliptic Curves in Cryptography", Advances in Cryptology: Proceedings of Crypto'85, pp. 417-426, 1986.

[12] H. Wang, B. Sheng and Q. Li, "TelosB Implementation of Elliptic Curve Cryptography over Primary Field – WM-CS Technical Report", October 18, 2005.

[13] WM-ECC: an Elliptic Curve Cryptography Suite on Sensor Motes, Technical Report WMCS-2007-11, 2007.

[14] J. Buchmann, M. Döring and R. Linder, "Efficiency Improvements for NTRU", Proceedings Sicherheit 2008, GI LNI 128, 2008

[15] B. Driessen, A. Poschmann and C. Paar, "Comparison of Innovative Signature Algorithms for WSNs", pp. 30-35, March 2008

[16] N. Challa and J. Pradhan "Performance Analysis of Public Key Cryptographic Systems RSA and NTRU", IJCSNS International Journal of Computer Science and Network Security, Vol.7 No8., pp. 87-96, August 2007

[17] A. Liu and P. Ning, "TinyECC: A Configurable Library for Elliptic Curve Cryptography in Wireless Sensor Networks", 2008 International Conference of Information Processing in Sensor Networks, pp. 245-256, April 2008.

[18] W. Diffie, "The First Ten Years of Public-Key Cryptography", Proceedings of the IEEE Vol. 26, No. 5, pp. 560-577, May 1998.

[19] R. Roman, C. Alcaraz and J. Lopez, "A Survey of Cryptographic Primitives and Implementations for Hardware-Constrained Sensor Network Nodes", pp. 231-244, August 2007.

[20] J. H. Silverman, "Almost Inverses and Fast NTRU Key Generation", NTRU Cryptosystems Technical Report, Report #014, Version 1, March 15, 1999.

[21] P. Hirschhorn, J. Hoffstein, N. Howgrave-Graham and W. Whyte, "Choosing NTRU Parameters in Light of Combined Lattice Reduction and MITM Approaches", NTRU Articles – http://www.ntru.com/cryptolab/articles.htm#2009_1.

[22] "IEEE P1363.1™/D12 – Draft Standard Specification for Public-Key Cryptographic Techniques Based on Hard Problems over Lattices", Institute of Electrical and Electronics Engineers, Inc. 2008.

[23] K. Wilhelm, "Aspects of Hardware Methodologies for the NTRU Public-Key Cryptosystem", Rochester Institute of Technology, February 2008.

[24] J. P. Kaps, "Cryptography for Ultra-Low Power Devices", Worcester Polytechnic Institute, Degree of Doctor of Philosophy, May 2006.

[25] Crossbow Technology MICAz Mote Datasheet, http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MICAz_Datasheet.pdf.

[26] Cyber Defense Laboratory, "TinyECC: A Configurable Library for Elliptic Curve Cryptography in Wireless Sensor Networks (Version 1.0)", October 2, 2007 - http://discovery.csc.ncsu.edu/software/TinyECC/

[27] TinyOS, http://www.tinyos.net/

[28] nesC 1.1 Language Reference Manual, http://www.tinyos.net/tinyos-1.x/doc/nesc/ref.pdf

[29] T. T. Dai, Cao T. Hieu and C. S. Hong, "A Resource-Optimal Pre-Distribution Scheme for Secure Wireless Sensor Networks", Kyung Hee University, pp. 546-549, 2006.

[30] W. Du, R. Wang and P. Ning, "An Efficient Scheme for Authenticating Public Keys in Sensor Networks", pp. 58-67, May 2005.

[31] NTRU. "The NTRU Public Key Cryptosystem", http://www.ntru.com/cryptolab/tutorials.htm

[32] G.V.S Raju and R. Akbani, “Elliptic Curve Cryptosystems and its Application”, IEEE International Conference on Volume 2, Issue, 5-8, pp. 1540 – 1543, Oct 2003.

[33] D. R. Hankerson, S. A. Vanstone and A. J. Menezes, “Guide to Elliptic Curve Cryptography”, Springer, 2004, pp.1-18.

[34] L. H.A Correia, D. F. Macedo, A. L. dos Santos, A. A. F. Loureiro and J. Marcos S. Nogueira, “Transmission Power Control Techniques for Wireless Sensor Networks”, ScienceDirect , July 2007, pp.4765 - 4779.

[35] R. Kouzmenko, “Generalizations of the NTRU Cryptosystem”, Winter Semester 2005 – 2006, pp. 1-65.

[36] J. Hoffstein, N. Howgrave-Graham, J. Pipher, J. H. Silverman and W. Whyte, “NTRUSign: Digital Signature Using the NTRU Lattice”, CT-RSA 2003 proceedings, pp. 1-18. http://www.ntru.com/cryptolab/pdf/NTRUSign_RSA.pdf

[37] Atmel 8-bit Microcontroller, http://www.atmel.com/dyn/resources/prod_documents/doc2467.pdf

[38] A. K. Lenstra and E. R. Verheul, “Selecting Cryptographic Key Sizes”, October 27, 1999, pp. 1-32.

[39] A. Weimerskirch and C. Paar, “Generalizations of the Karatsuba Algorithm for Efficient Implementations”, Communication Security Group, Department of Electrical Engineering & Information Sciences, pp. 1-17.

[40] A. Karatsuba and Y. Ofman, “Multiplication of Multidigit Numbers by Automata”, Soviet Physics-Doklady, vol. 7, 1963, pp. 595-596.

[41] W. Diffie and M. E. Hellman, “New Directions in Cryptography”, IEEE Transactions on Information Theory, November 1976, 22(6):644-654.

[42] “IEEE P1363-2000: Standard Specifications For Public Key Cryptography”, Institute of Electrical and Electronics Engineers, Inc. 2000.

[43] J. H. Silverman, “High-Speed Multiplication of (Truncated) Polynomials”, Report #010, Version 1, January 5, 1999, pp. 1-4 http://www.ntru.com/cryptolab/pdf/NTRUTech010.pdf

[44] NTRU. “Algebra Tutorial”, http://www.ntru.com/cryptolab/tutorial_algebra.htm

[45] NTRU. “The NTRU Public Key Cryptosystem: Enhancements I”, http://www.ntru.com/cryptolab/tutorial_advanced.htm

[46] R. A. Mollin, “RSA and public-key cryptography”, CRC Press, 2003, Chapter 3 – Public-Key Cryptography

[47] R. D. Silverman, “Has the RSA algorithm been compromised as a result of Bernstein’s paper?”, RSA Laboratories, April 8, 2002, http://www.rsa.com/rsalabs/node.asp?id=2007

[48] N. Howgrave-Graham, P. Nguyen, D. Pointcheval, J. Proos, J. H. Silverman, A. Singer and W. Whyte, “The Impact of Decryption Failures on the Security of NTRU Encryption”, in Proc. Crypto 2003, Santa Barbara, USA, 2003 pp. 1-22.

[49] Consortium for Efficient Embedded Security, “Efficient Embedded Security Standards (EESS)”, June 20, 2003, http://grouper.ieee.org/groups/1363/lattPK/submissions/EESS1v2.pdf

[50] J. H. Silverman and W. Whyte, “Estimating Decryption Failure Probabilities for NTRUEncrypt”, NTRU Cryptosystems Technical Report, Report # 18, Version1, http://www.ntru.com/cryptolab/pdf/NTRUTech018.pdf

[51] MoteWorks 2.0.F, http://www.xbow.com/support/wSoftwareDownloads.aspx

[52] M. Knežević, F. Vercauteren and I. Varbauwhede, “Speeding Up Barrett and Montgomery Modular Multiplications”, COSIC internal report, pp. 1-11, 2009.