FACULTY
OF ENGINEERING

DEPARTMENT OF
ELECTRICAL ENGINEERING – ESAT

DIVISION OF
COMPUTER SECURITY AND INDUSTRIAL
CRYPTOGRAPHY

KATHOLIEKE
UNIVERSITEIT
LEUVEN

NTRU software implementation for constrained devices

End work nominated to obtaining the diploma Master of Science in Telecommunication Engineering & Management

Mariano Monteverde

Promotor:
Prof. Dr. Bart Preneel
Daily supervisors:
ir. Carmela Troncoso
ir. Benedikt Gierlichs
Assessors:
Dr. Lejla Batina
Dr. Frederik Vercauteren

2007 – 2008

Foreword

This document is the result of a master thesis carried out at the *Katholieke Universiteit Leuven* as final work for the MASTEAM conducted at the *Escola Politècnica Superior de Casteldefells*.

The thesis was prepared in the Division of Computer Security and Industrial Cryptography at the *Katholieke Universiteit Leuven*, Belgium, from February 2008 to July 2008. Additionally, this document has been developed during the mentioned period as well as in Madrid, from August 2008 to September 2008, in Hong Kong during October 2008, in Beijing during November 2008 and in Hong Kong again during December 2008 and May 2009.

Beyond any doubt, the conclusion of this thesis represents the end of a cycle that has been both diverse and rich in experiences. The world of science cannot be understood without a principle of curiosity that triggers an action or initiative which results in the search of an answer to the unknown. Thanks to this effort, society can move forward with a better understanding of what surrounds us, affording a better interaction with the environment and an increase on our limited abilities. It is said that vocation makes the personality but, in the best cases, it is personality that shapes the vocation. During this learning period I had the privilege to work with people who fall under this unique virtue and I would like to thank them for their work and diligence.

In particular, I would like to thank the following, without their help and support the development of this master's thesis would not have been possible:

Benedikt Gierlichs and Carmela Troncoso for their excellent support and guidance during the thesis, as well as their flexibility and relentless effort while being my supervisors. I would also like to acknowledge Frederik Vercauteren and Lejla Batina for their assistance and availability as counselors in this master's thesis. I want to specially express my gratefulness to Professor Dr. Bart Preneel for giving me the chance to develop this thesis and for his finesse, proficiency and consideration as Promotor. Finally, I would like to thank my family for providing me with the opportunity to receive an education and encouraging my free development, with a balance in handing me responsibilities while supporting me when it has been necessary.

Abstract

The NTRUEncrypt is a public-key cryptosystem based on the shortest vector problem. Its main characteristics are the low memory and computational requirements while providing a high security level.

This document presents an implementation and optimization of the NTRU public-key cryptosystem for constrained devices. Specifically the NTRU cryptosystem has been implemented on the ATMega128 and the ATMega163 microcontrollers.

This has turned in a major effort in order to reduce the consumption of memory and optimize the computational resources. The different resulting optimizations have been compared and evaluated throught the AVR Studio 4 [1]. The final outcome has also been compared with other published public-key cryptosystems as RSA or ECC showing the great performance NTRUEncrypt is able to deliver at a surprising very low cost.

Contents

List of figures

List of tables

Chapter 1

Introduction

Cryptography is intrinsically linked to data communications so that, in addition to authentication and authorization, integrity and confidentiality can be guaranteed. This has obvious applications for specific sectors, such as banking or military, but in fact we use cryptography in our daily mundane actions. It is used from making a mobile phone call to withdrawing cash from an ATM, watching a DVD movie or opening a car with the remote.

In the current globalized world, cryptography is increasingly necessary as it allows us to extrapolate many processes to the electronic world in a safe way, making management easier. This can be observed in the rise of new applications for integrated circuits (IC), as they get improved and their capabilities are upgraded. For example, the smart ID cards used for immigration applications, digital signatures and as library cards. Or the new electronic medical cards which store one's medical records and issued prescriptions.

These applications deal with very sensitive data and require a high security level. But the requirements to provide enough security can be very expensive in terms of hardware. Depending on how ambitious our application is, this need may increase the product costs notably, or may be simply too high to be implemented in an embedded device.

This is where NTRU Encrypt [2] plays a leading role since it is capable of providing adequate levels of security at an extremely low cost. The NTRU public key cryptosystem [2], PKC, features reasonably short keys, high speed, and low memory requirements. NTRU Encrypt, compared to other PKC, offers an excellent trade-off between the memory consumption and the operational complexity. RSA [3], for example, has bigger keys and the operations to encrypt and decrypt are more complex, requiring more memory and time to provide with a similar security level. ECC [4], on the other hand, has smaller keys but it is more complex computationally, which means spending more time. For more detailed information please refer to section 6.4.

This characteristic makes NTRU extremely suitable to be implemented on embedded devices.

1.1 Goals

In this master's thesis, we focus on the implementation and the optimization of NTRU public-key cryptosystem for the ATMega 163. The main hardware specifications of the microcontroller ATMega 163 are very restrictive, especially in terms of memory.

Achieving an efficient implementation of NTRU Encrypt in this device has been the main purpose of this master's thesis.

Over the implementation, some notes are given referring to data reduction. We also propose some optimizations relating the addition of binary polynomials and the modulus reduction using efficient logic operators.

Additionally, two algorithms for the star multiplication have been developed and tested. The first star multiplication is suitable for polynomials with ternary coefficients (which can be ported very easily to binary polynomials) providing a potential cost of $N \cdot 4 \cdot d$ additions and N *AND* operations during encryption, where N is the degree of the polynomial and d the parameter which defines the space for this polynomial. The second star multiplication is customized for a particular form of the private key f of the NTRU cryptosystem, featuring a theoretical cost of $2 \cdot N$ multiplications, $N \cdot (4 \cdot d)$ additions and N *AND* operations during decryption. Finally, the last implementation of the cryptosystem is compared to RSA, ECC and HECC [5].

1.2 Structure

This thesis is structured as follows,

- Chapter 2 reviews the mathematical background necessary to understand how NTRU Encrypt works.
- Chapter 3 describes how the cryptosystem works. In addition, some security directives regarding the NTRU PKC are analyzed.
- Chapter 4 describes the software implementation. First, we introduce the code specifications describing the functionality structure of the code and the required inputs and outputs. Finally, in the code reference section, the implemented function headers in the first NTRU PKC developed in this thesis are described.
- Chapter 5 presents the hardware specifications. This specifications are necessary in order to adequate the code to this specific platform, in this case the microcontroller ATmega 163.
- Chapter 6 explains the optimization proposed and implemented for an embedded device. Most relevant aspects have been reducing the computational complexity of the encryption and decryption processes while using as little amount of memory as possible to be able to fit the cryptosystem into the device. Afterwards, the impacts of the optimizations are evaluated. Finally, there is a comparison of our last version of the cryptosystem containing all optimizations with other PKC published results.
- Chapter 7 presents the IEEE standards P1363.1 [6] proposed and approved during the development of this thesis.
- Chapter 8 contains a summary and conclusion with suggestions for future research.

Chapter 2

Mathematical Background

2.1 Description

The NTRU public-key cryptosystem (PKC) was published in [2]. It is a system based on polynomial algebra, number theory and probability. This chapter is intended to give the necessary mathematical knowledge to understand the NTRU PKC operations described in chapter 3.

2.2 Modular Arithmetic

Modular (or clock) arithmetic is an arithmetic system for equivalence of integer numbers named residue classes. A common example to describe the modular arithmetic behavior is a clock. In a common analogue clock after 12 hours the hour hand will return to the same position where it was before after this period of time. Twelve is then the modulo (see section 2.2.1) in this scenario. We could describe the clock as a system limited to 12 integers (from 0 to 11) that increases to the next position or value until the maximum value is reached to cyclically start again from the initial or smallest integer. This way twelve becomes zero. See figure 2.1

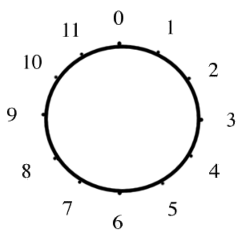

Figure 2.1: Clock

Analogously in a 24 hour clock, after 24 hours the value is again the initial one. In this case the modulo would be 24. We can observe that in the 24 hour clock the value 13 equals 1 in the

12 hour clock (14 equals 2 and so on until the value 24). These numbering equivalences are the so called congruence classes or residue classes. For more information see [7].

2.2.1 Modulo

In general a modulo n is defined as the resulting residue $\{0, 1, 2, \ldots, n-1\}$ after the division of a by n. For example 16 mod 12 equals 4. The classical definition is $a \equiv b$ modulo n if a and b are in the same residue class modulo n. This means that both a and b have the same residue when divided by n or that $a - b$ is a multiple of n. The notation used to express mathematically the modulo was first introduced by Gauss [8].

2.2.2 Group

A group $(G, \cdot)$ is a set of elements where a binary operation is defined satisfying the following axioms:

- Closure: $\forall\ x,\ y \in G$, the product $xy \in G$.
- Associativity: $(xy)z = x(yz), \quad \forall\ x,\ y,\ z \in G$.
- Identity element: $\exists$ a unique identity element $e \in G$ such that $ex = xe = x, \quad \forall\ x \in G$.
- Inverse element: $\forall\ x \in G,\ \exists\ y \in G$ such that $xy = yx = x^{-1}x = e$.

Groups may also consider the addition operation $(G, +)$ instead of multiplication. A common example of an infinite group is the group $\mathbb{Z}$ formed by the integers.

2.2.3 Ring

A ring is a set $(R, +, \cdot)$ with two binary operators, addition and multiplication, which satisfies the following conditions:

- Additive associativity: $(x + y) + z = x + (y + z), \forall\ x,\ y,\ z \in R$.
- Additive commutativity: $x + y = y + x, \quad \forall\ x,\ y \in R$.
- Additive identity element: $\exists$ an element $0 \in R$ such that $0 + x = x + 0 = x, \quad \forall\ x \in R$.
- Additive inverse element: $\forall\ x \in R\ \exists\ -x \in R$ such that $x + (-x) = (-x) + x = 0$.
- Left and right distributivity: $\forall\ x,\ y,\ z \in R, \quad x \cdot (y + z) = (x \cdot y) + (x \cdot z)$ and $(y + z) \cdot x = (y \cdot x) + (z \cdot x)$.

- Multiplicative associativity: $\forall\ x,\ y,\ z \in R, \quad (x \cdot y) \cdot z = x \cdot (y \cdot z)$.

Note that the ring multiplication does not have to be commutative, i.e. $a \cdot b \neq b \cdot a$. Rings that also fulfill the axiom of multiplication commutativity , $a \cdot b = b \cdot a$, are called commutative rings. The elements of a ring do not need to have multiplicative inverses. The elements that are invertible are called the units of a ring. The set of all units in R form a group in the ring multiplication. This group is denominated R^*. For more information see [9].

2.2.4 Field

The fields are a subset of the set of rings. In other words, all fields are rings but not all rings are fields. Fields differ from rings most importantly by the requirement that division may be possible and by the requirement that the multiplication operation in a field has to be commutative.
A field is a set F with at least two binary operations, addition "+" and multiplication "·" that fulfills the next axioms:

- Additive associativity: $(x + y) + z = x + (y + z), \forall\ x,\ y,\ z \in F$.
- Additive commutativity: $x + y = y + x, \quad \forall\ x,\ y \in F$.
- Additive identity element: $\exists$ an element $0 \in F$ such that $0 + x = x + 0 = x, \quad \forall\ x \in F$.
- Additive inverse element: $\forall\ x \in F\ \exists - x \in F$ such that $x + (-x) = (-x) + x = 0$.
- Additive distributivity: $x \cdot (y + z) = x \cdot y + x \cdot z, \quad \forall\ x,\ y,\ z \in F$.
- Product associativity: $(x \cdot y) \cdot z = x \cdot (y \cdot z), \quad \forall\ x,\ y,\ z \in F$.
- Product commutativity: $x \cdot y = y \cdot x, \quad \forall\ x,\ y \in F$.
- Product identity element: $\exists$ an identity element $e \in F$ such that $ex = xe = x, \quad \forall\ x \in F$.
- Product inverse element: $\forall\ x \in F$, $\exists y \in F$ such that $xy = yx = x^{-1}x = e$.
- Product distributivity: $(x + y) \cdot z = x \cdot z + y \cdot z, \quad \forall\ x,\ y,\ z \in F$.

A common example of a field is $\mathbb{Q}$, the field of rational numbers. Other important examples include the field of real numbers $\mathbb{R}$, the field of complex numbers $\mathbb{C}$ and, for any prime number p, the finite field of integers modulo p, denoted $\mathbb{Z}/p\mathbb{Z}$, $\mathbb{F}_p$ or $\mathrm{GF}(p)$.

If p is any prime number and n is a positive integer, we can have a finite field $\mathrm{GF}(p^n)$ with p^n elements; this is an extension field of the finite field $\mathrm{GF}(p) = \mathbb{Z}/p\mathbb{Z}$ that has p elements.

Fields have also the property that may be extended having as a result a new field which satisfies additional properties. These fields are called extension fields. The general idea of an extension field is to start with a base field and construct in some manner a larger field, which contains the base field. For example for any field K, the set $K(X)$ of rational functions with coefficients in K is also a field. On the other hand a subfield is a subset containing 0 and 1 that is closed under the operations of addition, negation, multiplication and multiplicative inverses for its nonzero elements.

It is common to construct an extension field of a given field K as a quotient ring of the polynomial ring $K[X]$ in order to “create” a root for a given polynomial $f(X)$. Suppose for instance that K does not contain any element x with $x^2 = -1$. Then the polynomial $X^2 + 1$ is irreducible in $K[X]$, consequently the ideal $(X^2 + 1)$ generated by this polynomial is maximal, and $L = K[X]/(X^2 + 1)$ is an extension field of K which does contain an element whose square is -1 (namely the residue class of X). For more information regarding fields see [10].

2.2.5 Lattice

A lattice [11] is a regular configuration of points in space with a periodic structure. Figure 2.2 shows some examples of 2 dimensional lattices.

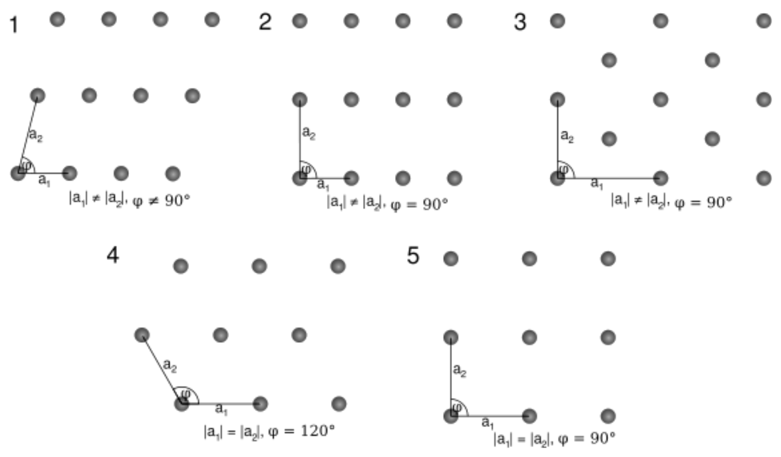
1
2
3
4
5
a_1
a_2
φ
$|a_1| \neq |a_2|, \varphi \neq 90°$
$|a_1| \neq |a_2|, \varphi = 90°$
$|a_1| \neq |a_2|, \varphi = 90°$
$|a_1| = |a_2|, \varphi = 120°$
$|a_1| = |a_2|, \varphi = 90°$

Figure 2.2: Lattice examples

In particular, for a linearly independent vector $v_1, \ldots, v_n \in \mathbb{R}^n$, the lattice generated is the set of vectors:

$$L(v_1, \ldots, v_n) = \left\{ \sum_{i=1}^{n} \alpha_i v_i \mid \alpha_i \in Z \right\} .$$

The vectors $v_1, \ldots, v_n$ are known as the basis of the lattice. The absolute value of the determinant of the vectors v_i is denoted by $d(L)$. One can think of a lattice as divisions of the whole R^n into equal polyhedral copies of an n-dimensional parallelepiped, known as the fundamental region of the lattice, then $d(L)$ is equal to the n-dimensional volume of this polyhedron.

2.3 Truncated Polynomial Rings

A polynomial ring is a ring formed by the set of polynomials with coefficients in a ring. In this section we describe polynomials and polynomial rings in order to introduce the truncated polynomial rings.

2.3.1 Polynomials

A polynomial in X with coefficients in a field K is an expression of the form:

$$F(X) = a_0 + a_1X + \cdots + a_{m-1}X^{m-1} + a_mX^m ,$$

where $a_0, \ldots, a_m$, the coefficients of $F(X)$, are elements of K and $X, X^2, \ldots, X^m$ are formal symbols ("the powers of X"). Such expressions can be added and multiplied, and then brought into the same form using the ordinary rules for manipulating algebraic expressions, such as associativity, commutativity, distributivity, or take common factors. Any term a_kX^k with zero coefficient, $a_k = 0$, may be omitted.
Using the summation symbol the same polynomial can be expressed more compactly as follows:

$$F(X) = \sum_{k=0}^{m} a_kX^k .$$

It is understood that the number of terms is finite, i.e. a_k is zero for all enough large values of k, in our case, for $k > m$. The degree of a polynomial is the largest k such that the coefficient a_k is not zero.

2.3.2 Polynomial Rings

Polynomials rings are essential in the NTRU public-key algorithm in order to generate random polynomials. A polynomial ring is defined by a ring which contains the values the coefficients can obtain and a delimiter or maximum degree when polynomials over one variable are represented. A polynomial ring $R[X]$ over the ring R in one variable X is formed by the set of all polynomials with coefficients in R. The elements of $R[X]$ are the polynomials with the form:

$$F(X) = a_0 + a_1X + a_2X^2 + \ldots + a_nX^n = \sum_{i=0}^{n} a_iX^i, \text{ where } a_i \in R \text{ and } 0 \leq i \leq n .$$

The symbol X is commonly called the *variable*, and the ring $R[X]$ is also called the ring of polynomials in one variable over R, to distinguish it from more general rings of polynomials in several variables. In general, X and its powers X^i are treated as formal symbols, not as elements of the field R. In order for $R[X]$ to form a ring, all powers of X have to be included, and this leads to the definition of polynomials as linear combinations of the powers of X, with coefficients in R for the ring $R[X]$.

A ring has two binary operations, addition and multiplication. In the case of the polynomial ring $R[X]$, these operations are explicitly given by the following formulas 2.1 and 2.2 respectively:

$$\left(\sum_{i=0}^{n} a_i X^i\right) + \left(\sum_{i=0}^{n} b_i X^i\right) = \sum_{i=0}^{n} (a_i + b_i) X^i \,. \tag{2.1}$$

$$\left(\sum_{i=0}^{n} a_i X^i\right) \cdot \left(\sum_{j=0}^{m} b_j X^j\right) = \sum_{n=0}^{m+n} \left(\sum_{k=0}^{n} a_k b_{n-k}\right) X^n \,. \tag{2.2}$$

In the formula 2.1 one of the polynomials may be extended by adding terms with coefficients values equal to zero, such that the same set of powers formally appears in both summands.

2.3.3 Truncated Polynomial Rings

As introduced in section 2.2.4, extension fields let us define polynomial rings. The NTRU public-key algorithm, explained in chapter 3, uses random polynomials which are generated from a polynomial ring of the form $R[X] = \mathbb{Z}[X]/(X^N - 1)$. The polynomials that form the ring $R[X]$ have a degree smaller than N. The polynomials in the truncated ring $R[X]$ are added in a regular way by adding their coefficients. The equation 2.1 shows the polynomial addition which stands in a truncated polynomial ring. The polynomial multiplication is a bit different since the resulting polynomial requires to satisfy the rule $X^N \equiv 1$. Said differently, the maximum degree of the resultant polynomial of a multiplication between two polynomials of the ring can not be greater than $N - 1$. The product operation of two polynomials in $R[X]$, shown in formula 2.2, is defined as $c(X) = a(X) * b(X)$ where c_k is the k^{th} coefficient of $c(X)$ and is computed as shown in formula 2.3:

$$c_k = a_0 b_k + a_1 b_{k-1} + \ldots + a_k b_0 + a_{k+1} b_{N-1} + a_{k+2} b_{N-2} + \ldots + a_{N-1} b_{k+1} \,. \tag{2.3}$$

The product of polynomials in $R[X]$ is also called the star multiplication.

2.4 Möbius Functions

2.4.1 Möbius Function

The Möbius function $\mu(n)$ [12] is an important multiplicative function in number theory and combinatorics. The Möbius function is a special case of a more general object in combinatorics. The $\mu(n)$ function is defined for all positive integers n and has its values in {-1, 0, 1} depending on the factorization of n into prime factors. It is defined as follows:

$$\mu(n) = \begin{cases} 1 & \text{if } n \text{ is a positive integer composed of an even number of distinct prime factors} \\ -1 & \text{if } n \text{ is a positive integer composed of an odd number of distinct prime factors} \\ 0 & \text{if } n \text{ is composed of one or more prime factor repeated} \end{cases}$$

Figure 2.3 represents the firsts 50 values of the Möbius function:

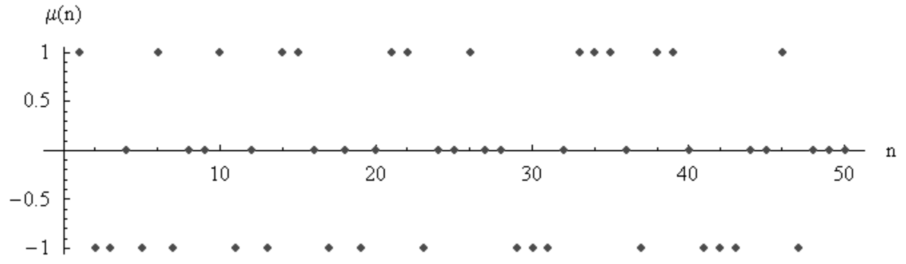

Figure 2.3: The Möbius function

The figure 2.3 shows the different values of $\mu(n)$ (-1, 0 or 1) in the y-axis for first fifty natural numbers in the x-axis.

2.4.2 Möbius Inversion

When a partially ordered set of natural numbers (ordered by divisibility) is replaced by other locally finite partially ordered sets, one has other Möbius inversion formulas [13]. The classic version states that if $g(n)$ and $f(n)$ are arithmetic functions both conditions are equivalent:

$$f(n) = \sum_{d\,|\,n} g(d) \quad \forall\, n \in \mathbb{N} \tag{2.4}$$

then,

$$g(n) = \sum_{d\,|\,n} \mu(d) f(\frac{n}{d}) \quad \forall\, n \in \mathbb{N} \tag{2.5}$$

where μ is the Möbius function and the sums extend over all positive divisors d of n. In effect, the original $f(n)$ can be determined given $g(n)$ by using the inversion function $\mu(d)$. In the notation above, f is called the Möbius transform of g, and formula 2.5 is called the Möbius inversion formula.

To proof the relation between Möbius transform and Möbius inversion it should be assumed $n \in \mathbb{N}$ having:

$$\sum_{d\,|\,n} \mu(d) f(\frac{n}{d}) = \sum_{d\,|\,n} \mu(d) \sum_{e\,|\,n/d} g(e) = \sum_{k\,|\,n} \sum_{d\,|\,k} \mu(d) g(\frac{n}{k}) = \sum_{k\,|\,n} g(\frac{n}{k}) \sum_{d\,|\,k} \mu(d) = g(n). \tag{2.6}$$

2.5 Invertibility in Truncated Polynomial Rings

In order to be able to compute the inverse of a randomly chosen polynomial in a certain polynomial ring R_q defined as

$$R_q = (\mathbb{Z}/q\mathbb{Z})[X]/(X^N - 1)\,,$$

it is important to take into account that not every polynomial might be invertible in the ring. The NTRU cryptosystem key generation is based on the computation of the inverse of a randomly generated polynomial from a polynomial ring.
It is then necessary to know the probability that a randomly chosen polynomial in the ring R_q has an inverse. The goal of the probability calculus is to be able to generate a ring with as many invertible elements as possible and how to choose randomly a polynomial to assure it has an inverse. The section starts explaining how this probability can be calculated and continues describing different methods to increase the chances that a random polynomial $f(X)$ has an inverse in R_q. The content beneath has been extracted from [14].

Assuming the ring of truncated polynomials $R_q = (\mathbb{Z}/q\mathbb{Z})[X]/(X^N - 1)$, where $N \geq 2$ and q is a positive integer. And having R_q^* as the group of inverses of elements in the ring R_q

$$R_q^* = \{f \in R_q : f * g = 1 \quad \text{for some } g \in R_q\}\,.$$

Then the probability of choosing a polynomial that is invertible is the ratio between the cardinality of the group of inverses and the number of all polynomials in the ring, $\dfrac{\#R_q^*}{\#R_q}$.

If $q = q_1 q_2$ and $gcd(q_1, q_2) = 1$, the Chinese Remainder theorem [15] allows us to state:

$$R_q = R_{q_1} \times R_{q_2} \quad \text{and} \quad R_q^* = R_{q_1}^* \times R_{q_2}^*\,.$$

Finally if q is a power of a prime p, then the following theorem holds.

Theorem

Let p be a prime, q be a power of p ($q = p^k$) and $N \geq 2$ be an integer where $gcd(p, N) = 1$. If n is the smallest positive integer $n \geq 1$ that fulfills:

$$p^n \equiv 1 \pmod{N}\,, \tag{2.7}$$

and if for each integer d that divides n, $d|n$, we have

$$v_d = \frac{1}{d}\sum_{e|d} \mu\left(\frac{d}{e}\right) gcd(N, p^e - 1)\,, \tag{2.8}$$

where μ represents the Möbius function (see section 2.4.1) . Then,

$$\frac{\#R_q^*}{\#R_q} = \prod_{d|n}\left(1 - \frac{1}{p^d}\right)^{v_d}\,. \tag{2.9}$$

Now if N is selected to be a prime number, then $v_d = 0$ for $1 < d < n$, obtaining:

$$\frac{\#R_q^*}{\#R_q} = \left(1 - \frac{1}{p}\right)\left(1 - \frac{1}{p^n}\right)^{(N-1)/n}\,. \tag{2.10}$$

Remark 1

Some of the non-invertible elements in the ring R_q are easy to identify. The evaluation map:

$$R_q \to \mathbb{Z}/q\mathbb{Z} \quad f(X) \longmapsto f(1)\,,$$

is a homomorphism of rings, i.e: $f_1(1) * f_2(1) = f_1 * f_2(1)$ and $f_1(1) + f_2(1) = f_1 + f_2(1)$ since f_1 and f_2 are polynomials satisfying this properties for $X = 1$ or as expressed for $f(1)$. This induces a group homomorphism $R_q^* \to (\mathbb{Z}/q\mathbb{Z})^*$. Since it is well-known that

$$(\mathbb{Z}/q\mathbb{Z})^* \cong \{a \in \mathbb{Z}/q\mathbb{Z} : gcd(a, q) = 1\},$$

it can be observed that if $f(1)$ has a common factor with q, then it cannot be invertible. Thus, when choosing a polynomial randomly it should be required that $gcd(f(1), q) = 1$. The polynomials with $f(1) = 0$ must be avoided. So the selected polynomials might be in the subsets of R_q and R_q^* satisfying $f(1) = 1$. These subsets are referred as $R_q(1)$ and $R_q^*(1)$ respectively.

The values of $f(1)$ are equidistributed in $\mathbb{Z}/q\mathbb{Z}$ as f ranges over R_q. From this it can be observed that:

$$\#R_q(1) = q^{-1}\#R_q\,.$$

Values of $f(1)$ are also equidistributed when f ranges over R_q^*. In this case:

$$\#R_q^*(1) = \varphi(q^{-1})\#R_q,$$

where φ is the Euler phi function. Particularly if $q = p^k$, then

$$\varphi(q) = p^k - p^{k-1} .$$

So if f is "intelligently" chosen, satisfying that $f(1) = 1$, then

$$\frac{\#R_q^*(1)}{\#R_q(1)} = \left(1 - \frac{1}{p}\right)^{-1} \frac{\#R_q^*}{\#R_q} .$$

Since a smaller p is desirable in applications due to resource savings, for N being prime it is obtained that:

$$\frac{\#R_q^*(1)}{\#R_q(1)} = \left(1 - \frac{1}{p}\right)^{-1} \frac{\#R_q^*}{\#R_q} \approx 1 - \frac{N-1}{np^n} .$$

Remark 2

From equation 2.10 in order to maximize the probability of getting a unit (see subsection 2.2.3 for unit definition) in R_q, it is desirable to choose N and p such that the order n of p in $(\mathbb{Z}/N\mathbb{Z})^*$ is as large as possible. The value of n is easy to obtain from values of N and p. Although for cryptographic purposes n should be large for a single N and two values of p, typically $p = 2$ and $p = 3$.
The possible orders of elements in $(\mathbb{Z}/N\mathbb{Z})^*$ are divisors of $\varphi(N)$, so if N is prime, the possible orders are divisors of $N - 1$. For this reason N should be selected such that $N - 1$ has few divisors.
A systematical way to achieve this is as follows. If N has the form $N = 2M + 1$ with M being prime, then divisors of $N - 1$ are $1, 2, M$ and $2M$. M is known as a Sophie Germain prime [16]. Hence if N does not divide $p^2 - 1$, the corresponding n might be M or $2M$. The probability that a randomly chosen polynomial satisfying $f(1) = 1$ is invertible is:

$$1 - \frac{N-1}{Mp^M} = 1 - \frac{2}{p^M} .$$

For example, if N is chosen 103, then M is 51, since $103 = 2 \cdot 51 + 1$. And since $p \geq 2$, the probability of having an invertible polynomial is almost 1:

$$1 - \frac{2}{2^{103}} .$$

Table 2.1 shows some values for the parameters N, p, n_p (n_p being the smallest integer that satisfies $p^n \equiv 1 \pmod{N}$) and $Prob_p$ that represents the probability that a random $f(X)$ in R_q is **not** invertible in this ring.

The values of N printed in bold in the table 2.1 are the ones with the form $2M + 1$, being M a Sophie Germain prime [16]. Values for N with this form have a bigger set of invertible elements or units.

Table 2.1: Probability $f(X)$ is **not** invertible in R_{p^k}

N	p	n_p	$Prob_p$		N	p	n_p	$Prob_p$
47	2	23	$10^{-7.22}$		**47**	3	23	$10^{-11.27}$
59	2	58	$10^{-17.46}$		**59**	3	29	$10^{-14.14}$
71	2	35	$10^{-10.84}$		71	3	35	$10^{-17.00}$
107	2	106	$10^{-31.91}$		**107**	3	53	$10^{-25.59}$
127	2	7	$10^{-3.36}$		127	3	126	$10^{-60.12}$
167	2	83	$10^{-25.29}$		**167**	3	83	$10^{-39.90}$
229	2	76	$10^{-23.36}$		229	3	57	$10^{-27.80}$
349	2	348	$10^{-104.76}$		349	3	174	$10^{-83.32}$
503	2	251	$10^{-75.86}$		**503**	3	251	$10^{-120.06}$
1019	2	1018	$10^{-306.45}$		**1019**	3	509	$10^{-243.16}$
1093	2	364	$10^{-110.05}$		1093	3	7	$10^{-5.53}$

Chapter 3

NTRU Cryptosystem

The NTRU Public Key Cryptosystem (PKC), also known as NTRUEncrypt, is an asymmetric key encryption algorithm for public key cryptography. NTRU Cryptosystems, Inc. was founded in 1996 by Joseph H. Silverman, Jeffrey Hoffstein, Jill Pipher and Daniel Lieman. The name NTRU is an abbreviation for N-th degree truncated polynomial ring. The main characteristic is that during the encryption and decryption the polynomial multiplication is the most complex operation, which is much faster than other asymmetric cryptosystems, such as RSA, El Gamal and elliptic curve cryptography. This chapter presents the NTRU PKC as described in [17]. First, we describe the parameters of the cryptosystem in order to afterwards explain how the cryptosystem works. Main operations involve polynomial algebra, as the computation of a polynomial inverse for the key generation or the multiplication in a truncated polynomial ring for encryption and decryption. Finally, we discuss the security provided by the polynomial operations and the difficulty of finding a very short vector in a lattice.

3.1 Algorithm Description

3.1.1 Notation

NTRU public-key algorithm is well described using the ring of polynomials

$$R = \mathbb{Z}[X]/(X^N - 1)\,.$$

The polynomials conforming R have integer coefficients:

$$a(X) = a_0 + a_1X + a_2X^2 + \ldots + a_{N-1}X^{N-1}\,,$$

that are multiplied together using the extra rule $X^N \equiv 1$. The product

$$c(X) = a(X) * b(X)$$

is given by

$$c_k = a_0b_k + a_1b_{k-1} + \ldots + a_{N-1}b_{k+1} = \sum_{i+j\equiv k \mod N} a_ib_j\,.$$

In particular, if we write $a(X)$, $b(X)$, and $c(X)$ as vectors

$$a = [a_0, a_1, \ldots, a_{N-1}], b = [b_0, b_1, \ldots; b_{N-1}], c = [c0, c1, \ldots, c_{N-1}],$$

then $c = a * b$ is the convolution product of two vectors having c a size of N positions.

The NTRU public-key algorithm is defined by the following parameters:

N	The degree parameter. Defines the degree $N-1$ of the polynomials in R.
q	Large modulo. Polynomial coefficients are reduced modulo q.
p	Small modulo. The coefficients of the message are reduced modulo p in decryption.
df	Private key space. Fixes the polynomial form defining the number of positive ones for the private key f, the negative ones are fixed by $df-1$.
dg	Public key space. Fixes the polynomial form defining the number of positive and negative ones for the random polynomial g used to calculate the public key.
dr	Blinding value space. Fixes the polynomial form defining the number of positive and negative ones of the random polynomial r used in the encryption process.
dm	Plaintext space. NTRUEncrypt requires the message to be in a polynomial form, therefore the need of dm to define the form of the message to be encrypted.

The more relevant properties of NTRU PKC are the following:

1. The parameters (N, p, q) are public and p and q must satisfy $gcd(p, q) = 1$.
2. Coefficients of polynomials are bounded modulo p and modulo q.
3. The inverse of $a(X)$ mod q is the polynomial $A(X) \in R$ satisfying $a(X) * A(X) \equiv 1$ mod q.

3.1.2 Key Generation

The key generation consists in the generation of the private key (f, f_p) and the public key h.

Choose random polynomials f and g from R with "small" coefficients. Meaning "small" much smaller than q, typically {-1,0,1} for $p = 3$. Then compute f_p, i.e. the inverse of $f \pmod{p}$ defined by

$$f * f_p = 1 \pmod{p}\,.$$

Compute f_q, the inverse of $f \pmod{q}$ that analogously satisfies the requirement:

$$f * f_q = 1 \pmod{q}\,.$$

Compute the polynomial

$$h = g * p \cdot f_q\,.$$

The public key is h and the private key is the set (f, f_p).

3.1.3 Encryption

The plaintext m is a polynomial with coefficients taken mod p. Note that convert the message m to a polynomial form is not part of NTRU public-key algorithm. Choose a blinding message r randomly from R with small coefficients. The ciphertext is

$$e = r * h + m \pmod q .$$

3.1.4 Decryption

The decryption returns the message m from the encrypted message e using the private key (f, f_p).

Compute

$$a = e * f \pmod q ,$$

choosing the coefficients of a to satisfy $-q/2 \leq a_i < q/2$.

Reduce a modulo p:

$$b = a \pmod p .$$

Compute

$$c = b * f_p \pmod p .$$

Then c mod p is equal to the plaintext m.

3.1.5 Mathematical Principle

The private key (f, f_p) is used on decryption to cancel f_q from the encrypted message and be able to cancel r and g reducing modulo p. The mathematical principle of decryption of the NTRU public-key algorithm is based on the following equations:

$$a = f * e \pmod q = f * (r * pf_q * g + m) \pmod q = pr * g + f * m \pmod q .$$

Multiplying the encrypted message by f cancels f_q but leaves f multiplied by the message. To get rid of $pr * g$ it is just necessary to reduce modulo p. This is possible since r and g are "smaller" polynomials with coefficients much smaller than q. This assures that if $c = pr * g$, any coefficient c_k is smaller than q, which means all coefficients of c have p as a common divisor. Then,

$$a = pr * g + f * m \pmod q \equiv f * m \pmod p .$$

The final step cancels f to obtain m modulo p. For this reason f_p is calculated in order to state the following,

$$c = f_p * a \pmod p = f_p * f * m \pmod p = m \pmod p .$$

This is how NTRU Cryptosystems, Inc. describes how to recover m. For this matter f have to satisfy the next properties:

1. f is invertible mod p
2. f is invertible mod q
3. f is small

3.2 Parameter Selection

The selection of the NTRU PKC parameters defines the different levels of security. It is very important that p and q have no common factors. This is indispensable as explained in section 2.5 to be able to compute the inverse of a certain polynomial. In table 3.1 is shown the recommended parameters for NTRU PKC security levels.

Table 3.1: NTRU Security parameters.

	N	q	p
Moderate Security	167	128	3
Standard Security	251	128	3
High Security	347	128	3
Highest Security	503	256	3

Typical parameter sets that yield security levels similar to 1024-bit RSA and 4096-bit RSA respectively are $(N, p, q) = (251, 3, 128)$ and $(N, p, q) = (503, 3, 256)$.

The public parameters (N, p, q) define the level security together with the parameters df, dg and dr. The parameters df, dg and dr define different spaces. In NTRUEncrypt for the parameter $p = 3$ a space $\mathcal{L}$ is defined as follows,

$$\mathcal{L}(d_1, d_2) = \{F \in R : F \text{ has } d_1 \text{ coefficients equal } 1,\ d_2 \text{ coefficients equal } -1 \text{ and the rest } 0\} .$$

Choosing the integer values for the parameters df, dr, dg set the spaces:

$$\mathcal{L}_f(df, df-1), \quad \mathcal{L}_g(dg, dg), \quad \mathcal{L}_r(dr, dr),$$

where $f \in \mathcal{L}_f,\ g \in \mathcal{L}_g$ and $r \in \mathcal{L}_r$. The standard values have changed over the past years and they may be susceptible to changes in the future. In an effort to create a method to design the security parameters, NTRU Cryptosystems, Inc. has published in [18] an algorithm which computes all the parameter values from an input k. The parameter k is the security parameter.

The algorithm computes the parameter values with binary underlying polynomials ($p = 2$). This algorithm also holds for ternary polynomials ($p = 3$).

We proceed to explain how this algorithm works.
The algorithm receives as input the security parameter k.

1. First N is searched to be the first prime greater than $3k + 8$.

2. Then d is fixed to be the smallest integer that satisfies:

$$\frac{1}{\sqrt{N}}\binom{N/2}{d/2} > 2^k\,.$$

   Now it is set $df = dr = d$ and $dg = N/2$.

3. For the message parameter d_m has to be the largest integer that:

$$2^{N-1}\sum_{i=0}^{d_m}\binom{N}{i} < 2^{-40}\,.$$

   If $\frac{1}{\sqrt{N}}\binom{N/2}{d_m} < 2^k$ then N should be increased to the next greater prime and the procedure is restarted to step 2.

4. Next q is set to be the first prime greater than $4d + 1$

5. Verify that the order of $q \pmod N$ is $(N - 1)$ or $(N - 1/2)$. If the order of q is different, then increase q to the next prime number until this statement holds.

6. Calculate $c = \sqrt{\frac{4\pi e\sqrt{d(N-d)/N}\sqrt{d_{m0}(N-d_{m0}/N}}{q}}$

   From table 3.2 obtain values A and B and check,

$$AN - B - max\left(\log_2\left(1-\left(1-\prod_{i=0}^{d-1}\left(1-\frac{r}{\sqrt{N-i}}\right)\right)^N\right)+Ar/2\right) < k\,.$$

   Finally output $\{N,\ q,\ p = 2,\ d_F,\ dr,\ dg,\ dm_0\}$. Otherwise increase N to the next largest prime and return to step 2.

More information on A and B constants can be found in the subsection 3.3.2.

Table 3.3 summarizes the parameter sets.

Table 3.2: Extrapolated bit security constants depending on (c, a).

c	a	A	B
1.73	0.53	0.3563	-2.263
2.6	0.8	0.4245	-3.440
3.7	2.7	0.4512	+0.218
5.3	1.4	0.6492	-5.436

Table 3.3: NTRU recommended parameters.

	N	p	q	df	dg	dr
NTRU167:3	167	3	128	61	20	18
NTRU251:3	251	3	128	50	24	16
NTRU503:3	503	3	256	216	72	55
NTRU167:2	167	2	127	45	35	18
NTRU251:2	251	2	127	35	35	22
NTRU503:2	503	2	253	155	100	65

3.3 Security Analysis

The NTRUEncrypt PKC is based on the shortest vector problem, SVP, in a lattice. When the lattice is large enough it is difficult to guess a random chosen polynomial and even harder to calculate this polynomial from its inverse. The process of solving this problem is called "Lattice Reduction". Although the hypothetical hardness of any public key cryptosystem can only be measured by the most effective known attack against it. Additionally, the attack efficiency is most of the times related with the parameter generation algorithm of the cryptosystem. As an example RSA is weak if (p, q, e) are chosen such that $d = e^{-1} \pmod{(p-1)(q-1)}$ is relatively small, or if p, q values are too close. In the NTRU PKC the security level depends directly on the public parameters N, p and q. However it is important to note that the value associated to the parameters df, dg and dr are crucial in order to achieve a certain security level. Until relatively recently, the hardness of NTRU PKC was subject to the lattice and meet-in-the-middle attacks. Per contra, the hybrid attack introduced in 2007 combining both attacks it has been proven to be the best known attack against NTRU PKC. The following sections explain the main ideas behind these attacks.

3.3.1 Meet-in-the-middle Attack

The meet-in-the-middle attack due to Odlyzko [19] has been one of the most effective known attacks against NTRUEncrypt (with the parameters recommended by NTRU). The attack relies on very particular properties of the NTRU lattice, more specifically in the structure of the short vectors and the presence of orthogonal q-vectors. Like most meet-in-the-middle attacks it essentially reduces the time of an exhaustive search to the square root. The meet-in-the-middle attack attempts to find a value in each of the ranges and domains of the composition of two functions such that the forward mapping of one through the first function is the same as the inverse image of the other through the second function. Let us take a look to the mentioned attack as described in [20]. The idea is to search for f in the form of $f = f_1 + f_2$, where f_1 and f_2 are both of length $N/2$ and have $df/2$ ones. Using the properties that:

$$\begin{gathered} f * h = g \pmod q \\ (f_1 + f_2) * h = g \pmod q \\ f_1 * h = g - f_2 * h \pmod q\,. \end{gathered}$$

Since g is a small polynomial, with binary $\{0,1\}$ or trinary $\{-1,0,1\}$ coefficients, $f_1 * h$ and $-f_2 * h$ can only differ by 0 or 1. With this in mind the attack searches for the pair (f_1, f_2) such that the corresponding coefficients have approximately the same value. Note that f does not have the property that half of its ones falls in the first $N/2$ entries, but is known that at least one rotation of f satisfies this property.

The attack first enumerates all f_1, which takes $\binom{N/2}{d/2}$ steps. This also occupies about $\binom{N/2}{d/2}$ coefficients. If we call T_l the time of a lookup, read or write a f_1 into a table, and T_c is time that takes a star multiplication (see formula 2.3) we have that the cost in time is:

$$T_1 = \binom{N/2}{d/2}(T_c + T_l)\,.$$

The vectors f_2 are enumerated, which takes also $\binom{N/2}{d/2}$ steps. Then f_2 is check against f_1 susceptible to have the same coefficients or that might have changed by adding 1 (if g binary) or also subtracting 1 in the case of g being trinary. Then a candidate $f = f_1 * f_2$ is formed and $f * h \pmod q$ is checked. If it is binary or ternary then returns f. This second part costs in time:

$$\begin{aligned} T_2 &= \#f_2 * (T_c + (\text{expected } f_1 \text{ for } f_2) * T_l + (\text{expected hits per } f_2) * T_c) = \\ &= \binom{N/2}{d/2}\left(T_c + \frac{2k}{q}T_l + \frac{\binom{N/2}{d/2}}{2^k}T_c\right)\,. \end{aligned}$$

Several improvements can be done to reduce T_2, as storing $f_1 * h \pmod q$ when storing f_1 that let us calculate $f_1 * h - f_2 * h \pmod q$ instead of calculating $f * h \pmod q$ reducing convolution time approximately to T_c/d. Improvements described in [20] end up with the next time and storage requirements $\frac{\binom{N/2}{d/2}}{\sqrt{N}}$.

3.3.2 Lattice Attack

Lattices have been recently introduced in cryptography taking advantage of the shortest vector problem, SVP. The SVP is the main problem associated to Lattices. The procedure to solve the SVP is named Lattice reduction. A lattice has many bases that normally contain very large vectors compared to the shortest nonzero vector. The SVP should output the shortest nonzero vector of a given lattice. However is generally more interesting to obtain the nonzero lattice vector with a norm greater than the shortest nonzero vector norm bounded by some tolerance factor, reducing the complexity.

From the Laticce-based attack article of Oded Regev [21] we extracted some time costs referring to the SVP. The well-known polynomial time algorithm of Lenstra, Lenstra, and Lovász (LLL) [22] from 1982 achieves an approximation factor of $2^{O(n)}$, where n is the dimension of the lattice. In 1987, Schnorr presented an improved algorithm obtaining an approximation factor that is slightly subexponential, namely $2^{O(n(\log\log n)2/\log n)}$. This was recently improved to $2^{O(n\log\log n/\log n)}$ [23]. We should also mention that if one insists on an exact solution to SVP, the best algorithm has a running time of $2^{O(n)}$ [23]. One might expect SVP to be NP-hard to approximate to within very large factors. However, the best known result only shows that approximating SVP to within factors $2^{(\log n)^{\frac{1}{2}-\varepsilon}}$ is NP-hard (under randomized quasi-polynomial time reductions)[24]. Moreover, SVP is not believed to be NPhard to approximate to within factors above $\sqrt{n/\log n}$ [25, 26, 27], since for such approximation factors it lies in classes such as $NP \cap coNP$. On the practical side, it is difficult to say the dimension n where solving the SVP becomes infeasible with today's computing power. A reasonable guess would be that taking n to be several hundreds make the problem extremely difficult. To conclude, the problem of approximating SVP to within polynomial factors n^c for $c \geq \frac{1}{2}$ seems to be very difficult, however it is not believed to be NP-hard.

In the NTRU PKC lattice-based attacks [28] may lead the attacker to recover the private key from public key h, or recover the plaintext from the ciphertext.

The NTRU lattice L_h is a lattice of dimension $2N$ generated by the row vectors of a matrix of the following form:

$$L_h = \{(f,g) \in R^2 : g \equiv h * f/p \pmod q\}, \text{satisfying}$$

$$\dim(L_h) = 2N \quad \text{and} \quad \text{Disc}(L_h) = q^N\,.$$

$$\begin{pmatrix} \alpha & 0 & \dots & 0 & h_0 & h_1 & \dots & h_{N-1} \\ 0 & \alpha & \dots & 0 & h_{N-1} & h_0 & \dots & h_{N-2} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \alpha & h_1 & h_2 & \dots & h_0 \\ 0 & 0 & \dots & 0 & q & 0 & \dots & 0 \\ 0 & 0 & \dots & 0 & 0 & q & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 0 & 0 & 0 & \dots & q \end{pmatrix}$$

where $h = (h_0, \dots, h_{N-1})$ is a known list of integers and the constant α is a balancing constant chosen to maximize the efficiency of the search for small vectors in the lattice. The attacker knows the lattice contains a short vector $v = (\alpha f_0, \dots, \alpha f_{N-1}, g_0, \dots, g_{N-1})$. And also knows the public key $h = f^{-1}g$. When f is of the form $f = 1 + pF$ the best lattice attack on the private key involves resolving the Close Vector Problem, CVP. An NTRU lattice of this form has been found empirically to be characterized by these two parameters:

$$a = N/q, \quad c = \sqrt{4\pi e \|F\| \|g\| / q}\,.$$

Running times T for lattice reduction have report experimentally that it is exponential in $AN + B$:

$$T = 10^{AN+B}$$

for some empirically defined constants A and B that can be observed in table 3.2.

The bit security in terms of computational complexity is $AN + B$ and can be converted to time in MIPS-years using the equality 80 bits $\sim 10^{12}$ MIPS-years [29].

3.3.3 Hybrid Attacks

In the 27th International Cryptology Conference, CRYPTO 2007 [30], Nick Howgrave-Graham presented a new attack against NTRUEncrypt, combining lattice reduction and a meet-in-the-middle strategy [31].

The closest vector problem (CVP) can be solved efficiently in the case that the given point in space is very close to a lattice vector [32]. The CVP algorithm takes a time t and a set S that has the property that it includes at least one point $v_0 \in S$ which is very close to a lattice vector. Therefore v_0 can be found in time $O(|S|t)$ by exhaustively enumerating the set S. In [31] it is shown that if the points of S can be represented as $S = S' \oplus S'$, i.e. for every $(v, v') \in S \cdot S'$ there exists a $v'' \in S'$ such that $v = v' + v''$, then there exist conditions under which there is actually an efficient meet-in-the-middle algorithm on this space to find the point v_0 in time $O(|S|^{1/2}t)$. We can translate this CVP result to a result about lattice basis reduction by defining the set S to be some linear combinations of the last $n - m$ rows of a given basis $\{b_1 \ldots, b_n\}$, and then using the CVP algorithm on the elements of S and the basis $\{b_1, \ldots, b_m\}$. It is also pointed that a similar approach is taken by Schnorr in [33] for reducing generic lattices with the SHORT algorithm. Schnorr also suggests that "birthday" improvements might be possible for his method (generalizing results from [34]) but concludes that generally storage requirements may be prohibitive.

In the case of searching for the NTRUEncrypt private key, meet-in-the-middle techniques are possible as explained in section 3.3.1 but [31] shows that Odlyzko's storage ideas may be generalized to remain efficient even when used after lattice reduction, optimizing the set S for the structure of the NTRUEncrypt private key.

Chapter 4

Software Implementation

4.1 Code Specifications

This section gives an overall view of the implementation of the NTRU PKC. Sections are divided by the three major functionalities of the NTRU PKC: *Key Generation*, *Encryption* and *Decryption*. These three processes are characterized together with the necessary functions to implement them.

4.1.1 Key Generation

Key Generation creates the private key set (f, f_p), and the public key h as shown in figure 4.1.

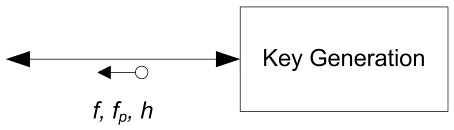

Figure 4.1: Key Generation

4.1.1.1 Random Polynomial

During *Key Generation* the process *Random Polynomial* is invoked to generate the polynomial f, part of the private key. Also *Random Polynomial* is required to generate the polynomial g in order to calculate the public key h. These polynomials are generated with random coefficients from a truncated ring of polynomials R. *Random Polynomial* receives the number of positive and negative ones and generates the random polynomial of N coefficients.

As it can be observed in figure 4.2 *Random Polynomial* takes the number of positive and negative ones and outputs a randomly generated polynomial r. The distinction between positive

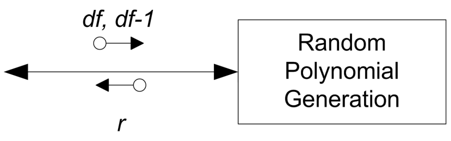

Figure 4.2: Random Polynomial

and negative ones is necessary because the polynomial f has different values for these, having df positive ones and $df - 1$ negative ones.

4.1.1.2 Inversion modulo p

Inversion modulo p computes the inverse of a polynomial f in modulo p such that $f * f_p \equiv 1 \pmod{p}$ illustrated in figure 4.3.

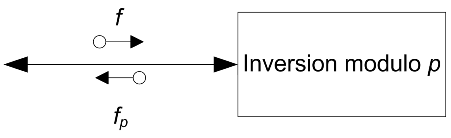

Figure 4.3: Inversion modulo p

For the calculus of f_p, the addition and the subtraction of polynomials modulo p are necessary.

4.1.1.3 Addition modulo p

The *Addition modulo p* calculates the sum of two given polynomials as in formula 2.1 reducing the coefficients modulo p, $addPol = pol1 + pol2 \pmod{p}$. Figure 4.4 specifies the inputs and output parameters.

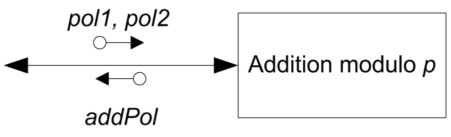

Figure 4.4: Addition modulo p

4.1.1.4 Subtraction modulo p

The *Subtraction modulo p* represented in figure 4.5 calculates the difference of two given polynomials modulo p, $subPol = pol1 - pol2 \pmod{p}$.

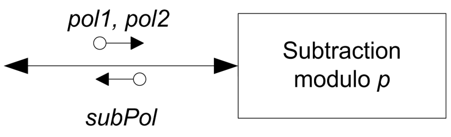

Figure 4.5: Subtraction modulo p

4.1.1.5 Inversion modulo q

During *Key Generation* the polynomial f_q, the inverse of f modulo q, is also computed. The polynomial f_q is necessary, together with g, to calculate the public key h as shown in figure 4.6. *Inversion modulo* q computes the inverse of a certain polynomial f modulo q in f_q, computing $f_q = f^{-1} \pmod q$ which satisfies $f * f_q \equiv 1 \pmod q$.

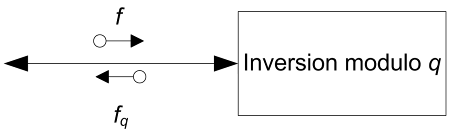

Figure 4.6: Inversion modulo p

Analogously to the *Inversion modulo p*, the *Inversion modulo q* requires to perform the addition of polynomials but in this case modulo q.

4.1.1.6 Addition modulo q

Addition modulo q performs the addition $addPol = pol1 + pol2 \pmod q$. Figure 4.7 specifies the required inputs to obtain the output addPol.

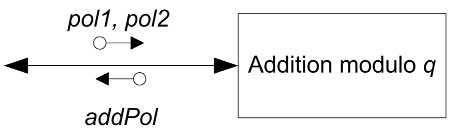

Figure 4.7: Addition modulo q

4.1.1.7 Star Multiplication modulo q

Star Multiplication modulo q is required in *Key Generation*, *Encryption* and *Decryption*. The *Star Multiplication modulo* q computes in $mulPol$ the polynomial product, see formula 2.2, given two polynomials, $pol1$ and $pol2$, and the parameter q. Having $mulpol = pol1 * pol2 \pmod q$ as shown in figure 4.8 where each coefficient is computed as in formula 2.3.

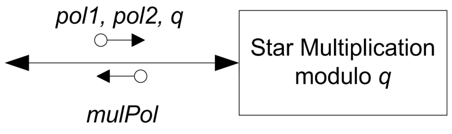

Figure 4.8: Star Multiplication modulo p specification

Note that q is generally a power of two ($q = 2^k$) when using ternary polynomials ($p = 3$). When q has the form 2^k the *Inverse modulo q* computes first the inverse modulo 2 to later convert it to the inverse in modulo 2^k or q through Newton's iterations. This requires reducing modulo different values (powers of two) during the Newton Iteration when computing the polynomial multiplication. This is the main reason why multiplication receives the parameter denoted q since when computing the inverse this parameter might be a power of 2 smaller than q.
Finally we note that the resulting polynomial $mulPol$ has no more than N coefficients since the multiplication is done in the truncated polynomial ring R, where $g = X^N - 1$ is the irreducible polynomial.

The figure 4.9 summarizes the *Key Generation* process modular structure.

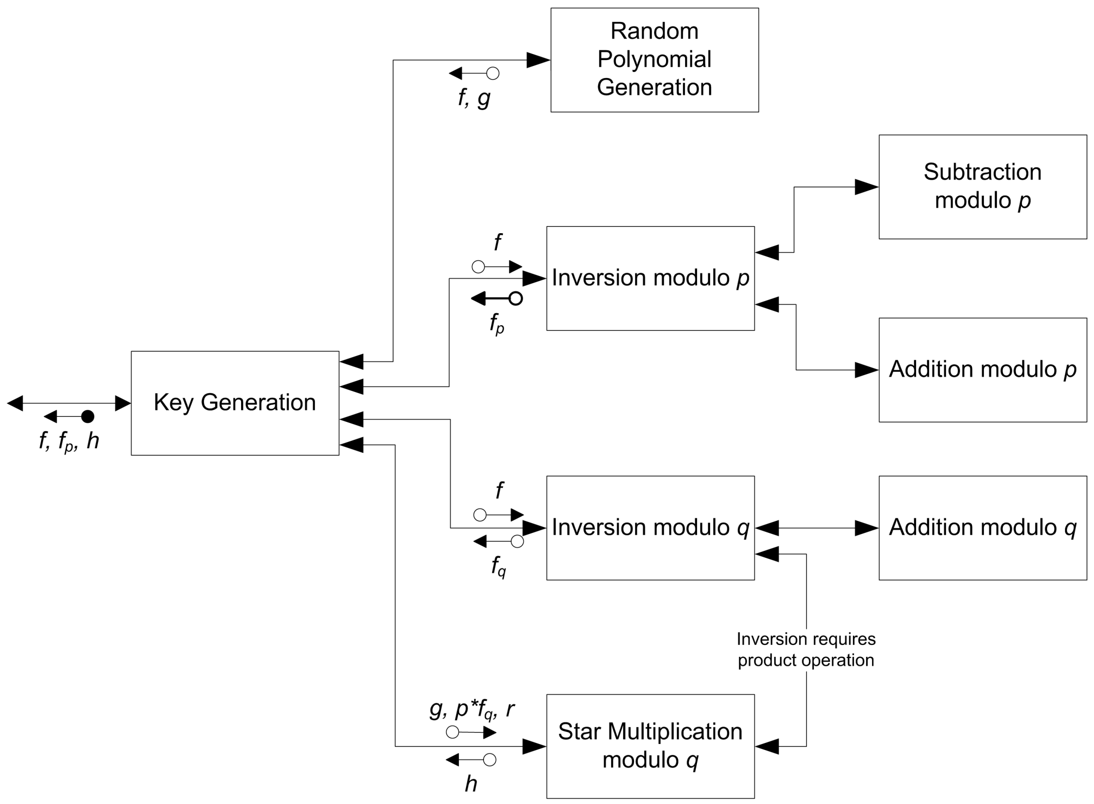

Figure 4.9: Diagram of processes invoked by Key Generation

4.1.2 Encryption

Encryption is the simplest part in the NTRU PKC. *Encryption* only requires to generate a random polynomial r from the ring R that obscures the message. Then the polynomial r is multiplied by the public key h. And finally the product of r and h is added to the the desired message to encrypt. This means *Encryption* just needs to receive a message in the polynomial form m and the public key h illustrated in figure 4.10.

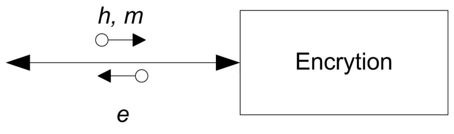

Figure 4.10: Encryption specification

The encrypted message $e = r * m + h \pmod{q}$ is the output.

The figure 4.11 illustrates the modular structure of the *Encryption* process.

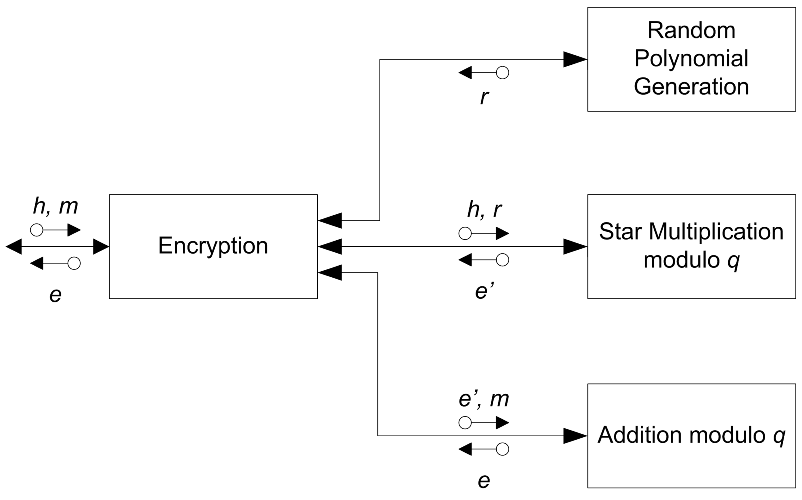

Figure 4.11: Diagram of processes invoked by Encryption

Random Polynomial which has been introduced in section 4.1.1.1 is used to generate r.

4.1.3 Decryption

The *Decryption* process requires the encrypted message e and the private key set (f, f_p) to decrypt the encrypted message e into the clear message c. Figure 4.12 specifies the required inputs and output parameters.

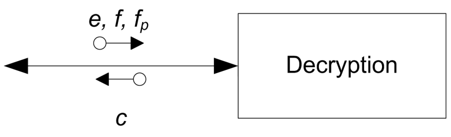

Figure 4.12: Decryption

4.1.3.1 Star Multiplication modulo p

Star Multiplication modulo p receives polynomials $pol1$ and $pol2$ and outputs in $mulPol$ as it can be observed in 4.13. *Star Multiplication modulo p* ensures the product modulo p of both polynomials in $mulPol$ as described in formula 2.3, $mulPol = pol1 * pol2 \pmod{p}$.

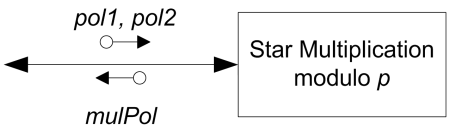

Figure 4.13: Star Multiplication modulo p

The *Decryption* process is summarized in the figure 4.14.

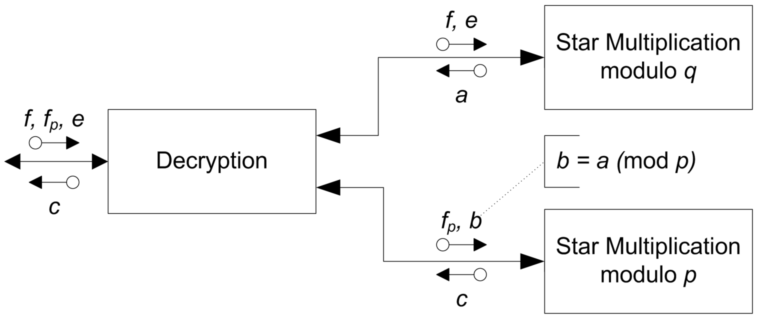
f, f_p, e
c
Decryption
f, e
a
Star Multiplication modulo q
$b = a \pmod{p}$
f_p, b
c
Star Multiplication modulo p

Figure 4.14: Diagram of processes invoked by Decryption

The *Decryption* first computes the star multiplication of the private key f by the encrypted message e reducing modulo q the coefficients. This product is calculated with the *Star Multiplication modulo q* and stored in the polynomial a, having $a = f * e \pmod{q}$. The coefficients a_i of a a are then centered in the range $-q/2 \leq a_i < q/2$ to subsequently reduce modulo p the coefficients of a obtaining as a result the polynomial b, where $b = a \pmod{p}$. At the end b is multiplied modulo p by the inverse of f modulo p obtaining the original message m stored in c, having $c = f_p * b \pmod{p} = m$.

4.2 Reference Code

Our first implementation of NTRU PKC has been developed in standard ANSI C, more precisely ISO/IEC 9899:1999 standard [35], on a x86/Linux platform. This implementation has been very useful to later on optimize the code for a constrained device. Actually, this implementation was ported to the ATmega128 [36] microcontroller. This section describes the code functions defined in first place following the code specifications in previous section.

The developed algorithms were mainly extracted from [2] referring to the *Key Generation*, *Encryption* and *Decryption* while the *Inverse modulo p* and *Inverse modulo q* functions have been extracted from [37].

Major changes have been implemented in the code structure and in the code functions which are reflected in chapter 6.

4.2.1 Header File

In the header file we can find all the defined parameters and all the shared function headers. The implementation has been designed for the security parameter values recommended by NTRU Cryptosystems, Inc., presented in table 3.1, where p is chosen to be 3 while N and q parameters are chosen to be 167 and 128 respectively. In order to make the software scalable for the different security levels recommended by NTRU Cryptosystems, Inc., the parameters have been predefined in the header file with the directive $\#define$ provided in C. For NTRU:167:3 we have the following,

```
#define N 167
#define MOD_Q 128
#define MOD_P 3
#define df 61
#define dg 20
#define dr 18
```

where N represents the number of coefficients of the polynomial, which bounds the degree of the polynomial to $N-1$. As an exception the polynomial g used in the inversion functions is the irreducible polynomial of the ring and therefore has a degree N and consequently $N+1$ coefficients. The parameters q and p are defined in the code as MOD_Q and MOD_P. Also the parameters *df*, *dg* and *dr* indicating the number of positive and negative ones for the polynomials f, g and r respectively are predefined and should match the parameters set in table 3.3.

Finally, for a more intelligible code and easy modification, the types *char* and *unsigned char* are defined as *int8_t* and *uint8_t* respectively.

```
#define int8_t signed char
#define uint8_t unsigned char
```

The explanation of the functions proceeds in the following order: *Key Generation*, *Encryption* and *Decryption*.

4.2.2 KeyGeneration

The KeyGeneration function generates the private key set $(f,\ f_p)$ and the public key h. The polynomials are stored in arrays of size N which are initialized to zero. The degree of a polynomial is stored independently on a separate variable of the type *uint8_t*. The function header is:

void KeyGeneration(int8_t *f, uint8_t *fdeg, int8_t *f_p, uint8_t *f_pdeg, int8_t *h, uint8_t *hdeg).

4.2.2.1 RandPol

The first step is to generate f randomly from a ring of truncated polynomials R. For this matter we developed the function *RandPol*, equivalent to the *Random Polynomial* process presented in the Code Specifications section 4.1.. The header of *RandPol* is:

uint8_t RandPol (int8_t *r, uint8_t num_pos_1, uint8_t num_neg_1).

RandPol receives the pointer to the array r and two variables. Since the function is called to generate f, g and r it is also necessary to pass the spaces d_i to indicate the number of positive and negative ones. Note this function generates the so-called "small" polynomials with coefficient values $\{-1,\ 0,\ 1\}$. At last but not least, *RandPol* returns the degree of r.

4.2.2.2 Rand

The pseudo random core engine has been separated into the function *Rand*. This allows to improve the pseudo random generation independently without modifying *RandPol*. The main purpose of this function is to generate random numbers in the range 0 to $N-1$. The generated random numbers indicate the positions where the positive and negative ones are placed.

uint_8 rand ().

With the set of these two functions the program is able to generate a random polynomial for an specific ring. In the *Key Generation* process *RandPol* is called to generate the private key f and the polynomial g, required to calculate the public key h. *RandPol* is also called during *Encryption* to generate the blinding polynomial r.

4.2.2.3 InverseGFp

The header for *InverseGFp* is:

void InverseGFp (int8_t *f_p, uint8_t *f_pdeg, int8_t *f, uint8_t fdeg).

The parameters are passed by reference to avoid unnecessary memory usage and also hence they are necessary in contiguous parts of the cryptosystem.

4.2.2.4 Sum2PolP and Sub2PolP

The addition and subtraction of polynomials modulo p are both necessary for the computation of the inverse modulo p. Heathers are defined as:

void Sum2PolP(int8_t *pol1, uint8_t *pol1deg, int8_t *pol2, uint8_t pol2deg)
void Sub2PolP(int8_t *pol1, uint8_t *pol1deg, int8_t *pol2, uint8_t pol2deg).

These two functions implement the addition and subtraction respectively. The value of p is not required since it has been previously defined in *MOD_P* with the *#define* statement, so before compilation time this string is replaced for the defined value for *MOD_P*. Also the degree *pol1deg* is passed by reference to recycle *pol1* since the addition result is stored in this polynomial in order to save memory.

4.2.2.5 InverseGFq

During the *Key Generation* the inverse of f modulo q, referred to as *f_q* in the code, is also required (the polynomial f_q is necessary together with g to calculate the public key h). The header of the function *InverseGFq* is very similar to the header of the function *InverseGFp*:

void InverseGFq(int8_t *f_q, uint8_t *f_qdeg, int8_t *f, uint8_t fdeg).

Parameters are passed exactly the same way as before obtaining the inverse of f modulo q in the array *f_q*.
The main difference respect to *InverseGFp*, rather than the modulo reduction, are the final steps. Computing the inverse modulo a power of a prime is done in two parts. First the inverse modulo the prime (in this case modulo 2) is computed. Then the inverse modulo the prime is converted to the desired modulo power of the prime (modulo q or 2^7 for $q = 128$). In chapter 6 is explained how finally this function has been split in half in order to compute the inverse modulo two with binary coefficients saving computational and memory resources.

4.2.2.6 Sum2PolQ

Sum2PolQ receives *pol1*, *pol2* and *pol1deg* by reference whereas *pol2deg* and *mod* are passed by parameter as defined in the header:

void Sum2PolQ (int8_t *pol1, uint8_t *pol1deg, int8_t *pol2, uint8_t pol2deg, uint8_t mod).

The modulo value is passed by parameter in the variable *mod* since *Sum2PolQ* is recursively called for different values when executing *InverseGFq*. This is due in order to reuse the function in the conversion from an inverse modulo a prime to the inverse modulo a power of a prime. The addition of the two polynomials is stored in *pol1* and the degree of resulting polynomial in $pol1deg$.

4.2.2.7 PolMulQ

In order to compute the public key h we need to compute the star multiplication of polynomials f_q and g. The star multiplication modulo q is also necessary in our first implementation during

Encryption and *Decryption*. For this function three array pointers and a variable are passed,

uint8_t PolMulQ (int8_t *mul_pol, int8_t *pol_p, int8_t *pol2, uint8_t mod).

Note that the variable *mod* holds the value of the modulus. This is required as in *Sum2PolQ* in order to reuse this function during *KeyGeneration* to obtain the inverse of f modulo q. During the inversion process when q is a power of a prime is first computed the inverse of f modulo the prime to later obtain the inverse modulo q. This last step involves having to compute the multiplication modulo based on different powers of two when $q = 128$. Hence the need to pass the variable *mod* despite we have set the value of q in the file headers. For more information referring to the inversion please refer to [37].

For a better understanding of the KeyGeneration function figure 4.15 shows the hierarchical structure of the invocated functions.

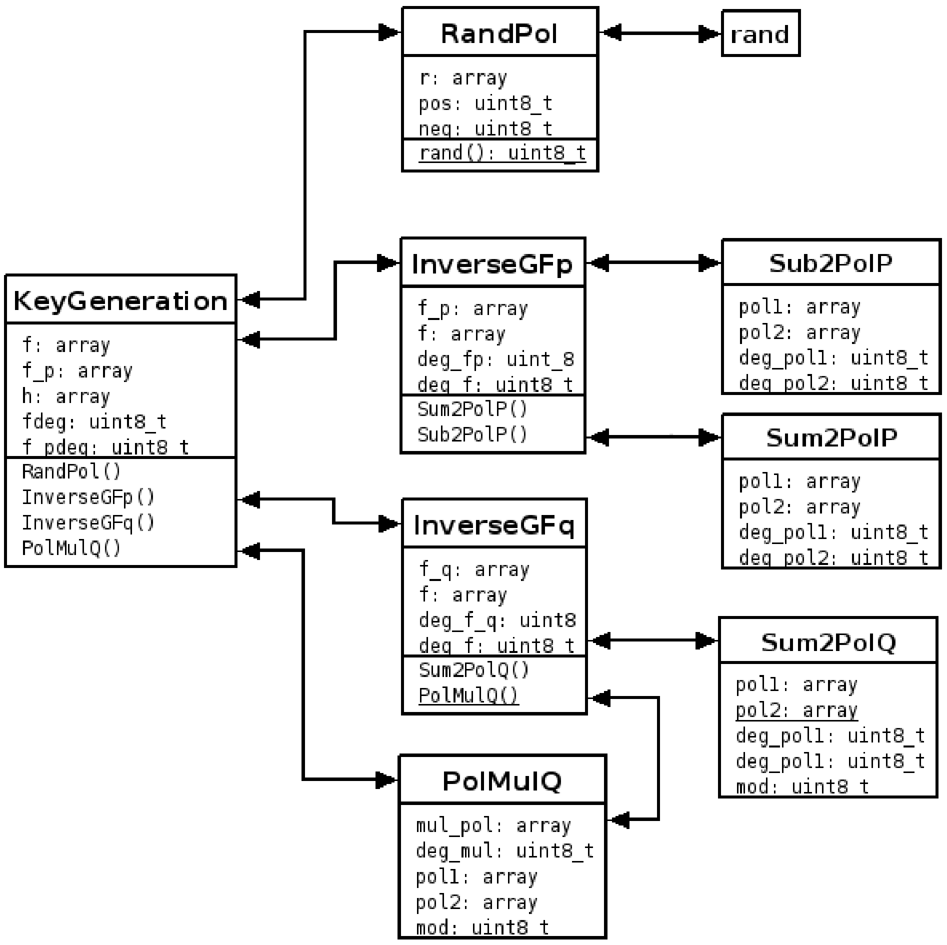

Figure 4.15: Key Generation functions

4.2.3 Encryption

For the *Encryption* function we pass the polynomial e, the degree of e, the clear message m and the public key h as shown in the header:

void Encryption(int8_t *e, uint8_t *edeg, int8_t *h, int8_t *m).

To generate the blinding message r the function *RandPol* is used as described in section 4.2.2.1. The polynomial r is declared locally because is just used in the encryption process. Then, once r is obtained it can be computed the multiplication by h modulo q with the function *PolMulQ* presented in section 4.2.2.7. The result is stored in the array e. At last m is added to e with the function *Sum2PolQ*, introduced in section 4.2.2.6, obtaining the final encrypted message in the array e.

The diagram of the *Encryption* functions invocations is shown in figure 4.16.

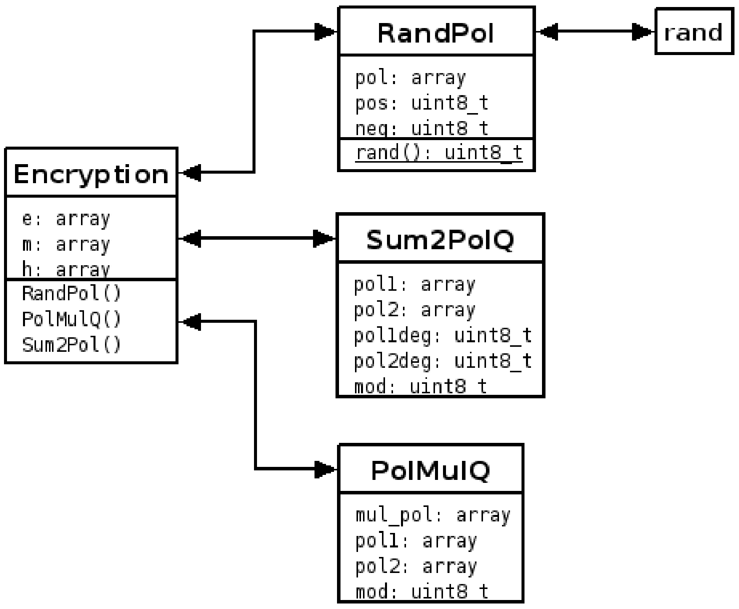

Figure 4.16: Diagram of functions invoked by Encryption

4.2.4 Decryption

The *Decryption* function decrypts the encrypted message e into the array c, both in polynomial form. For this reason the *Decryption* function requires us to pass the encrypted message e, the private keys f and f_p and the array where to store the decrypted message, c.

void Decryption(int8_t *e, uint8_t *edeg, int8_t *f, int8_t *c).

First step is to multiply one of the private keys, f, by the encrypted message e, see section 3.1.4. This is stored in the array a created locally and it is computed in the function PolMulQ presented in section 4.2.2.7. Then the a coefficients are centered lying between $-q/2$ and $q/2$

and reduced modulo p. To implement this, a loop goes through all coefficients centering the coefficients and reducing them to minus one, zero or one.

4.2.4.1 PolMulP

Last step of *Decryption* is the product modulo p of a by the private key, f_p to obtain the original message m. For this last step a new function has been coded to implement efficiently the modulo p star multiplication. The header of this function is:

uint8_t PolMulP (int8_t *mul_pol, int8_t polynomial *pol_p, int8_t polynomial *pol2).

Three arrays are passed as in PolMulQ and the degree is returned. Note that arrays are alway passed by reference in C.

The diagram of the *Decryption* function invocations is shown in figure 4.17.

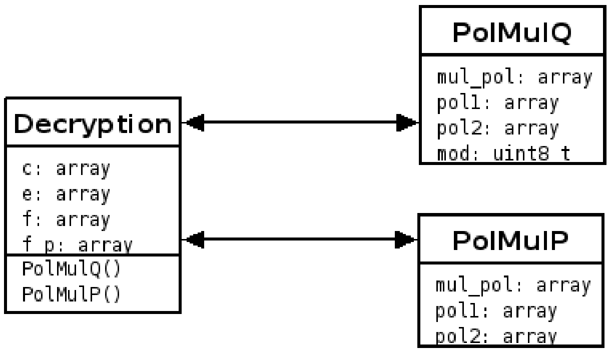

Figure 4.17: Diagram of functions invoked by Decryption

Chapter 5

Hardware

Embedded devices are designed to do some specific task. The memory and computational resources on an embedded device are much more limited compared to a workstation. This chapter intends to overview the most relevant aspects of the device where the NTRU PKC has been implemented in the development of this thesis.
The device in question is the ATmega163 [38] microcontroller. The ATmega163 is a low-power CMOS 8-bit microcontroller running up to 8 MHz. based on the AVR architecture. It can execute powerful instructions in a single clock cycle. The AVR core combines a rich instruction set [39] with 32 general purpose working registers. All the 32 registers are directly connected to the Arithmetic Logic Unit (ALU), allowing two independent registers to be accessed in one single instruction executed in one clock cycle.
The AVR uses a Harvard architecture, with separate memories and buses for program and data. The Program memory is executed with a two stage pipeline. While one instruction is being executed, the next instruction is pre-fetched from the Program memory. This architecture enables instructions to be executed in every clock cycle. The Program memory is In-System Re-programmable Flash memory. It also has 1 KB of SRAM and 512 bytes of EEPROM memory.
Specifically, the ATmega163 has 1280 Data Memory locations as shown in figure 5.1 from [38].

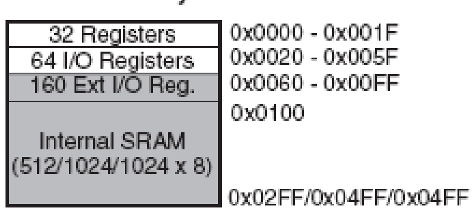

Figure 5.1: Data Memory

The firsts 32 locations are for the Register file where the register addresses are mapped, the next 64 locations are for the standard I/O memory, there are also 160 locations of Extended

I/O memory, and finally 1024 location addresses for the internal SRAM data.

The memory access time are 2 clock (clk) cycles as it can be observed in figure 5.2 from [38].

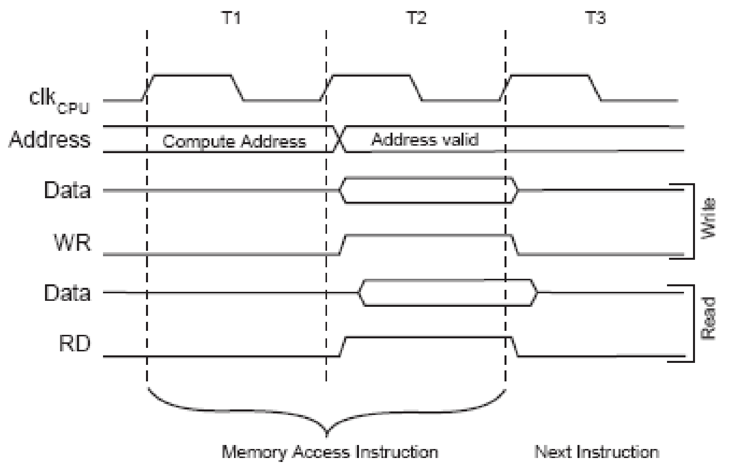

Figure 5.2: Data SDRAM Access Cycles

The ATmega163 contains 512 bytes of EEPROM memory. It is organized as a separate data space in which single bytes can be read and written. The write access time for the EEPROM is 3.3 ms considering an 8 MHz clock is used; more precisely it takes 26.368 CPU cycles. This performance is much slower than the SRAM memory.

The ATmega163 also contains 16K bytes On-chip In-System Reprogrammable Flash memory for program storage. Timing diagrams for instruction fetch and execution are presented in figure 5.3 from [38].

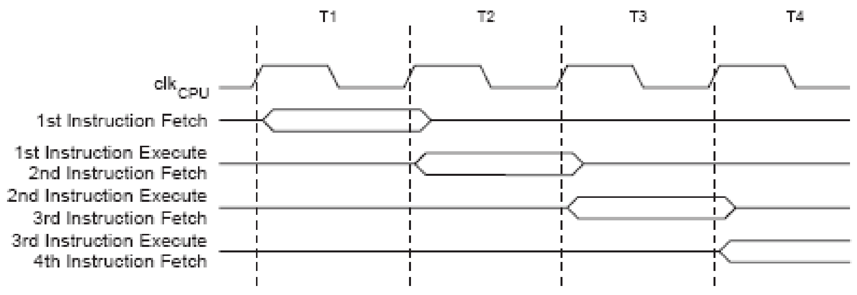

Figure 5.3: The Parallel Instruction Fetches and Instruction Execution

Although program memory is much more than enough to store the NTRU PKC, the SRAM is quite small for the data required to manage. For more information about this device refer to [38].

Chapter 6

Optimizations

Our first implementation of NTRUEncrypt, introduced in chapter 4, for the parameter set NTRU:167:3 took around 176 ms for encryption and 405 ms for decryption on the ATmega128 microcontroller, refer to section 4.2. This first version was implemented on the ATmega128 microcontroller due to design restrictions. Specifically, the data memory necessary in the first version exceeded the hardware specifications of the microcontroller ATmega163.
We chose this option because the ATmega163 and the ATmega128 microcontrollers have very similar RISC architectures along with the instruction set. This made the transition easier to finally implement NTRUEncrypt on the ATmega163.
The source code size of the first version was about 5746 bytes, but the biggest problem was the memory RAM used, around 1 KB.
From this starting point, we adapted and improved the source code to consume fewer resources and be able to run it on the ATmega163 efficiently. The following sections refer to optimizations in order to save memory resources, algorithmic optimizations and the proper usage of the arithmetic operators to make our NTRU PKC implementation even faster.

6.1 Memory Optimizations

The memory in a constrained device is typically very limited. The software developed in this thesis has been customized and optimized for the ATmega163 microcontroller. The ATmega163 is an 8-bit microcontroller running up to 8 MHz with 1024 bytes of SRAM, 512 bytes of EEPROM and 16 KBytes of flash memory. These specifications are extremely reduced, therefore an extra effort has been done to port the NTRU PKC to its minimum expression in terms of SRAM consumption.

6.1.1 Variable Types

The AVR-GCC compiler for the ATmega163 microcontroller the default size of an integer is 16 bits. For an improved performance all variables have been defined as *uint8_t* or *int8_t* whenever possible. Both types are 8 bit size and store unsigned or signed values respectively. These types have been defined on the header file.

```
#define int8_t signed char
#define uint8_t unsigned char
```

As a result of using *char* as variable type instead of *integer*, besides the memory data reduction, the application works faster since the ATmega163 has an 8-bit data bus.

6.1.2 Parameter Passing

To avoid unnecessary memory usage the majority of the parameters have been passed by reference as it can be observed in the function headers presented in section 4.2, except for the variables of eight bit size. This decision is due to the fact memory addresses in an 8 bit CPU are 8 bit large. This means that passing a pointer occupies the same as a variable of such a type. Having in mind the code might be executed or ported to a higher bit CPU, a 32-bit for example, then is even more costly to pass the address than the value itself.

6.1.3 Storing Random Polynomials

Random polynomials used in NTRU PKC have ternary coefficients in our implementation. Since the values of the coefficients are {1, 0, -1} (binary polynomials can also be used but the system turns out to be more vulnerable to lattice attacks) we could store them with two binary digits as {01, 00, 11}, instead of using eight bits for each coefficient. Using two bit coefficients instead of eight bit coefficients reduces the size of the polynomial a 75%. However, we have used a different technique to store the coefficients for the polynomials g and r. The technique is to store the **position** of the coefficients which differ from zero (one or minus one in this case). Imagine we have a random polynomial g of degree nine with the following coefficients, $g = [1, 0, \mathbf{-1}, 0, 0, \mathbf{-1}, 1, 0, 1, \mathbf{-1}]$. Storing the positions we obtain polynomial $g = [0, 6, 8, \mathbf{2}, \mathbf{5}, \mathbf{9}]$. To allow efficient processing of the polynomial the positive ones are stored in the first half of the array while the negative ones are stored in the second half. When working with polynomials of $N = 167$ or $N = 251$ coefficients we need to use at least 8 bits to store each position. For the spaces defined by dg and dr in general stands that the percentage of zero terms is around a 75%. Consequently storing the positions provides a similar reduction in memory terms as storing using two bits per coefficent. In addition, we benefit of a faster multiplication for both polynomials, see section 6.2.4.2. This is reflected in the code with a new *RandPol* function. For scalability the generation is done with a temporal array of N positions which afterwards is stored as described in a new array of $2 \cdot d$ positions while the temporal polynomial is wiped out.

```
void RandPol (uint8_t *r, uint8_t d )
```

Where d is the parameter that defines the space of the polynomials since g and r have even amounts of positive and negative ones.

6.1.4 Storing the Private Key

The way of storing the private key f is stored in a similar way as described for the random polynomials g and r in the previous section 6.1.3. However, when applying the form of $f = 1 + pF$ some singularities are taken into account, please refer to section 6.2.1 for more information regarding the form of f.
Since F has integer coefficients between one and minus one, the form of $f = 1 + pF$ is pseudo deterministic in the sense that assures us all coefficients, excluding the first one, acquire the values $-p$, zero or p. More exactly we have three possible scenarios since $f(0)$ can assume the values $\{1 - p, 1 \text{ or}, 1 + p\}$. For $f(0) = 1 - p$ we might have $df - 1$ coefficients with the value $+p$ and $df - 2$ coefficients with $-p$. For $f(0) = 1$ we obtain $df - 1$ coefficients with value p and $df - 1$ coefficients with value $-p$. And for $f(0) = 1 + p$ results in $df - 2$ coefficients p and $df - 1$ coefficients with value $-p$. Being concerned about these three possibilities we store the polynomial f assigning the coefficient value to the first array position. So for $p = 3$ we assign the values $f(0) = -2$, $f(0) = 1$ or $f(0) = 4$ depending on the random generation. For the rest of the array we store the coefficients as we did with the random polynomials g and r in previous section 6.1.3. Let us see a practical example. If $p = 3$ we may have:

1. $f = [1, 0, -3, \mathbf{3}, 0, \mathbf{3}, 0, 0, \mathbf{3}, -3, -3] = [1, \mathbf{3}, \mathbf{5}, \mathbf{8}, 2, 9, 10]$.
2. $f = [4, \mathbf{3}, -3, 0, 0, -3, 0, \mathbf{3}, 0, 0, -3] = [4, \mathbf{1}, \mathbf{7}, 4, 5, 10]$.
3. $f = [-2, 0, \mathbf{3}, 0, -3, 0, \mathbf{3}, -3, 0, \mathbf{3}, 0] = [-2, \mathbf{2}, \mathbf{6}, \mathbf{9}, 4, 7]$.

Note that in the first case we have the same amount of positive and negative threes but in the last two cases we have one extra zero term and one missing negative or positive three depending on $f(0)$. For this reason we coded a dynamic way of storing f based on the value of f_0. The new function *FGen* is coded with the next header:

void FGen(struct polynomial *f, uint8_t *fdeg)

Where f is the polynomial and $fdeg$ the polynomial degree. This storage modification not only reduces the array where we store f to have a maximum size of $2 \cdot df - 1$ bytes but also is fundamental to save computational resources when computing the optimized multiplication for f presented in section 6.2.4.3.

6.1.5 Binary Compression

During the key generation the inverse of f modulo q is computed. This is actually done in two main steps. First we compute the inverse modulo two which can be achieved in a very fast way through the Extended Euclidean Algorithm [37] to later compute the inverse of f modulo q through a method based on Newton iteration. This last step is the most complex part during the key generation since involve a loop with two polynomial multiplications. Despite this fact, we get a very simple polynomial inverse function. Working with coefficients reduced modulo two means we can work with binary coefficients which only require one bit each. For this reason after generating the private key f we convert f into modulo two and store it in the bit array $fbin$ which occupies ceil($N/8$) bytes of data. After obtaining $fbin$ the original f is stored, in the form introduced in section 6.1.4, in the EEPROM memory and the memory space occupied by f in the SRAM is freed.

6.1.6 Inverse modulo two

The *InverseModTwo* function is computed using the Extended Euclidean Algorithm. Algorithm 1 shows the pseudo code of the *InverseModTwo* function.

Algorithm 1 Inverse of a(x) mod two

```
Require: $a(X) = f(x),\ g(X) = X^N - 1,\ b(X) = 1,\ c(X) = 0,\ k = 0;$
Ensure: The inverse of $a(X)$ stored in $b(X)$.
 1: loop:
 2: while $a_0 = 0$ do
 3:    for all $i$ such that $0 \leq i < floor(N/8)$ do
 4:       $a_i = a_i >> 1$ $\{a = a/X\}$
 5:       $a_i = a_i + (a_{i+1}\&1) << 7$
 6:    end for
 7:    $a_{N/8} = a_{floor(N/8)} >> 1$
 8:    for all $i$ such that $0 < i \leq floor(N/8)$ do
 9:       $c_i = c_i << 1$ $\{c = c * X\}$
10:       $c_i = c_i + c_{i-1} >> 7$
11:    end for
12:    $c_0 = c_0 << 1$
13:    $k++$
14: end while
15: if $a = 1$ then
16:    return k
17: end if
18: if $deg(a) < deg(g)$ then
19:    exchange $a$ and $g$ and $b$ and $c$
20: end if
21: SumPolBin($a, m, deg(a), deg(b)$)
22: SumPolBin($b, c, deg(b), deg(c)$)
23: go to loop
```

Exchanging a for g and b for c in line 19 it is done by switching pointers to avoid copying each array position over a loop. Implementing the division and multiplication of X in the while loop in line 3 through 11 is done by making a right bit shift and a left bit shift respectively. But most remarkable for memory savings is the usage of bit arrays to store the polynomials f, a, b and g as described in section 6.1.3. This has lead to code a function which adds two polynomials with coefficients of one bit.

6.1.6.1 Binary Addition

The function *SumPolBin* calculates the addition of two polynomials reducing the coefficients modulo two. This function has been developed for the addition operations during the computation of the *InverseModTwo.* Since coefficients can be represented with one bit, consequently an array position stores eight coefficients. Therefore, all polynomials involved in the Algorithm 1 and Algorithm 2 have a size of ceil($N/8$). This not only reduces the memory data, but the

complexity of the operations thanks to the *SumpolBin* function shown in Algorithm 2. Using the *XOR* operator enables us to add in one instruction eight coefficients, reducing approximately by eight the complexity of the polynomial addition. More precisely, Algorithm 2 shows we iterate ceil($deg(a)/8$) times to add two polynomials.
For example, if we have the polynomials $a(X)$ and $b(X)$ where,

$$a(X) = X^7 + X^5 + X^4 + X^3 + 1$$

and,

$$b(X) = X^6 + X^5 + X^3 + X^2 + 1 \ .$$

Then we store the eight binary coefficients of $a(X)$ in one byte array position in decimal, $a[0] = 185 \Rightarrow 10111001$. We do the same for $b(X)$ storing $b[0] = 109 \Rightarrow 01101101$. Since performing the addition with the "+" operator we obtain bit carries in the addition $a[0] + b[0]$, we use the XOR operator instead.

$$a[0] \text{ XOR } b[0] = 185 \text{ XOR } 109 = 212 \Rightarrow 10111001 \text{ XOR } 01101101 = 11010100 \ .$$

Binary masks can also be used to calculate bit by bit additions. Instead we decided to make a byte by byte addition and using the *XOR* operator we obtain the desired result.

Algorithm 2 SumPolBin

```
Require: $a(X)$, $b(X)$, $deg(a)$, $deg(b)$;
Ensure: The sum of $a(X)$ and $b(X)$ modulo two is stored in $a(X)$.
1: if $deg(a) < deg(b)$ then
2:     $deg(a) = deg(b)$
3: end if
4: for all $i$ such that $0 \le i < ceil(deg(a)/8)$ do
5:     $a_i = a_i$ XOR $b_i$
6: end for
```

In this case to update the degree the cost associated is the same as the degree search in *SumPolQ* function but with some extra code lines, see section 6.2.3. The loop goes form floor($N/8$) to zero but some conditions are needed to differentiate the coefficients values.

6.2 Computational Optimizations

6.2.1 The Form of f

In section 3.1.5 we explained how NTRU public-key algorithm works. Now we explain an optimization for the private key f [40] recommended by NTRU Cryptosystems, Inc. for commercial applications. NTRU PKC requires f to satisfy the following properties:

1. f is invertible mod p,

2. f is invertible mod q,
3. and f is small.

If f is taken with the form

$$f = 1 + pF\,,$$

where F is a “small” polynomial, then

$$f = 1 + pF \equiv 1 \pmod{p}\,.$$

Therefore,

$$f^{-1} \pmod{p} \equiv 1 \pmod{p}\,.$$

This algorithmic optimization has two major consequences. First it eliminates the computation of the inverse of f modulo p in the key generation process and second it is no longer necessary to store f_p since we know is 1. Furthermore it eliminates the last step of decryption since f is canceled out when reducing modulo p:

$$a = f * e \pmod{q} = pr * g + f * m \pmod{q} = pr * g + (1 + pF) * m \pmod{q}\,.$$

$$c = a \pmod{p} = pr * g + (1 + pF) * m \pmod{p} = (1) * m \pmod{p} = m \pmod{p}\,.$$

When reducing modulo p the product $pr * g$ equals zero but also pF, obtaining directly the plaintext of the message m without the last multiplication by f_p. This lets decryption timings approach to encryption timings since only one star multiplication is processed, refer to version 3 results in table 6.1. This fact reduces the memory data required because f_p is not stored, but also reduces the memory data for decryption because computation of $f_p * e \pmod{p}$ is no longer necessary which permits us calculate all operations in one polynomial, c. Moreover, code data is also reduced since functions *InverseGFp, Sum2PolP, Sub2PolP* and *MulPolP* are not required anymore.
Note that for the generation of $f = 1 + pF$ is necessary a new random function just for f, already introduced in section 6.1.4. Since f now equals $1 + pF$, *df* is used to define the number of ones and also the number of negative ones of F, giving as a result $F(1) = 0$ and $f(1) = 1$. Where $f(1)$ represents the sum of all coefficients of f in order to “assure” is invertible, see section 2.5 for more details. As a result the parameter *df* is not longer needed in the header of the function since is defined as a constant.

6.2.2 Modulo Operation

ANSI C provides the operator “%” to calculate the modulus. The syntax is *value%mod*. This operation is very costly since it basically divides the value by the modulo and returns the remainder. For this reason the modulo operator “%” is avoided whenever possible.

We have taken the parameter q to be a power of two when p is a prime number greater than 2 in order to satisfy the axiom that q and p must be coprime. Particularly for integers, the modulus powers of two can be calculated in a very simple way in binary. If we compute the values in binary and throw away the bits equal or greater than the modulo value we obtain the modulo operation. With an *AND* operator we can accomplish the modulo computation. For example:

$$35 \equiv 3 \pmod{32} \text{ expressed in binary is } 1\mathbf{00011} \equiv \mathbf{00011}\ .$$

As observed in this example the most significant bit is thrown away considering its value equals to thirty two. As said, this applies to any modulo power of two.

$$325 \equiv 5 \pmod{64} \Rightarrow 101\mathbf{000101} \equiv \mathbf{000101}\ .$$

In this last example we are throwing the first three more significant bits since their values are greater or equal to the modulo value. We can say then that reducing modulus of a power of two can be achieved computing the next logic operation: element *AND* (modulo-1).

$$325 \equiv 5 \pmod{64} \Rightarrow 325\,\mathrm{AND}\,(64-1) = 5 \Rightarrow 101\mathbf{000101}\,\&\,111111 = \mathbf{000101}\ .$$

So for all operations involving q instead of using the "%" operator we use the *AND* operator represented in C by the operator "&".

6.2.3 Addition Modulo q Operation

The ATmega163 microcontroller is able to perform the addition of two bytes in one clock cycle. The addition of polynomials algorithm has been optimized in two ways as observed in figure 3. First the degree is used to reduce the sum of coefficients, the addition of two polynomials only requires as many additions as the greater degree of both polynomials, often smaller than N. Second, the modulo operation is done as described in previous section 6.2.2.
Remark the modulo operator as the division is not natively present in the assembler instructions of the ATmega163. To accomplish a division operation six assembler instructions are required. For this reason in the addition we use the "&" operator instead of the "%" operator.

Algorithm 3 SumPolModQ

```
Require: $a(X)$, $b(X)$, $deg(a)$, $deg(b)$;
Ensure: The sum of $a(X)$ and $b(X)$ modulo $q$ is stored in $a(X)$.
1: mod=mod-1
2: if $deg(a) < deg(b)$ then
3:     $deg(a) = deg(b)$
4: end if
5: for all $i$ such that $0 \leq i < deg(a)$ do
6:     $a_i = (a_i + b_i)\&mod$
7: end for
```

On the other hand, after the complete operation between two polynomials the degree of the resulting polynomial is updated. For this purpose a loop checks the positions of the vector to determine the degree.

In figure 4 we can see that to reduce this search the loop sequence has been inverted since most polynomials have a degree close to N.

Algorithm 4 Degree Search

```
Require: $r(X)$, $deg(r)$;
Ensure: $deg(r)$ stores the degree of the polynomial $r$.
1: for $i = N$ to 0 do
2:     if $r(i) \neq 0$ then
3:         $deg(r) = i$
4:         break
5:     end if
6: end for
```

This loop evaluates approximately the positions between N and the degree value of the greatest polynomial involved in the operation. It might be possible the addition of two polynomials of the same degree return zero for the highest coefficient. In this case the loop searches until reaches a coefficient with a non zero value.

6.2.4 Multiplication

The Multiplication of polynomials is a key factor during the different processes of the NTRU PKC. Multiplication is used in the key generation when calculating the Newton iteration method to obtain the inverse (from binary to modulo q form) and in the calculus of the public key h. But also it is the operation most costly during *Encryption* and *Decryption*. Improving the multiplication turned out to be one of the premises of this thesis.

NTRU Cryptosystems, Inc. proposed two solutions for the multiplication of polynomials. The technical note High-Speed Multiplication of Truncated Polynomials [41] suggests an algorithm based on Karatsuba multiplication done recursively for polynomials of arbitrary coefficients.

This algorithm saves coefficients multiplications at the cost of extra additions compared to the schoolbook multiplication. But still there are multiplications on the calculus.

The second option that NTRU Cryptosystems, Inc. proposed for embedded devices is the usage of the Fast Convolution Algorithm [40], a fast algorithm for the multiplications where the private key f with the form $1 + pF$ is involved.

6.2.4.1 Fast Convolution Algorithm

Assuming the polynomial f has binary coefficients, NTRU Cryptosystems, Inc. has developed an algorithm to compute the multiplication of a binary polynomial by another polynomial. Scanning the b array permits calculate only the inner product terms which may be different from zero.

The algorithm begins by zero-initializing an array of coefficients which holds the result $c(X) = f_i(X) \cdot a(X)$. For each entry of the f array, the algorithm calculates the N inner product terms corresponding to a non-zero coefficient in $f_i(X)$. Since $f_i(X)$ is binary, each non-zero inner product term is simply a coefficient of $a(X)$. These terms are individually accumulated in their corresponding location in the c array. Repeating this process for all non-zero coefficients computes $f_i(X) * a(X)$ at the cost of $d_i N$ additions of $\log_2 q$ bit numbers. The Algorithm 5 presents the pseudo code.

Algorithm 5 Fast Convolution Multiplication

```
Require: $b$ an array of $d_1 + d_2 + d_3$ nonzero coefficient locations representing the polynomial
    $f(X) = 1 + p * (f_1(X) * f_2(X) + f_3(X))$, $a$ the array $a(X) = \sum a_i, N$ the number of
    coefficients in $f(X)$; $a(X)$.
Ensure: $c$ is the array where $c(X) = f(X) * a(X)$
 1: for $j = 0$ to $d_1 - 1$ do
 2:     for $k = 0$ to $N - 1$ do
 3:         $t_{k+b_j} = t_{k+b_j} + a_k$ $\{t(X) = a(X) f_1(X)\}$
 4:     end for
 5: end for
 6: for $j = d_1$ to $d_2 - 1$ do
 7:     for $k = 0$ to $N - 1$ do
 8:         $c_{k+b_j} = c_{k+b_j} + t_k$ $\{c(X) = t(X) * f_2(X) = a(X) * f_1(X) * f_2(X)\}$
 9:     end for
10: end for
11: for $k = 0$ to $N$ do
12:     $t_k = 0$
13: end for
14: for $j = d_2 + 1$ to $d_3 - 1$ do
15:     for $k = 0$ to $N - 1$ do
16:         $t_{k+b_j} = t_k + b_j + a_k$ $\{t(X) = f_3(X) * a(X)\}$
17:     end for
18: end for
19: for $k = 1$ to $N - 1$ do
20:     $c_k = c_k + t_k \pmod q$ $\{c(X) = c_k + t_k \mod N = f_3(X) * a(X) + f_1(X) * f_2(X) * a(X)\}$
21: end for
```

6.2.4.2 Optimized Multiplication Algorithm

Our proposed algorithm has been developed for the computation of the star multiplication when the random polynomials r or g are present. Storing the polynomials as described in section 6.1.3 permits the algorithm only compute the coefficients that differ from zero. Moreover, the algorithm does not computes the product operation since coefficients are ternary. The addition or subtraction is computed for coefficients with value one or minus one respectively.

These two factors speed the computation. The product of two coefficients takes two clock cycles while addition or subtraction takes only one clock cycle for the AVR ATmega microcontrollers family. Finally having in mind the polynomials involved in the NTRU PKC we observe the random polynomials g and r have around a 70% of zero coefficients. Since the zero coefficients are not even stored and therefore computed, the clock cycles are also reduced around this percentage. How it works? Let's take a look to the pseudo code in the Algororithm 6.

Algorithm 6 Optimized Multiplication

```
Require: $a(X)$, $b(X)$, $c(X) = 0$, $sizepos$;
Ensure: polynomial $c(X) = a(X) * b(X)$.
 1: $sizeneg = 2 * sizepos - 1$
 2: for $k = 0$ to $N - 1$ do
 3:   for $i = sizepos - 1$ to $0$ do
 4:     $y = k - a_i$
 5:     if $y < 0$ then
 6:       $y = y + N$
 7:     end if
 8:     $c_k = c_k - b_y$
 9:   end for
10:   for $i = sizeneg$ to $sizepos$ do
11:     $y = k - a_i$
12:     if $y < 0$ then
13:       $y = y + N$
14:     end if
15:     $c_k = c_k - b_y$
16:   end for
17:   $c_k = c_k \& (q - 1)$
18: end for
```

The requirements of this algorithm are that the polynomial $a(X)$ must be stored as described in section 6.1.3. The polynomial, $b(X)$, is expected to have coefficients reduced modulo q. The restrictions for the coefficients of the polynomial $b(X)$ are given by the variable type used in the array declaration where is stored the polynomial.
The main calculus is done as usual in the truncated polynomial product going through all the N coefficients of the resulting polynomial c. The only difference is that the Algorithm 6 just adds or subtracts b_k instead of multiplying $a_i \cdot b_y$. This is possible since a_i is one or minus one, $c_k = c_k + (1)b_y$ or $c_k = c_k + (-1)b_y$.
The first nested loop computes the product of the coefficients of $a_i \cdot b_y$ where a_i equals to one. The next nested loop computes the terms of b_y that are multiplied by the negatives ones of a_i. Finally the resulting c_k coefficient is reduced modulo q with an *AND* operation, c_k AND $q - 1$. Note that instead of going through all the coefficients of a and b to do the product operation it just goes over the space d, represented by the variable *sizepos* for the positive coefficients of a and $sizeneg = 2 \cdot d$ for the negative coefficients. This process is done N times for all the c_k coefficients. So at the end, assuming we have a polynomial of N coefficients and a space defined by d, the potential cost of the Algorithm 6 is $N \cdot 4 \cdot d$ since there is an extra addition to calculate the index y. This algorithm is a fast solution for the random polynomials r and g with ternary coefficients, also denominated as small polynomials.

6.2.4.3 Optimized Multiplication Algorithm for $f = 1 + pF$

To compute the star multiplication with $f = 1 + pF$ we can use a similar algorithm than with ternary polynomials. In section 6.1.4 we introduced the way of storing f which is fundamental for the computation. First step is to make sure we compute the multiplication operation only the essential number of times. When computing $e * f$ $(e + pF * e)$ we should multiply by p no more than N times. As an example if we have the polynomials $f = [1, 0, -3, 3, 0, 3, -3]$ and $r = [5, 16, 9, 32, 29, 18, 1]$ and we want to compute $c = f * r$, the first coefficient of c results in,

$$c_0 = 1{*}5{+}0{*}1{+}(-3{*}18){+}3{*}29{+}0{*}32{+}3{*}9{+}(-3{*}16) = 5{+}(-3{*}18){+}3{*}29{+}3{*}9{+}(-3{*}16)\ .$$

The previous Algorithm 6 eliminates the zero-term computations which is enough with ternary coefficients. But with $f = 1 + pF$ we want to reduce multiplication of p over all the coefficients so we can write,

$$c_0 = 1 * 5 + 3 * (-18 + 29 + 9 - 16)\ .$$

As it can be observed we can reduce the number of times we multiply p in one coefficient to one. For this reason we compute the coefficients like if f had only values between $\{-1, 0, 1\}$ and at the end we multiply c_k by p and add the corresponding coefficient of b_k by f_0. The pseudo code is shown in the Algorithm 7.

This multiplication reduces the cost to $2 \cdot N$ multiplications, $2 \cdot (N - 1) \cdot df$ additions and N *AND* operations.

6.3 Evaluation

The major optimizations applied to the NTRU PKC have been implemented in different code versions, making it easier to compare empirically the CPU and memory optimizations. The optimizations are structured as follows:

- Version 1: Version with classical setup and 8 bit variables.
- Version 2: Implements the reduction modulo q operation using the "&" operator.
- Version 3: Takes f with the form of $f = 1 + pF$ together with version 2 optimizations.
- Version 4: Implements the *Optimized multiplication*, the new *RandPol* function plus version 3 optimizations.
- Version 5: New f storage, the *Optimized f multiplication* together with version 4 optimizations.

The code has been tested in the AVR Studio simulator for the ATmega128 and ATmega163 microcontrollers running at 4 MHz.

6.3.1 ATmega128

The developed software was first implemented on the ATmega128 due to its larger memory resources which made easy the platform portation. The table 6.1 shows the results of the key

Algorithm 7 Optimized Multiplication for f

```
Require: $f(X),\ b(X),\ c(X) = 0$;
Ensure: polynomial $c(X) = a(X) * b(X)$.
 1: if $f_0 = 1$ then
 2:     $sizepos = df - 1$
 3:     $sizeneg = 2 \cdot df - 2$
 4: else if $f_0 = 4$ then
 5:     $sizpos = df - 2$
 6:     $sizeneg = 2 \cdot df - 3$
 7: else
 8:     $sizepos = df - 1$
 9:     $sizeneg = 2 \cdot df - 3$
10: end if
11: for $k = 0$ to $N - 1$ do
12:     $c_k = 0$
13:     for $i = sizepos$ to $i = 1$ do
14:         $y = k - f_i$
15:         if $y < 0$ then
16:             $y = y + N$
17:         end if
18:         $c_k = c_k + b_y$
19:     end for
20:     for $i = sizeneg$ to $i = sizepos + 1$ do
21:         $y = k - f_i$
22:         if $y < 0$ then
23:             $y = y + N$
24:         end if
25:         $c_k = c_k - b_y$
26:     end for
27:     $c_k = c_k \cdot p$
28:     $c_k = c_k + (signed) f_0 \cdot b_k$
29:     $c_k = c_k \& (q - 1)$
30: end for
```

generation, encryption and decryption for the security parameters N=167, q=128 and p=3 and N=251, q=128 and p=3 described in table 3.3 on this device.

To clarify the optimizations impacts the figure 6.1 shows the evolution along the different versions of *Key Generation*.

From figure 6.1 we observe how the operator "&" versus "%" reduces a 25% the timing of key generation for the parameter set NTRU167:3 since is used in several operations. The functions which get more benefit from this operator are the *Inverse Modulo Two* and the Newton iteration method. Furthermore the multiplication of f_q by p and the computation of h involve modulo q reduction.
Applying the form of $f = 1 + pF$ has a major impact for *Key Generation*. This is due to the fact the inverse f_p is no longer required, saving up to a 33% for the parameter set NTRU167:3

Table 6.1: Results on ATMega128 @ 4Mhz.

Version	Security Parameters	Key generation		Encryption		Decryption	
		Code Size	Time	Code Size	Time	Code Size	Time
1	NTRU167:3	3236 Bytes	6.062 s	1382 Bytes	177 ms	934 Bytes	406 ms
2	NTRU167:3	3236 Bytes	4.435 s	1382 Bytes	156 ms	934 Bytes	396 ms
	NTRU251:3	3236 Bytes	9.543 s	1382 Bytes	313 ms	934 Bytes	784 ms
3	NTRU167:3	2850 Bytes	2.953 s	856 Bytes	158 ms	714 Bytes	221 ms
	NTRU251:3	2850 Bytes	6.008 s	856 Bytes	315 ms	714 Bytes	398 ms
4	NTRU167:3	3132 Bytes	2.849 s	970 Bytes	52 ms	714 Bytes	221 ms
	NTRU251:3	3132 Bytes	5.764 s	970 Bytes	64 ms	714 Bytes	398 ms
5	NTRU167:3	3556 Bytes	2.272 s	970 Bytes	52 ms	786 Bytes	124 ms
	NTRU251:3	3556 Bytes	4.315 s	970 Bytes	64 ms	786 Bytes	157 ms

and almost a 40% for the NTRU251:3.
On the other hand the *Optimized multiplication* in version 4 reduces slightly the key generation timing improving the computation of h while version 5 optimization affects the Newton iteration method to convert the inverse of f.
Analogously figure 6.2 and figure 6.3 show the evolution of *Encryption* and *Decryption* respectively.

Figure 6.2 shows a reduction around a 12% during the encryption when we compare version 2 versus version 1 since the multiplication to encrypt the message is reducing modulo q the coefficients, see section 3.1.3.
The other major optimization in encryption is produced in version 4 when is applied the *Optimized multiplication*. Reduction is around a 66% for the parameter set NTRU167:3. This is possible since multiplication is the most complex operation during encryption and the space of the random polynomial r has around a 78% of zero coefficients.
More interesting is the reduction around an 80% of the total cost of encryption in version 4 for the parameter set NTRU251:3 since r has around an 87 % of zero terms. From these results we can conclude the *Optimized multiplication* developed is highly scalable. Although the total cost for Algorithm 7 is higher than $4 \cdot N \cdot dr$ additions; but the computational reduction cost is very significant compared to previous versions.

In figure 6.3 we see how decryption is also affected by the modulo implementation in version 2 but only around a 3%. This is due to the fact that Decryption in version 2 has one multiplication and a centering process which computes the modulo p not taking advantage of the *AND* operation. On the other hand version 3 eliminates the last multiplication modulo p, see section 6.2.1. This is reflected with a time reduction of almost a 50% for Decryption for both security levels because the multiplication reduced modulo p is more costly than reduced modulo q.
At last but not least version 5 shows how the *Optimized Multiplication for f* shrinks the decryption cost around a 40% and more than a 60% for the security levels NTRU167:3 and NTRU251:3 respectively. The *Optimized Multiplication for f* function for the private key f has a similar behavior to the *Optimized Multiplication* for the random polynomial. Although the final cost of the *Optimized Multiplication for f* is still not yet $2 \cdot N$ multiplications and $4 \cdot N \cdot df$ additions.

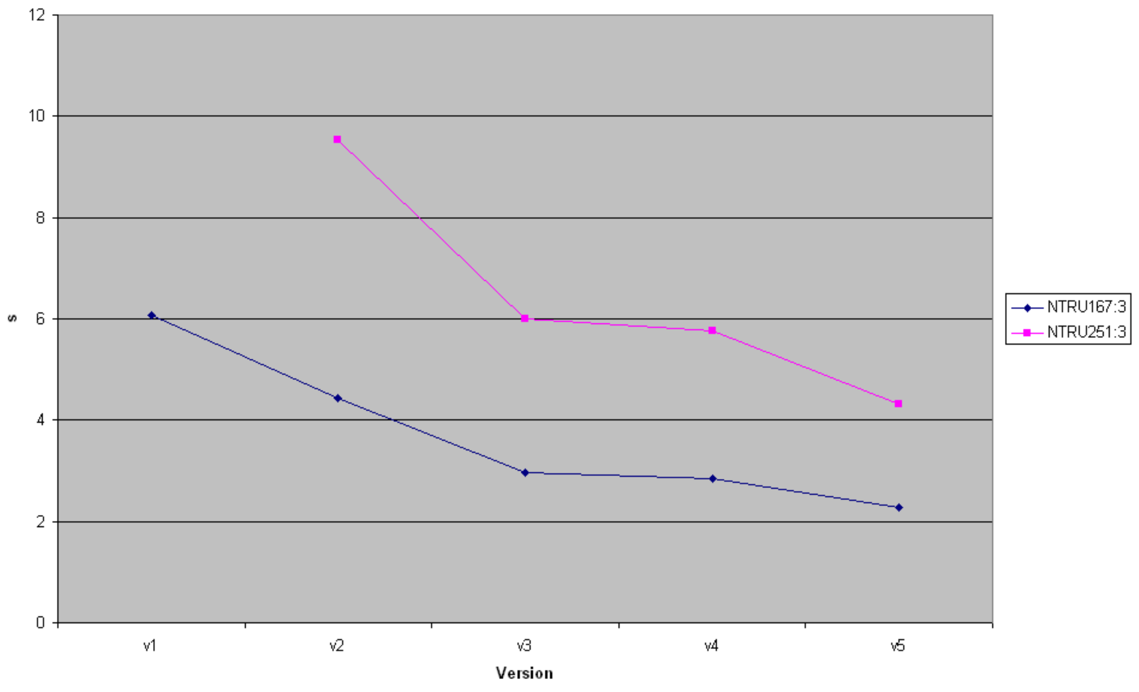

Figure 6.1: Key Generation timings on ATMega128 @ 4 Mhz.

6.3.2 ATMega163

In this section we presented the final results of the implementation done for the ATmega163 microcontroller. Table 6.2 presents the results for computational cost.

Table 6.2: NTRU167:3 on ATMega163 @ 4Mhz.

Version	EEPROM	Keygeneration		Encryption		Decryption	
#		SRAM	Time	SRAM	Time	SRAM	Time
3	334 Bytes	674 Bytes	2.953 s	672 Bytes	159 ms	506 Bytes	222 ms
4	334 Bytes	674 Bytes	2.852 s	541 Bytes	53.3 ms	506 Bytes	222 ms
5	334 Bytes	625 Bytes	2.307 s	541 Bytes	53.3 ms	457 Bytes	131 ms

The small difference from the results on table 6.2 and table 6.1 are due to the EEPROM access. For the ATmega163 in our implementation the EEPROM is required for the storage of the keys due to the lower SRAM resources. Besides the EEPROM memory used, table 6.2 also shows the maximum SRAM peak in each process of the NTRU scheme. It is interesting to note that *Key Generation* requires to store during the Newton iteration method a minimum of four polynomials plus some extra bytes for counters. On the other hand, Encryption and Decryption can be implemented storing three polynomials in SRAM if the private and public keys are stored in EEPROM. Although in our implementation, Encryption requires enough

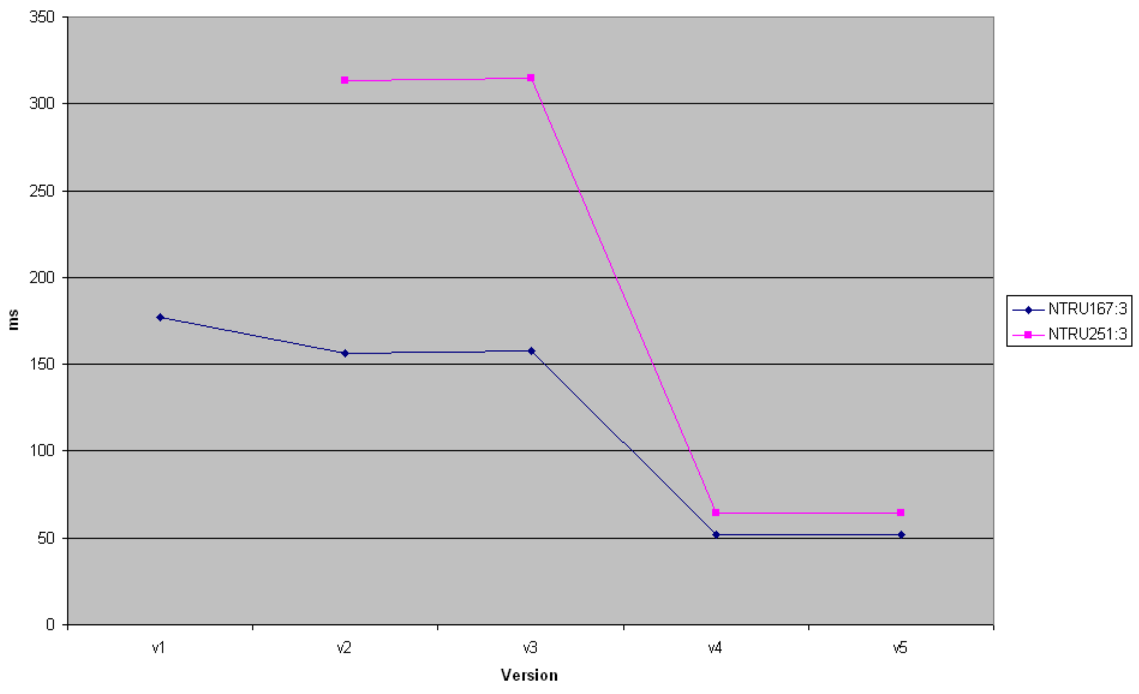

Figure 6.2: Encryption timings on ATMega128 @ 4 Mhz.

space in SRAM for four polynomials in order to pass the clear message and the public key as input parameters of the Encryption function. This is done since Encryption may require different public keys depending on the user or entity we want to address. In addition, we avoid extra readings from the EEPROM, making the process faster.

Finally is shown in figure 6.4 a comparison between *Encryption* and *Decryption* for the different versions running on the ATmega163 microcontroller for the parameter set N=167, q=128 and p=3.

From figure 6.4 we obtain encryption is *3x* faster in version 3 than in version 4 while decryption is 1.7x faster from version 4 to version 5 when using NTRU167:3. Also the maximum SRAM required is reduced around a 10% during the *Key Generation* and *Encryption*, and a 20% for *Decryption*. But most significant is the average SRAM utilization which decreases in a higher percentage during the inverse process. Also the multiplications presented in this thesis are highly scalable. Still *Decryption* seems to be much slower than expected compared to *Encryption*. This is due to the parameter sets, even both multiplication implemented in encryption and decryption have similar costs, the parameter *df* is around three times greater than *dr* having consequently a higher number of operations in the *Optimized Multiplication for f*.

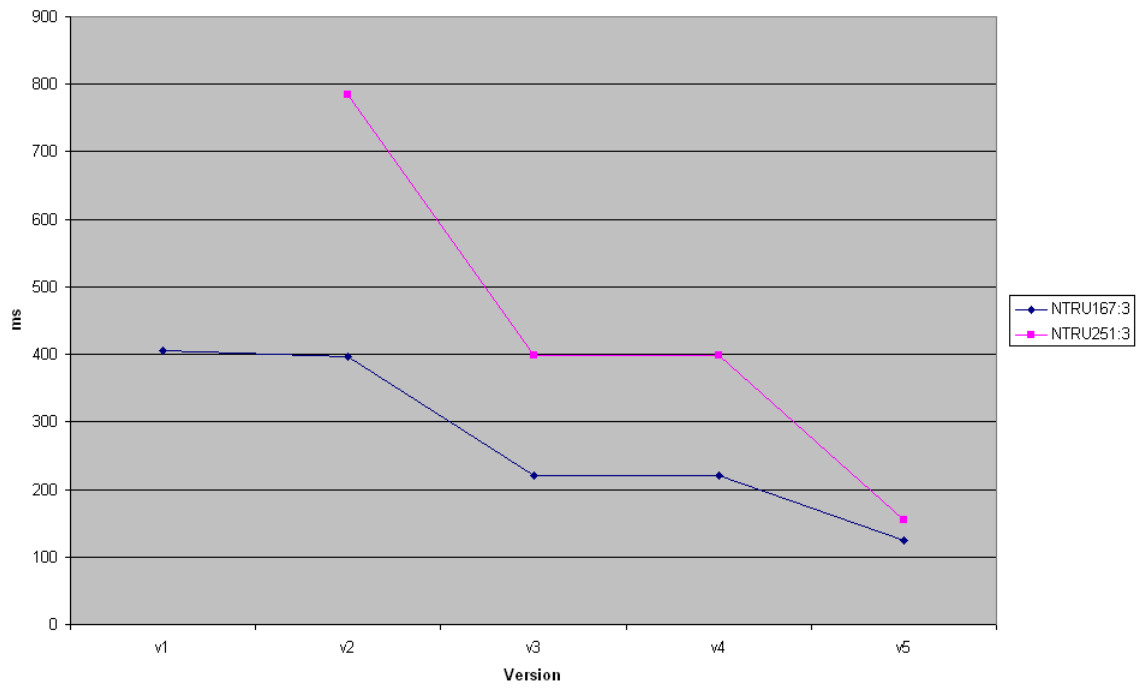

Figure 6.3: Decryption timings on ATMega128 @ 4 Mhz.

6.4 Comparison with RSA, ECC and HECC

This section compares the performance of our implementation of NTRUEncrypt versus the published implementation results for other public key cryptosystems when providing similar security. More precisely we compare NTRUEncrypt with RSA, ECC and HECC. The RSA and ECC timings for encryption and decryption have been extracted from [42] and can be observed in table 6.3.

Table 6.3: ATmega128 @ 8MHz

	Code	Data mem	Time
ECC secp160r1	3682 Bytes	282 Bytes	0.81 s
RSA-1024 public-key e = 216 + 1	1073 Bytes	542 Bytes	0.43 s
RSA-1024 private-key w. CRT	6292 Bytes	930 Bytes	10.99 s
NTRU251:3 encryption	970 Bytes	804 Bytes	32 ms
NTRU251:3 decryption	786 Bytes	618 Bytes	78 ms

The published results for ECC and RSA in table 6.3 has been executed on the ATmega128 microcontroller with the clock frequency set to 8 MHz (ATMega128 can be set up to 16 MHz). Table 6.4 shows the result published for HECC running on an 8051 microcontroller at 12 MHz in [43]. While Table 6.5 results for HECC on an ARM7 published also in [43] and originally

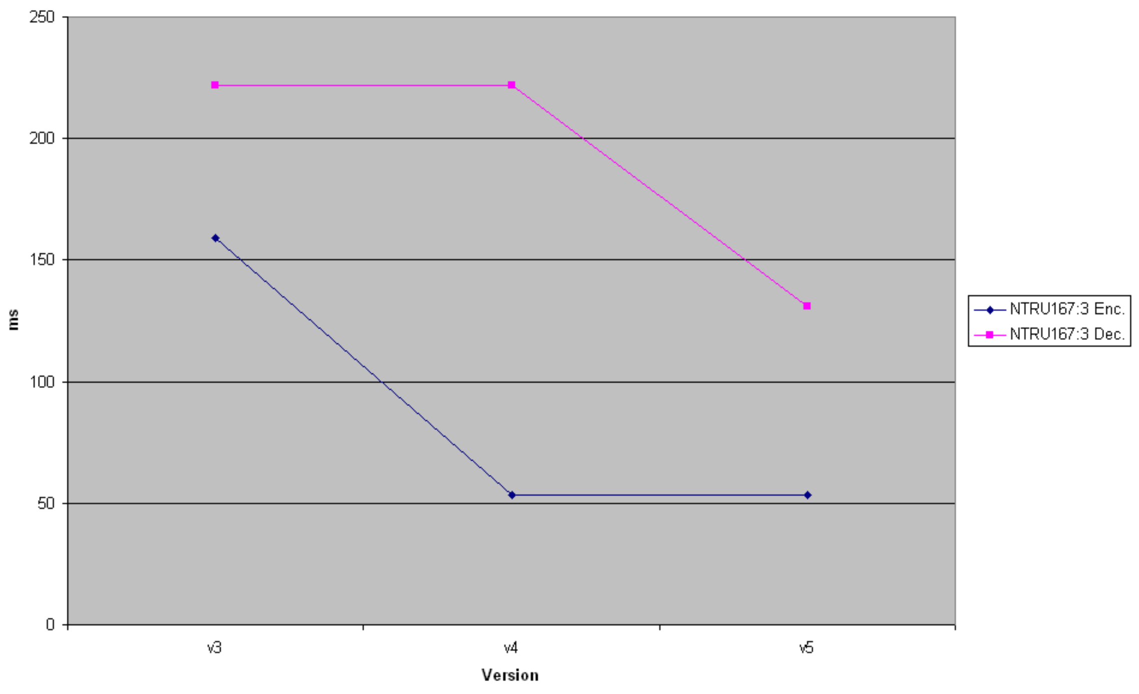

Figure 6.4: Encryption and Decryption timings on ATMEga163 @ 4 Mhz.

published in [44] and [45].

Table 6.4: HECC on 8501 microcontroller plus Keil C51 @ 12 MHz.

Implementation	ROM	XRAM	FPGA	Performance
C (Inversion SW)	11754 Bytes	820 Bytes	3300	191.7 s
C+ASM (Inversion SW)	12284 Bytes	820 Bytes	3300	64.9 s

Table 6.5: HECC on ARM7

Field	Frequency	Perf.
GF(2^{83})	80 MHz	71.56 ms
GF(2^{80})	80 MHz	374 ms

For reference it is assumed RSA-1024, ECC secp160r1 and HECC GF(2^{83}) and HECC GF(2^{80}) provide a key strength of 80 bits. It is also assumed NTRU251:3 provides 80 bits key security strength. Although some research point to NTRU167:3 with the right spaces can provide a security strength of 80 bits different techniques which take advantage of decryption failures or the implementation of an hybrid attack may question this strength even for NTRU251:3.

From table 6.3 we can observe memory data for RSA-1024 encryption is a bit smaller than NTRU251:3 SRAM usage. Concerning memory requirements ECC presents very small keys which are reflected in the data memory consumption, only 282 bytes for ECC secp160r1 for both, the encryption and the decryption. For RSA-1024 the memory data during decryption is incremented over passing the memory required in NTRU251:3 which is 618 bytes in version 5. Table 6.4 shows HECC requires 820 bytes in XRAM and more than 10KB in ROM making impossible to implement it in such a device as the ATmega163.

On the other hand to compare the computational cost it has to be considered the NTRU results presented in table 6.1 are tested simulating ATmega128 μC running at **4 MHz** instead of 8 MHz. When running at **8Mhz** we obtain the **32 ms** for *Encryption* and **78 ms** for *Decryption*. Compared to the 810 ms of ECCsecp160r1 there is one order of magnitude of difference. The closest approach is RSA-1024 in encryption which takes 430 ms but decryption goes up to almost 11 seconds which is very far from NTRU's 78.5 ms HECC is even further from NTRU's efficiency in computational resources taking up to 64.9 seconds. HECC encryption and decryption timings are only in the NTRU performance when using a microcontroller with a 10x higher frequency and a 32-bit architecture as shows table 6.5.

Chapter 7

Last security standards

During the development of this thesis new security standards have been released in the IEEE P1363.1/D10 draft, see [6]. This draft defines new standards based on the best known attack techniques until July 2008.

More specifically the draft considers an hybrid attack defined in section 3.3.3. This hybrid attack combines the lattice reduction and the meet-in-the-middle attack in order to reduce the total amount of work.

The lattice reduction work has been defined in the draft as W_{latt} while the meet-in-the-middle work is referred as W_{mitm}. To have an efficient attack these phases should be balanced to take the same amount of time.

In an hybrid attack the lattice reduction algorithm is implemented using a selected sublattice of the main lattice. The sublattice should not include any vector with length shorter than a certain Gaussian value. Since the Gausian heuristic assures with a high probability no short vector is present is then measured the amount of reduction that can be performed in a given amount of time.

Empirically is obtained the running time t to remove a given number N_q of q-vectors using the best known currently method. It is given by

$$t = 0.9501N_q - 3\ln 2N_q - 123.58 \ .$$

The running time to obtain a slope d if there is no cliff can be related directly to the time to remove N_q q-vectors: if there is no cliff, the reduction is symmetric about N (in order to keep the determinant constant) so the slope $d = 1/(y_2 - y_1) = 1/2Nq$ resulting time t,

$$t = 0.4750/d + 3\ln 1/d - 123.58 \ .$$

Since lattice attacks are improving constantly the parameter sets in the draft IEEE P1363.1/D10 assume the following extrapolation line,

$$t = 0.2/d - 3\ln 1/d - 50 \ .$$

In the combinatorial phase the attacker searches a space of size K for a trinary polynomial with $c_1 + 1s$ and $c_2 - 1s$. The calculated amount of work the attacker must do to search this space using a standard collision search method is:

$$W_{search} = \frac{\binom{K}{c_1/2}\binom{K-c_1/2}{c_2/2}}{\sqrt{\binom{c_1}{c_1/2}\binom{c_2}{c_2/2}}} .$$

Wagner's generalized birthday paradox search [34] may highly reduce the search to

$$W_{search} = \frac{\binom{K}{c_1/2}\binom{K-c_1/2}{c_2/2}}{\binom{c_1}{c_1/2}\binom{c_2}{c_2/2}} .$$

Even it is not known how this attack could be implemented, the draft P1363.1 contemplates this possibility when assigning a given security level k.

It is also considered the probability that the search might not be successful which depends on the probability that the lattice reduction allows a correct guess to be confirmed, P_s. Where,

$$P_s = \binom{c_1}{c_1/2}\binom{c_2}{c_2/2} .$$

Also is considered the probability that the attacker has guessed the right values for c_1 and c_2 for a single rotation of the key is,

$$P_{split,1} = \frac{\binom{N-K}{d_1-c_1}\binom{N-K-(d_1-c_1)}{d_2-c_2}\binom{K}{c_1}\binom{K-c_1}{c_2}}{\binom{N}{c_1}\binom{N-c1}{c_2}} .$$

If the attacker is able to take advantage that the lattice contains N rotations of the key the probability P_{split} improves as follows,

$$P_{split,N} = 1 - (1 - P_{split,1})N .$$

Although is considered the private key $f = 1 + pF$ requires to solve the closest vector problem, CVP, where there is only one single rotation of the key, the draft in order to avoid future improved reduction algorithm considers $P_{split} = P_{split,N}$ instead of $P_{split,1}$.

Finally for the lattice reduction using Babai's method involves multiplying by a $2N$x$2N$ transformation matrix. Empirically has been obtained a bit security around $W_{reduction} = N^2/2^{1.06}$ for this multiplication. Still hence the matrix is the same in the different cases the bit security estimated is $W_{reduction} = N/2^{1.06}$ due to a possible optimization.

Having all this considerations the amount of work for certain c_1, c_2 given the values K, a, y_1 and y_2 obtained from the lattice reduction is

$$W_{mitm}(c_1, c_2) = W_{reduction} * W_{search} * W_{P_s}/P_{split} .$$

For the security parameters the draft P1363.1/D10 considers the meet-in-the-middle cost to be,

$$W_{mitm} = \min(c_1, c_2) W_{mitm}(c_1, c_2) \ .$$

The resulting security parameters presented in the standard P1363.1 [6] are given in table 7.1. Each security parameter set shown in the first column is defined by the parameters N, q and df. These parameters provide a security level offered against an attacker using the best techniques known in July 2008 shown in column "Known Strength". The column "Recommended security" gives the security level recommended in this standard [6] considering more powerful new attacks against the NTRU PKC may appear.

Table 7.1: IEEE P1363.1/D10 standards

Parameter set	N	q	df	Known strength	Recommended security
ees401ep1	401	2048	113	154.88	112
ees541ep1	541	2048	49	141.766	112
ees659ep1	659	2048	38	137.861	112
ees449ep1	449	2048	134	179.899	128
ees613ep1	613	2048	55	162.385	128
ees761ep1	761	2048	42	157.191	128
ees653ep1	653	2048	194	276.736	192
ees887ep1	887	2048	81	245.126	192
ees1087ep1	1087	2048	63	236.586	192
ees853ep1	853	2048	268	376.32	256
ees1171ep1	1171	2048	106	327.881	256
ees1499ep1	1499	2048	79	312.949	256

Chapter 8

Conclusions and Future Lines

NTREncrypt makes it possible to achieve high security levels without requiring a great investment in hardware. Encryption and Decryption have been the primary focus in which to implement computational optimizations. Key Generation has also taken some research to reduce its complexity along with the reduction of SRAM consumption to be able to run the NTRU PKC on the AMTEL ATMega163.

The final version of our implementation is able to generate the keys on the ATMega163, taking no more than 2.3 seconds and encrypting a message in close to 25 ms while decrypting in 62 ms for the parameter set NTRU167:3. The difference between the encryption and the decryption is remarkable; despite the fact both operations only have one polynomial star multiplication. This asymmetry between the encryption and the decryption is primarily due to the polynomial structure. Using the parameters NTRU167:3 involves using $dr = 18$ and an ideal total cost of the *Optimized multiplication* of $4 \cdot N \cdot dr = 4 \cdot 167 \cdot 18 = 12024$ additions. While in decryption the parameter $df = 60$ means having cost around $2 \cdot N = 2 \cdot 167 = 334$ multiplications and also $4 \cdot N \cdot df = 4 \cdot 167 \cdot 60 = 40080$ additions. The encryption and the decryption are highly scalable due to the multiplication functions' behavior, which depends not only in N but in the different space parameters.

Regarding the SRAM requirements, our implementation demands at least $4 \cdot N \cdot \lg_2 q$ bits of memory space if wanted to generate the keys in the constrained device. The final conclusion is that NTRUEncrypt seems to be very scalable and ideal for embedded devices, since the decryption and encryption timings are incredibly fast.

Future changes to version 5 of our implementation would be optimizing the developed multiplication functions into assembly to achieve the theoretical costs of the algorithms 6 and 7. Regarding the key generation there is a bottleneck in the Newton iteration method to convert the inverse modulo a prime to modulo q. In any case, the key generation can be executed in an external machine and is not used as frequently as encryption or decryption.

Finally, it is very interesting to remark that the NTRU PKC has several advantages in constrained devices. It is the fastest cryptosystem in the market making possible to provide different security levels at high speed with very low resources. Its simplicity makes the NTRU PKC ideal for low cost and low consumption devices.

Bibliography

[1] AVR Studio 4. Last checked: 10/5/2009. [Online]. Available: http://www.atmel.com/dyn/products/tools_card.asp?tool_id=2725

[2] J. Hoffstein, J. Pipher, and J. H. Silverman, "Ntru: A ring-based public key cryptosystem," in *ANTS*, ser. Lecture Notes in Computer Science, J. Buhler, Ed., vol. 1423. Springer, 1998, pp. 267–288.

[3] R. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, pp. 120–126, 1978.

[4] N. Koblitz, "Elliptic curve cryptosystems," *Mathematics of Computation*, vol. 48, no. 177, pp. 203–209, 1987.

[5] L. M. Adleman, J. DeMarrais, and M.-D. A. Huang, "A Subexponential Algorithm for Discrete Logarithms over Hyperelliptic Curves of Large Genus over GF(q)," *Theor. Comput. Sci.*, vol. 226, no. 1-2, pp. 7–18, 1999.

[6] William Whyte (editor) and Nick Howgrave-Graham and Jeff Hoffstein and Jill Pipher and Joseph H. Silverman and Phil Hirschhorn, "IEEE P1363.1 Draft 10: Draft Standard for Public Key Cryptographic Techniques Based on Hard Problems over Lattices." Cryptology ePrint Archive, Report 2008/361, 2008, http://eprint.iacr.org/2008/361.

[7] R. L. Rivest and C. E. Leiserson, *Introduction to Algorithms.* New York, NY, USA: McGraw-Hill, Inc., 1990.

[8] C. F. Gauss, *Disquisitiones Arithmeticae.* Paris: Blanchard, 1953.

[9] I. Kaplansky, *Fields and rings*, ser. Chicago Lectures in Mathematics. Chicago-London: The University of Chicago Press, 1972.

[10] R. McEliece, Ed., *Finite Fields for Computer Scientists and Engineers.* Kluwer Academic Publishers, 1987.

[11] M. R. Darnel and J. Martinez, "Michael R. darnel: Theory of lattice-ordered groups," Sep. 03 1997.

[12] T. M. Apostol, *Introduction to Analytic Number Theory.* Springer-Verlag, 1976.

[13] G. H. Hardy and E. M. Wright, *An Introduction to the Theory of Numbers*, 5th ed. Oxford University Press, 1979.

[14] H. Silverman, "NTRU cryptosystems technical report report 009, version 1 title: Invertibility in truncated polynomial rings author: Joseph H. silverman release date: Thursday, october 1, 1998," Nov. 13 1998. [Online]. Available: http://www.ntru.com/cryptolab/pdf/NTRUTech009.pdf

[15] A. Kondracki, "The Chinese Remainder Theorem," *Formalized Mathematics*, vol. 6, no. 4, pp. 573–577, 1997.

[16] H. Dubner, "Large Sophie Germain primes," vol. 65, no. 213, pp. 393–396, 1996.

[17] Hoffstein, Pipher, and Silverman, "NTRU: A ring-based public key cryptosystem," in *ANTS: 3rd International Algorithmic Number Theory Symposium (ANTS)*, 1998.

[18] N. Howgrave-Graham, J. H. Silverman, and W. Whyte, "Choosing parameter sets forwithand," in *CT-RSA*, ser. Lecture Notes in Computer Science, A. Menezes, Ed., vol. 3376. Springer, 2005, pp. 118–135.

[19] Andrew Odlyzko. Last checked: 20/9/2008. [Online]. Available: http://www.dtc.umn.edu/~odlyzko

[20] N. Howgrave-Graham, J. H. Silverman, and W. Whyte, "A meet-in-the-middle attack on an NTRU private key," Jun. 26 2003. [Online]. Available: http://www.ntru.com/cryptolab/pdf/NTRUTech004v2.pdf

[21] O. Regev, "Lattice-based cryptography," in *CRYPTO*, ser. Lecture Notes in Computer Science, C. Dwork, Ed., vol. 4117. Springer, 2006, pp. 131–141. [Online]. Available: http://dx.doi.org/10.1007/11818175_8

[22] Lenstra, Lenstra, and Lovasz, "Factoring polynomials with rational coefficients," *MATHANN: Mathematische Annalen*, vol. 261, 1982.

[23] R. Kumar and D. Sivakumar, "A sieve algorithm for the shortest lattice vector problem," in *Proc. 33rd ACM Symp. on Theory of Comput*, 2001, pp. 601–610.

[24] Khot, "Hardness of approximating the shortest vector problem in lattices," *JACM: Journal of the ACM*, vol. 52, 2005.

[25] Goldreich and Goldwasser, "On the limits of nonapproximability of lattice problems," *JCSS: Journal of Computer and System Sciences*, vol. 60, 2000.

[26] Aharonov and Regev, "Lattice problems in NP intersect coNP," *JACM: Journal of the ACM*, vol. 52, 2005.

[27] Lagarias, Lenstra, and Schnorr, "Korkin-zolotarev bases and successive minima of a lattice and its reciprocal lattice," *COMBINAT: Combinatorica*, vol. 10, 1990.

[28] N. Howgrave-graham, J. Hoffstein, J. Pipher, W. Whyte, and N. Cryptosystems, "On estimating the lattice security of NTRU," 2005.

[29] Keylength.com. Last checked: 18/9/2008. [Online]. Available: http://www.keylength.com/

[30] A. Menezes, Ed., *Advances in Cryptology - CRYPTO 2007, 27th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2007, Proceedings*, ser. Lecture Notes in Computer Science, vol. 4622. Springer, 2007.

[31] N. Howgrave-graham, "A hybrid lattice-reduction and meet-in-the-middle attack against ntru," in *CRYPTO*, ser. Lecture Notes in Computer Science, A. Menezes, Ed., vol. 4622. Springer, 2007, pp. 150–169.

[32] D. Micciancio, "The hardness of the closest vector problem with preprocessing," *IEEE Transactions on Information Theory*, vol. 47, p. 2001, 2001.

[33] Schnorr, "Lattice reduction by random sampling and birthday methods," in *STACS: Annual Symposium on Theoretical Aspects of Computer Science*, 2003.

[34] Wagner, “A generalized birthday problem (extended abstract),” in *CRYPTO: Proceedings of Crypto*, 2002.

[35] *ISO/IEC 9899-1999: Programming Languages—C*, International Organization for Standardization, Dec. 1999.

[36] ATmega128 Datasheet.

[37] H. Silverman, “NTRU cryptosystems technical report report 014, version 1 title: Almost inverses and fast NTRU key creation author: Joseph H. silverman release date: March 15, 1999,” Apr. 13 1999. [Online]. Available: http://www.ntru.com/cryptolab/pdf/NTRUTech014.pdf

[38] ATmega163 Datasheet. Last checked: 8/10/2008. [Online]. Available: http://www.atmel.com/dyn/resources/prod_documents/doc1142.pdf

[39] ATmega163 Instruction Set. Last checked: 10/04/2009. [Online]. Available: http://www.atmel.com/dyn/resources/prod_documents/doc0856.pdf

[40] “Optimizations for NTRU,” Dec. 17 2002. [Online]. Available: http://www.ntru.com/cryptolab/pdf/TECH_ARTICLE_OPT.pdf

[41] H. Silverman, “NTRU cryptosystems technical report report 010, version 1 title: High-speed multiplication of (truncated) polynomials author: Joseph H. silverman release date: Tuesday, january 5, 1999,” Jan. 07 1999. [Online]. Available: http://www.ntru.com/cryptolab/pdf/NTRUTech010.pdf

[42] N. Gura, A. Patel, A. Wander, H. Eberle, and S. C. Shantz, “Comparing elliptic curve cryptography and RSA on 8-bit CPUs,” in *CHES*, ser. Lecture Notes in Computer Science, M. Joye and J.-J. Quisquater, Eds., vol. 3156. Springer, 2004, pp. 119–132. [Online]. Available: http://springerlink.metapress.com/openurl.asp?genre=article&amp;issn=0302-9743&amp;volume=3156&amp;spage=119

[43] L. Batina, D. Hwang, A. Hodjat, B. Preneel, and I. Verbauwhede, “Hardware/software co-design for hyperelliptic curve cryptography (HECC) on the 8051μP,” in *CHES*, ser. Lecture Notes in Computer Science, J. R. Rao and B. Sunar, Eds., vol. 3659. Springer, 2005, pp. 106–118. [Online]. Available: http://dx.doi.org/10.1007/11545262_8

[44] J. Pelzl, T. Wollinger, and C. Paar, *Embedded Cryptographic Hardware: Design and Security.* NY, USA: Nova Science Publishers, 2004, ch. Special Hyperelliptic Curve Cryptosystems of Genus Two: Efficient Arithmetic and Fast Implementation, editor Nadia Nedjah.

[45] S. Baktir, J. Pelzl, T. Wollinger, B. Sunar, and C. Paar, “Optimal Tower Fields for Hyperelliptic Curve Cryptosystems,” in *38th Asilomar Conference on Signals, Systems and Computers, November 7-10, 2004, Pacific Grove, USA*. IEEE Signal Processing Society, November 2004.