Master Thesis

Computing Science
Cyber Security Specialisation

Radboud University

Optimizing NTRU using AVX2

Author:
Oussama Danba

First supervisor/assessor:
dr. Peter Schwabe

Second assessor:
dr. Lejla Batina

July, 2019

Abstract

The existence of Shor's algorithm, Grover's algorithm, and others that rely on the computational possibilities of quantum computers raise problems for some computational problems modern cryptography relies on. These algorithms do not yet have practical implications but it is believed that they will in the near future. In response to this, NIST is attempting to standardize post-quantum cryptography algorithms. In this thesis we will look at the `NTRU` submission in detail and optimize it for performance using AVX2. This implementation takes about 29 microseconds to generate a keypair, about 7.4 microseconds for key encapsulation, and about 6.8 microseconds for key decapsulation. These results are achieved on a reasonably recent notebook processor showing that `NTRU` is fast and feasible in practice.

Contents

Introduction

Cryptography has become ubiquitous in modern society even though we may not realize it. Whenever one uses something that is digital there is almost certainly some kind of cryptography involved. A few examples are: car doors that unlock when your car key gets close, smart cards that are used for payment or authentication, online web browsing (including digital purchases), or just regular instant messaging. For these applications we *require* cryptography to be unbroken.

In order for our cryptography to be secure we rely on the hardness of some mathematical problem such that no amount of achievable processing power is able to solve that problem. With the advent of quantum computers and accompanying quantum algorithms it turns out that some of the mathematical problems that are used in practice today are becoming efficiently solvable. For example, Shor's algorithm [50] poses a threat to RSA and Diffie-Hellman (even when using elliptic curves) since it is capable of finding the prime factors of an integer and solving the discrete-logarithm problem. Grover's algorithm [30] is relevant to symmetric-key cryptography and hash pre-images. It provides a quadratic speedup compared to non-quantum computers. Both are described in more detail in Section 2.5.

Fortunately, building quantum computers large enough to break these problems in practice is not yet feasible. However, it seems likely that this will happen at some point so we should prepare by creating cryptography that relies on mathematical problems that are still not solvable by quantum computers. This is known as post-quantum cryptography.

The National Institute of Standards and Technology (NIST) set up a "competition" in which (teams of) cryptographers can submit algorithms that are resistant to quantum computers [53]. These post-quantum algorithms are reviewed by other cryptographers for their security. While commonly referred to as a competition it is not quite a competition since a portfolio of different algorithms will be standardized at the end. Also, what constitutes a better algorithm is not quite clear yet in the context of post-quantum cryptography. Besides meeting the security goals of NIST, there is a wide variety in performance, keysizes, and signature sizes. The competition had about seventy submissions in the first round. The second round, which is the current round, has 26 submissions. Some of the submissions from the first round have been withdrawn due to weaknesses or being incomplete while others have alternatives that are considered preferable. There are also some submissions from the first round that have been merged into a single submission.

In this thesis we will look at the `NTRU` [48] submission and maximize its performance. `NTRU` is based on an older cryptosystem with the same name [32]. The original cryptosystem will be shown as NTRU while the cryptosystem proposed in the NIST competition will be shown as `NTRU`. `NTRU` is a merger of the first round NTRUEncrypt [31] and NTRU-HRSS-KEM [33] submissions.

This thesis will be separated into chapters. In Chapter 2 some cryptographic background will be covered that is necessary for the other chapters. Chapter 3 will cover the construction of `NTRU` and its operations. Chapter 4 will be about the work that was performed during this Master's thesis. It aims to explain the work performed, why certain decisions were made, and the results of those decisions. Chapter 5 will look at the final results of the AVX2 implementation and how it compares to other submissions in the NIST competition. The thesis closes out with a short conclusion in Chapter 6.

Cryptographic background and related work

This chapter covers key concepts in cryptography such that the context around `NTRU` is clear. The sections about post-quantum cryptography and lattice-based cryptography are especially important to understand the cryptographic security of `NTRU`.

2.1 Symmetric-key cryptography

A cryptographic system that uses a secret key that is shared between all parties involved (typically two parties) is known as a symmetric-key system. It is possible for the inner workings of the cryptographic system itself to be a secret as well but is considered poor practice in modern cryptography because it comes with several problems for only a small gain in short-term security. The first problem is that such a secret system might contain weaknesses an attacker uncovers but the original designers are unaware of. If a cryptosystem is public there is a higher chance of such weaknesses being found before they are being exploited. A second reason for why it is considered poor practice is that the security of the cryptosystem may rely on the fact that the cryptosystem is secret. Consequently, if the cryptosystem loses its secrecy then it is permanently compromised. If the cryptosystem is public then only the key must be kept secret and can be changed if compromised. Finally, a somewhat pragmatic reason, is that in order to rely on a cryptosystem there has to be some amount of trust in that cryptosystem. A public cryptosystem allows others to verify the security claims made.

Auguste Kerckhoffs stated in the nineteenth century that a cryptosystem should not require secrecy and that it should not be a problem if the system falls into enemy hands [38]. This is now known as Kerckhoffs' principle and is one of the first statements that objects to the "security through obscurity" design.

For modern symmetric-key ciphers there are typically two variants: block ciphers and stream ciphers. For encryption a block cipher takes a secret key and a fixed length input which produces an unreadable output, the so-called ciphertext. Decryption is similar and requires the secret key and the ciphertext in order to produce the original plaintext. Since block ciphers work on fixed-length inputs they are commonly used in so called modes of operation to support longer length inputs. For this to work the input is (securely) padded to a multiple of the block length and depending of the mode of operation the secret key is expanded to a keystream.

Stream ciphers differ from block ciphers in that they work per unit of the input (typically a bit or a single digit). In order to support this, the secret key is expanded to a pseudorandom keystream of the same length as the input and is then combined; the combination in practice is typically an exclusive-or. The specific expansion to the keystream is dependent on which cipher is used. Due to the fact that block ciphers are used with modes of operation they can strongly resemble stream ciphers where the unit length is the block length.

The main disadvantage of symmetric-key cryptography is that a shared key is necessary. As a result, there must be some way to securely exchange keys beforehand. Another disadvantage is that both parties must keep the key secret instead of only on party.

2.2 Public-key cryptography

Public-key cryptography, also known as asymmetric cryptography, differs from symmetric-key cryptography in that there is no longer a shared key but a public key and a private key. This idea comes from the 1976 paper *New directions in cryptography* [22] by Diffie and Hellman, which proposes to take a secret key and split it into a private part which can solely be used for decryption and a public part (available to anyone) which can solely be used for encryption. Constructing such a system did not happen until later with RSA [46] being the first public one (in 1973 Clifford Cocks independently discovered RSA but this was classified information until 1997).

In the same paper they did propose the Diffie-Hellman key exchange where two parties can establish a shared secret key over an unsecured channel without an adversary getting to know the secret key. The existence of the Diffie-Hellman key exchange made symmetric-key cryptography much easier to use as there is a way to establish a secret key without other channels. It also made it possible for keys to be used only for one communication session such that leaking a secret key has no influence on past or future communication.

Note that public-key cryptography, unlike symmetric-key cryptography, is only possible for one-way communication as the party which holds the private key has no means to send a confidential message to a specific holder of a public key. There are a few common ways of getting around this. One option is to use a key exchange algorithm like Diffie-Hellman to agree on a secret key and then use symmetric-key cryptography. Another option is to send a symmetric key using public-key cryptography and use that as the key. One last option: both parties have a keypair such that messages are encrypted with each other's public key. Some other mechanism is required to ensure you have the correct public key.

Definition 1 (Public-key cryptography scheme)**.** *A public key cryptography scheme PKE is a tuple PKE=(`KeyGen`, `Encrypt`, `Decrypt`) of polynomial time algorithms which are defined as:*

- *Key generation `KeyGen` is a probabilistic algorithm that given a security parameter 1^k produces a keypair (`sk`, `pk`).*
- *`Encrypt` is a (probabilistic) algorithm that, given a message `M` and a public key `pk`, produces a ciphertext `c`.*
- *`Decrypt` is a deterministic algorithm that produces a plaintext `M` or possibly an error when given a secret key `sk` and a ciphertext `c`.*

From the definition of a PKE one can see that encryption can be probabilistic. In order to distinguish between probabilistic and deterministic encryption the terms probabilistic public-key encryption (PPKE) and deterministic public-key encryption (DPKE) are used. The properties of PPKE and DPKE are different when proving security and must thus be viewed separately.

2.2.1 Digital signatures

Besides the fact that public-key cryptography can be used for confidential communication through encryption and decryption it can also provide authenticity, integrity, and non-repudiation through a scheme called digital signatures. If one party wants to send some message it can apply a secret function on that message; this is known as *signing* and produces a *signature* over that message. If the receiver then applies a public function over that message in combination with the signature, which is called *verification*, the output tells you whether the signature belongs to that message and thus verifying that the secret function was applied correctly.

In practice the secret function requires the private key while the public function requires the public key. Since there is only one party that holds the private key only that party is able to produce a valid signature and thus a valid signature proves that the message indeed came from the holder of the private key. This provides authenticity of the message. The same reasoning holds for non-repudiation. Since only the holder of the private key can produce a valid signature that means that a valid signature is proof that the holder of the private key sent this message at some point. Integrity is provided by the idea that if the message, signature, or both are modified that the signature will no longer match the message thus the message is no longer guaranteed to be unaltered. This can happen by an adversary or transmission errors.

The concept of signatures was also described by Diffie and Hellman [22] and is incredibly important for secure communication. Note that all of these properties are only guaranteed as long as the private key has not been compromised. In practice revocation lists exist in order to keep track of keypairs that have been compromised.

2.2.2 Key encapsulation mechanisms

As described earlier, public-key cryptography can be used to encrypt a secret key which is then decrypted be the receiver such that both parties share a secret key for use in symmetric-key cryptography. Since symmetric keys are fairly small there will be some padding involved which must not reveal any details about the key itself in any way. This can be a tricky process and thus there is an alternative option called key encapsulation mechanisms (KEM) [21]. Rather than using a normal public-key encryption algorithm with as input a padded normal symmetric key we have a specific algorithm for the establishment of shared secret keys. Typically a random element is taken as input and is then encapsulated (similar if not identical to encryption). The receiver decapsulates it (similar if not identical to decryption). Both parties hash (more detail on hashing later) the random element which is then used as the shared secret key. The sampling of a random element has the length that the public-key scheme requires which has the result that there is no need for padding in a KEM. For the competition NIST provides an interface for such KEMs besides an interface for public-key encryption algorithms.

Definition 2 (Key encapsulation mechanism). *A key encapsulation mechanism KEM is a tuple KEM=(`KeyGen`, `Encapsulate`, `Decapsulate`) of polynomial time algorithms which are defined as:*

- *Key generation `KeyGen` is a probabilistic algorithm that given a security parameter 1^k produces a keypair (`sk`, `pk`).*
- *`Encapsulate` is a probabilistic algorithm that given a public key `pk`, produces a key `k` and the ciphertext `c`.*

- `Decapsulate` *is a deterministic algorithm that produces a key* `k` *when given a secret key* `sk` *and a ciphertext* `c`*. In case of failure it can return an error.*

In order to make assertions about the correctness and security of PKEs and KEMs it is typical to prove these assertions. Proving security is discussed in Section 2.4.

2.3 One-way functions

A one-way function is a function that is easy to compute but hard to invert. The implementation of public-key cryptography mentioned above relies on these one-way functions to be truly one way unless one has some special information. This subset of one-way functions is known as trapdoor functions. Some algorithms such as RSA rely on the integer-factorization problem. In this problem the efficient one-way function is the multiplication of two prime numbers to form one large integer whereas the inverse is to find the two prime numbers forming that large integer (which is hard). If the prime factors are sufficiently large you can not efficiently find the prime factors using classical computers. If you were able to find the prime factors you can compute the Euler totient function which can be used together with the public exponent e to find the private exponent d.

Other cryptographic schemes such as Diffie-Hellman rely on the discrete log problem. Given some group G and an element g in that group the computation of $g^a = x$ (where a is a positive integer and $x \in G$) is efficient. However finding a given G, g, and x is inefficient.

2.3.1 Cryptographic hash functions

A cryptographic hash function H is a one-way function defined as follows:

$$H : \{0,1\}^* \to \{0,1\}^n$$

In other words, H takes an arbitrary length input and transforms it into a fixed length output. The output is known as a *hash* or *digest.* In order for a cryptographic hash function to be considered secure for use in cryptographic purposes the following properties should hold (of which the first two come from the definition of a one-way function):

Pre-image resistance Given some hash h it should be difficult to find an input m that satisfies $H(m) = h$.

Second pre-image resistance Given some input m_1 it should be difficult to find a different input m_2 for which $H(m_1) = H(m_2)$.

Collision resistance Finding two arbitrary inputs m_1 and m_2 that are different for which $H(m_1) = H(m_2)$ should be difficult.

Cryptographic hash functions on their own have some uses such as password verification but they are typically used as building blocks within other schemes. For example, hashing a message before signing allows the signature to be generated over a smaller input which can be beneficial to efficiency depending on the signature scheme. Another use is within HMAC where a hash function is used to create a message authentication code (MAC). A MAC is used to provide authenticity and integrity over a message without the use of public-key cryptography (a single symmetric key is used for both generation and verification). A final example mentioned earlier is the use within a KEM where the random element is hashed. This reduces the random element to a suitable length to be a session key and means that the chooser of the random element can not choose a specific session key (although a hash can be reused by using the same random element twice).

2.4 Proving security

When dealing with cryptography ideally we would be using cryptography that is provably secure. That is, the capabilities of an adversary are well-defined and a correct mathematical proof exists that shows that breaking the cryptosystem requires solving a hard problem. However, in practice this turns out to be quite difficult and many proofs are not quite correct [44, 36] [9, 51] [15, 27] [43, 19][1]. An attacker model might be subtly wrong or incomplete, assumptions may be too strong and are in fact false, or the proofs are simply interpreted wrongly.

An alternative approach is to define properties a cryptosystem wants to reach and proof for a cryptosystem how it accomplishes those properties. Two of the properties that are commonly used for PPKEs and KEMs are indistinguishability (IND) and non-malleability (NM). Indistinguishability describes the notion that an adversary is unable to learn anything about the plaintext when given a ciphertext [29]. Non-malleability describes the notion that an adversary can not alter a ciphertext and have the plaintext be meaningfully altered [23]. Indistinguishability says something about the privacy of a cryptosystem while non-malleability says something about how the cryptosystem is tamper-proof. Typically, indistinguishability in a PPKE is described as a game with a challenger and an adversary that has some capabilities (depending on his strength) and his target is to break the system. We will now describe the games and how the strength of the adversary differs.

[1]Paper with security proof followed by paper that shows flaws in the proof. In some cases an alternative proof can be provided such that the security still holds.

Definition 3 (IND-CPA)**.** *Indistinguishability under Chosen Plaintext Attack.*

- *Challenger generated a keypair (`sk`, `pk`).*
- *Adversary generates two messages m_1 and m_2 of the same length and sends both to the challenger. The adversary is allowed to do any polynomial time operations including calls to an encryption oracle.*
- *Challenger randomly chooses one of the two messages to encrypt and sends the ciphertext to the adversary.*
- *Adversary is allowed to do any polynomial time operations including calls to an encryption oracle. The adversary outputs a guess which plaintext was encrypted.*
- *If the guess was correct then the adversary wins.*

In the IND-CPA game we would expect an adversary to have no advantage in winning (in reality, negligible advantage) in polynomial time if the PPKE is well-constructed. Without the polynomial time constraint an adversary could theoretically break the system and be always correct.

Definition 4 (IND-CCA1)**.** *Indistinguishability under non-adaptive Chosen Ciphertext Attack.*

- *Challenger generated a keypair (`sk`, `pk`).*
- *Adversary calls the encryption or decryption oracle for arbitrary plaintexts and ciphertexts (some polynomial amount of times).*
- *Adversary generates two messages m_1 and m_2 of the same length and sends both to the challenger.*
- *Challenger randomly chooses one of the two messages to encrypt and sends the ciphertext to the adversary.*
- *Adversary is allowed to do any polynomial time operations including calls to an encryption oracle. The adversary outputs a guess which plaintext was encrypted.*
- *If the guess was correct then the adversary wins.*

IND-CCA1 is essentially the same as IND-CPA with the addition that an adversary is allowed to have access to the decryption oracle before receiving the ciphertext. If a PPKE is IND-CCA1 secure then the expectation is that the PPKE does not weaken over time.

Definition 5 (IND-CCA2)**.** *Indistinguishability under adaptive Chosen Ciphertext Attack.*

- *Challenger generated a keypair (`sk`, `pk`).*

- *Adversary calls the encryption or decryption oracle for arbitrary plaintexts and ciphertexts (some polynomial amount of times)*
- *Adversary generates two messages m_1 and m_2 of the same length and sends both to the challenger.*
- *Challenger randomly chooses one of the two messages to encrypt and sends the ciphertext to the adversary.*
- *Adversary is allowed to do any polynomial time operations including calls to a decryption and encryption oracle with the exception that the adversary can not send the challenge ciphertext to the decryption oracle. The adversary outputs a guess which plaintext was encrypted.*
- *If the guess was correct then the adversary wins.*

IND-CCA2 allows the decryption oracle to be used after receiving the ciphertext. The idea here is that using the decryption oracle after receiving the ciphertext should not reveal anything that can help making the correct guess. Since IND-CCA2 gives an adversary the most capabilities it implies that a IND-CCA2 secure PPKE is also IND-CCA1 and IND-CPA secure. The games are essentially the same for non-malleability. There is one important fact to note and that is that IND-CCA2 secure implies NM-CCA2 secure [8].

2.5 Post-quantum cryptography

The complexity class P is for decision problems for which solutions can be found (and verified) in polynomial time and are commonly considered efficient to compute. The complexity class NP (non-deterministic polynomial) on the other hand is for decision problems where given the answer “yes” to a problem and the proof, then it is verifiable in polynomial time. One might imagine that there are other complexity classes such as when the answer is “no” and the proof is verifiable in polynomial time (known as co-NP). One of these complexity classes is known as BQP which stands for bounded-error quantum polynomial time. It describes the set of decision problems where a solution to a problem can be found by a quantum computer in polynomial time where the probability of the answer being wrong is at most $\frac{1}{3}$.

It turns out that some problems relied on in cryptography such as integer factorization and discrete-logarithm are in fact contained within BQP (besides being contained within NP; whether this holds for all problems is not known). Shor’s algorithm [50], showed this by proposing an algorithm that factors an integer using $\mathcal{O}((\log N)^2(\log\log N)(\log\log\log N))$ steps on a quantum computer and some polynomial amount of post-processing on a classical computer. This algorithm is also able to solve the discrete-logarithm problem. So far only small integers have been factored due to the fact that building a reliable large quantum computer with enough qubits has proven to be difficult.

Besides Shor's quantum algorithm there is also Grover's algorithm [30]. This algorithm finds the input belonging to a specific output for a black-box function in $\mathcal{O}(\sqrt{N})$ steps rather than $\mathcal{O}(N)$ steps. This provides a quadratic speedup instead of an exponential one like Shor's algorithm but still significantly reduces the amount of attempts needed to brute-force search a symmetric key or hash preimage. For symmetric-key cryptography a simple countermeasure is to simply double the length of keys. For hashes it is a little harder to estimate how much gain there actually is [5] but a doubling of the hash output size should be sufficient.

Thus far we have seen that the existence of quantum computers pose theoretical problems for modern cryptography. However, in practice we have not yet seen any evidence that any cryptography in use has been compromised. That is not to say we should disregard quantum-resistant cryptography for now. The development of post-quantum cryptography takes time and so does building confidence in those algorithms [10]. It should also be considered that any software built today might still be operational a decade or two from now. Those systems would not be secure anymore unless someone is willing to support older systems. It should also be considered that any encrypted information might be stored such that it can be decrypted when quantum computers breaking cryptography becomes reality. If the gap between when post-quantum cryptography starts being used and when quantum computers can break current cryptography is large then that stored encrypted information will be older. It is for these reasons that efforts such as the NIST post-quantum cryptography "competition" [53] and PQCRYPTO [45] are working on standardization and implementation of post-quantum cryptography.

Luckily there exist other problems on which cryptography can rely. These are currently believed not to be broken by the existence of quantum computers. Some of these will be briefly discussed here while lattice-based cryptography will be discussed in more detail since the security of `NTRU` relies on it.

Hash-based digital signature schemes. Hash-based signatures schemes are conservative in approach and make use of well-known cryptographic hash properties to build a signature scheme that is resistant to quantum computers. The reliance on hash functions means that the amount of security assumptions can be small. Another common advantage of using hash functions is that the hash function itself can be swapped out for another hash function. This provides flexibility in case a hash functions turns out to be a problem or when a specific platform has better support for other hash functions. The main disadvantage of hash-based signature schemes is that the signatures tend to be rather large in comparison to other signatures. Smaller signatures are possible but sacrifice performance. The hash-based signature scheme XMSS [18] has been published as an informational RFC and the NIST competition contains SPHINCS$^+$ [6] (which has the advantage of not being stateful).

Code-based cryptography. Code-based cryptography schemes work by using error-correcting codes. The typical idea is that some error (below a threshold) is introduced in the ciphertext depending on the public key. Only the holder of the private key can efficiently restore those errors and thus get the corresponding plaintext. The first implementation was proposed by McEliece [42]. The largest downside is that the size of the public key required is large. Attempts have been made to add more structure in order to reduce the size of the public key but most of those attempts have been shown to have problems [10]. The original McEliece scheme with additions (such as a KEM with IND-CCA2 security) has been proposed to the NIST competition as Classic McEliece [12]. Another code-based cryptography scheme that is in the NIST competition is NTS-KEM [2].

Multivariate cryptography. The idea of multivariate cryptography is to use quadratic polynomials over a finite field. Given the input for a polynomial it is easy to compute the resulting value; the inverse is not true and is known to be NP-hard [10]. This is known as the MQ problem. Rainbow, a second round signature submission that is based on multivariate cryptography uses an Unbalanced Oil and Vinegar (UOV) scheme for its security. It has a large size for the public and private keys as a result but has short signature sizes. Rainbow does not have a formal proof that connects the security of Rainbow to the hardness of the MQ problem; this is also the case for most of the other submissions that use multivariate cryptography. The exception is MQDSS [20], MQDSS has a reduction from the MQ problem. MQDSS has much shorter keysizes than the other submissions but as a result has large signature sizes.

Supersingular elliptic curve isogeny cryptography. This class of cryptography started out with a single cryptographic system: Supersingular isogeny Diffie-Hellman key exchange (SIDH). The idea was to create a Diffie-Hellman replacement. The key exchange ended up being similar to Diffie-Hellman with the difference that instead of using finite cyclic groups or normal elliptic curves, a supersingular elliptic curve is used together with supersingular isogeny graphs. The result is that SIDH is an easy replacement for ECDH with a small keysize compared to other post-quantum schemes.

The same idea was later used to create a digital signature scheme. In the NIST competition there is SIKE (Supersingular Isogeny Key Encapsulation) that makes use of supersingular isogenies.

2.6 Lattice-based cryptography

In order to understand lattice-based cryptography we will first define *what* a lattice is. Imagine one has an n-dimensional space which has n-linearly independent vectors called a *basis*. An example would be the space $\mathbb{R}^2$ with vectors $\mathbf{b}_1 = (0, 1)$ and $\mathbf{b}_2 = (1, 0)$ (these need not be orthogonal). The lattice is then generated by all integer linear combinations of these vectors. In the example the result would be a regular tiling of $\mathbb{R}^2$ with each point having at least a distance of 1 to another point. See figure 2.1 for a visual representation.

Definition 6 (Lattice)**.** *A lattice $\mathcal{L} \in \mathbb{R}^n$ with basis vectors $\mathbf{b}_1, \ldots, \mathbf{b}_n \in \mathbb{R}^n$ is defined as the set $\mathcal{L}(\mathbf{b}_1, \ldots, \mathbf{b}_n) = \left\{ \sum_{i=1}^{n} a_i \mathbf{b}_i \,\middle|\, a_i \in \mathbb{Z} \right\}$.*

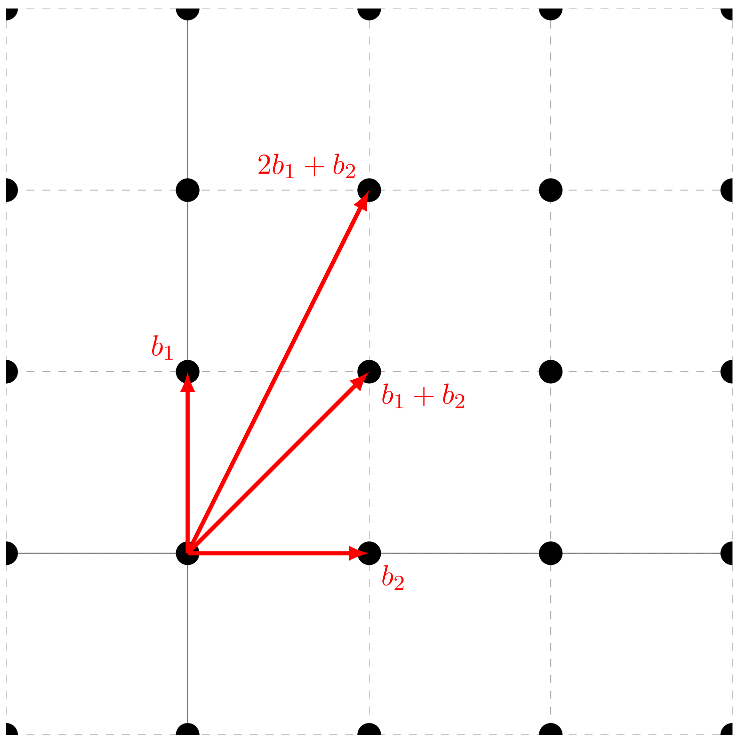

Figure 2.1: Lattice over $\mathbb{R}^2$ with $\mathbf{b}_1 = (0, 1)$ and $\mathbf{b}_2 = (1, 0)$

There are a few well known hard computational problems that exist on lattices that are interesting:

Shortest Vector Problem (SVP) Given some basis $\mathbf{B}$ for a lattice, find the shortest nonzero vector in the lattice $\mathcal{L}(\mathbf{B})$. The length is measured by the norm N (typically the Euclidean norm).

Closest Vector Problem (CVP) Given some basis $\mathbf{B}$ for a lattice, find the point p in the lattice ($p \in \mathcal{L}(\mathbf{B})$) such that it is the closest point to a target vector t (which is not necessarily in the lattice).

Shortest Independent Vector Problem (SIVP) Given some basis $\mathbf{B}$ for a lattice of dimension n, find a linearly independent set of n vectors such that all vectors are shorter or equal than the n-th successive minima.

These problems can be approximated and are typically known as SVP_γ, CVP_γ, and SIVP_γ. For SVP_γ the problem is to find a nonzero vector where the length is at most a multiple (γ) of the length of the shortest vector.

How lattice problems can be used for cryptography was first shown by Ajtai [1]. Ajtai created a family of one-way functions based on the SIS (Short Integer Solution) problem and showed that the average case of the SIS problem is at least as hard as solving the approximate SVP in the worst case. This is noteworthy because it implies that one must be able solve *any* instance of the approximate SVP in order to compromise the one-way function instead of only a subset of instances. It was later shown that these hash functions were also collision-resistant making them cryptographic hash functions [28].

`NTRU` [48], as described in the second round of the NIST competition (and not the original cryptosystem [32]), works on polynomial rings and does not have a formal proof that it relies on one of the problems above. However, lattices are still relevant to `NTRU` in that `NTRU` can be seen as lattices with special structure. As a result, attacks based on lattices such as lattice reduction are applicable to `NTRU`. In lattice reduction one wants to find a short basis with nearly orthogonal vectors when given a lattice with a different basis. One algorithm that accomplishes this in polynomial time is the Lenstra-Lenstra-Lovász (LLL) algorithm [40]. Since lattice reduction produces short vectors by design it can aid in solving the SVP. The parameter sets proposed account for the existence of these attacks and are chosen somewhat conservatively.

2.7 Side-channel resistance

When working with cryptography it is important to consider that it will end up running on hardware that can be susceptible to side-channel attacks. Side channel attacks allow adversaries to gain information about secret data through weaknesses of the implementation rather than the algorithm. Examples of side channels are timing, power consumption, electromagnetic radiation, fault behavior, and data remanence. A well-known side-channel example attack is square-and-multiply exponentiation in RSA. During decryption every bit in the private key corresponds to a squaring, whenever that bit is a 1 there is an additional multiplication. A power trace of the decryption will thus reveal whether a bit is a 0 or a 1 due to the difference in time (and power) it takes. In order to avoid this problem the exponentiation must be implemented in such a manner that the amount of time it takes is independent of the value of private key. The idea of writing code that separates execution time from secret data is known as constant-time code and is essential when implementing cryptography. For designers of cryptography side-channel attacks are typically not the main focus as it is in essence an implementation problem. However, it is beneficial to at least consider since a cryptographic scheme that is full of implementation pitfalls will result in practical problems and may harm performance if constant-time code is difficult to achieve.

2.8 Related work

Besides `NTRU` in the NIST competition (which was formed out of NTRU-Encrypt [31] and NTRU-HRSS-KEM [33]) there is also NTRU Prime [13]. NTRU Prime differs from `NTRU` in that the rings chosen avoid special structures present in other NTRU schemes and a prime q parameter is chosen instead of a power of two. NTRU Prime, understandably, has a lot of similarities to `NTRU` such as the need for constant-time sorting (given that the input length is always the same). Bernstein (one of the authors of NTRU Prime) created djbsort [11] that performs this constant time sorting. Due to the efficiency of this sorting method and the fact it was already heavily optimized it found its way into implementations of `NTRU`.

Other similarities of `NTRU` and NTRU Prime is that they both need to do inversion modulo 3 and modulo 2. Bernstein and Yang looked into fast gcd computation and modular inversion [14]. Their method (and code) for inversion modulo 3 outperformed the optimized code described in NTRU-HRSS-KEM by a factor of 1.7 and as a result ended up in the optimized code of `NTRU` after being adapted to the different parameters.

NTRU-HRSS-KEM was previously optimized [34] using AVX2 before being merged into `NTRU`. Many of the techniques used in the implementation were also applicable within `NTRU`.

Note that `NTRU` does not exactly contain NTRU-HRSS-KEM but a variant called HRSS-SXY. Saito, Xagawa, and Yamakawa [47] slightly tweaked NTRU-HRSS-KEM in order to eliminate a length-preserving message confirmation hash in the ciphertext. The result was a slightly more costly DPKE rather than a PPKE. The `NTRU` submission is slightly more efficient by eliminating a step in the decapsulation routine which offsets the cost of adding the DPKE a bit.

Overview of `NTRU`

In this chapter the general design of `NTRU` is described. The first section will cover the core operations used in `NTRU`. These are important to fully understand in order to understand the way `NTRU` works. Additionally, it is these core operations that benefit the most from heavy optimization. The other two sections will give an overview of the `NTRU` DPKE and KEM. A full definition of `NTRU` can be found in the NIST submission [48].

Note that the NIST submission has defined four different parameter sets. These are known as `hps2048509`, `hps2048677`, `hps4096821`, and `hrss701`. The first three come from the NTRUEncrypt submission while `hrss701` comes from the NTRU-HRSS-KEM submission. The parameter n, which determines the amount of coefficients in the polynomials is respectively 509, 677, 821, and 701. While parameter q, the modulus that is a power of two, is respectively 2048, 2048, 4096, and 8192. These instances were chosen to accomplish the security goals set for the NIST competition. The `hps` (Hoffstein, Pipher, Silverman) and `hrss` (Hülsing, Rijneveld, Schanck, Schwabe) variants are largely the same but differ in some details (such as the sample spaces and the Lift function). When important, the differences will be explicitly mentioned. Constants that are used for the variants can also be found in the NIST submission [48].

Additionally, it should be noted that the `NTRU` KEM is IND-CCA2 secure in the random oracle model and has a tight reduction from the OW-CPA security of the NTRU DPKE. Proven secure in the random oracle model means that the KEM is provably secure when every cryptographic hash function has been replaced by a random oracle. A random oracle is a black box that given an arbitrary input responds with a uniformly random output from its domain.

Before we can fully describe NTRU we must establish some notation that will be used in the description.

1. Φ_1 is the polynomial $(x-1)$.
2. Φ_n is the polynomial $(x^n-1)(x-1)$.
3. R is the quotient ring $\mathbb{Z}[x]/(\Phi_1\Phi_n)$.
4. S is the quotient ring $\mathbb{Z}[x]/(\Phi_n)$.
5. $R/3$ is the quotient ring $\mathbb{Z}[x]/(3,\Phi_1\Phi_n)$. Polynomials have $n-1$ coefficients with all coefficients in the set $\{-1,0,1\}$.
6. R/q is the quotient ring $\mathbb{Z}[x]/(q,\Phi_1\Phi_n)$. Polynomials have $n-1$ coefficients with all coefficients in the set $\{-q/2,-q/2+1,\ldots,q/2-1\}$.
7. $S/2$ is the quotient ring $\mathbb{Z}[x]/(2,\Phi_n)$. Polynomials have $n-2$ coefficients with all coefficients in the set $\{0,1\}$.
8. $S/3$ is the quotient ring $\mathbb{Z}[x]/(3,\Phi_n)$. Polynomials have $n-2$ coefficients with all coefficients in the set $\{-1,0,1\}$.
9. S/q is the quotient ring $\mathbb{Z}[x]/(q,\Phi_n)$. Polynomials have $n-2$ coefficients with all coefficients in the set $\{-q/2,-q/2+1,\ldots,q/2-1\}$.

Whenever one sees an operations followed by mod $(2,\Phi_n)$ then the operation is performed such that at the end one has $n-2$ coefficients and all coefficients are either 0 or 1. This is the case for any of the quotients rings described above except that some have $n-1$ coefficients and other values for those coefficients.

3.1 Important `NTRU` operations

3.1.1 Sampling

In some parts of NTRU there is a need to turn randomness (which is a uniformly random sequence of bytes) into polynomials. An example of this is during the key generation part since NTRU keys are polynomials. Going from randomness to polynomials is known as sampling in NTRU. There are two sampling methods in NTRU, they are: `sample_fg` and `sample_rm`. The first method outputs two polynomials f and g whereas the second method outputs two polynomials r and m. The exact algorithm used in the sampling depends on the variant of NTRU.

In order to illustrate this sampling let us take a look at `sample_fg` for `hps2048509`. It takes 4064 random bits (so 508 bytes) and transforms each byte into a value of -1, 0, or 1 by reducing modulo 3; in practice they are 0, 1, 2 but that is equivalent due to reduction with modulo 3. The result is

508 coefficients in the range of $\{-1, 0, 1\}$. The 509th coefficient is always set to 0. These 509 coefficients are called the f polynomial, which is a ternary polynomial (every coefficient in the range $\{-1, 0, 1\}$) with the values being almost evenly distributed (0 occurs slightly more often). A similar process happens for the g polynomial except that it is a ternary polynomial with exactly 127 coefficients being equal to 1 ($q/16-1$) and exactly 127 coefficients being equal to -1. The other $(509 - 127) - 127 = 255$ coefficients are 0. For `hrss` it would be similar except that the output polynomials also have a special property (non-negative correlation property).

Cryptographic sorting

The example above glossed over how exactly the g polynomial is created. We know that the g polynomial is a ternary polynomial with $q/16-1$ coefficients set to 1, $q/16-1$ coefficients set to -1, and the remaining coefficients set to 0. The most straightforward implementation would be to simply create such a polynomial and then randomly permute it by using an algorithm such as the Fisher-Yates shuffle [39]. However, implementing Fisher-Yates in such a way that there is no side channel is difficult. It must run in constant time which is possible but difficult to accomplish as the polynomials are too large to fit entirely into the processor cache. The result is that some subsets of the polynomial may be in the cache thus causing a timing difference if elements from the same part of the polynomial are picked twice. Additionally, picking random elements from the polynomial must have no bias.

An alternative approach taken by `NTRU` is to rely on sorting. Sorting also poses a security problem as sorting is dependent on the input values (assuming the length of the array is always the same). The duration of a sort might reveal something about the g polynomial such as the fact that this specific g polynomial has a larger amount of ones at the start than typical. This kind of information gives an attacker slight advantages when trying to break a cryptosystem. The difference between shuffling and sorting is that building a constant-time sorting method is easier to accomplish.

In `NTRU` a carefully designed and implemented sorting algorithm must be used. To fulfill this requirement an implementation can only use constant-time operations such as addition, subtraction, and logical operations. This implementation is also not allowed to use branches that rely on secret data (such as the values of the array) and must be careful about the effect of processor caches when accessing memory. These strict requirements seem problematic since they are difficult to achieve and will likely cost some performance but it can be done. This sorting is only required in the `hps` variants of `NTRU`.

3.1.2 Polynomial addition

Polynomial addition is one of the simpler operations present in `NTRU`. As the name suggest it involves adding two polynomials together. Since all polynomials in `NTRU` have the same length for a specific parameter set, addition is performed by simply adding each term together and then performing modular reduction on each resulting coefficient.

Note that `NTRU` coefficients have maximum values between 2047 (`hps2048509`) and 8191 (`hrss701`). One byte would not be sufficient to store these coefficients so two bytes are used in the reference implementation. The maximum value of a coefficient after addition and before modular reduction is $2047 + 2047 = 4094$ and $8191 + 8191 = 16382$. As a result most `NTRU` implementations do not have to worry about overflow since the result of addition still fits within two bytes. Implementations are free to choose sizes that are not multiples of a byte such as 12 bits for `hps2048509` and 14 bits for `hrss701` (or even smaller if overflow is dealt with separately). The expectation is that this is reserved for hardware implementations or for situations where memory usage is crucial (because a polynomial admittedly wastes a lot of space; especially for the smaller parameter sets).

3.1.3 Polynomial reduction

Unlike polynomial addition there are some polynomial operations that do result in polynomials with more coefficients. One such operation is polynomial multiplication where multiplication of two polynomials with n coefficients (degree $n-1$) results in a polynomial with $2n-1$ coefficients (degree $2n-2$)). Due to the rings `NTRU` operates on, all polynomials must have at most n coefficients. In order to achieve this we do an operation called polynomial reduction.

Terms above x^{n-1} are brought back down by subtracting n from the power and then adding the coefficient of the new term to the term that already occupies that place. One has to take care that the new value of the coefficient does not exceed the modulus. A different, perhaps easier, way to view this is that the resulting polynomial is split up into two parts where the upper part is added to the lower part using polynomial addition.

Let us take $n = 509$, $q = 2048$ and the following excerpt of a resulting polynomial as an example:

$$\cdots + 1021x^{510} + 2019x^{509} + 1151x^{508} + \cdots + 632x^2 + 15x^1 + 400$$

This polynomial is split in two and the upper part of the polynomial becomes:

$$\cdots + 1021x^1 + 2019$$

This is then added to lower part which becomes:

$$1151x^{508} + \cdots + 632x^2 + 1036x^1 + 2419$$

The last step is to ensure all coefficients are within the range $[0, 2047]$ due to $q = 2048$. In order to do this, every coefficient is reduced modulo q:

$$1151x^{508} + \cdots + 632x^2 + 1036x^1 + 371$$

This operation can be fairly expensive for large polynomials which is why it is often integrated with the computation of other operations. Take polynomial multiplication for example; there is no need for the resulting polynomial coefficients to have already been reduced if they are going to be reduced again during polynomial reduction. The first reduction of the coefficients can be deferred until the polynomial reduction. There is a risk here that an *overflow will happen* since polynomials are added where the coefficients might be larger than $q - 1$ (due to the deferment). However, this is not an issue in practice (especially the reference implementation) as the overflow that will happen is for 16-bit values. An overflow in 16-bit values is implicitly the same as reducing modulo 2^{16}. In all `NTRU` parameter sets there is no reduction that is larger than 2^{16} which means that implicit reduction modulo 2^{16} does not affect the final result.

3.1.4 Polynomial multiplication

Polynomial multiplication (with polynomial reduction) is an extremely important and frequent operation in `NTRU` and is thus the most important operation to fully understand in order optimize it. It was already briefly discussed during polynomial reduction but will be covered here in more detail together with a description of what common algorithms exist.

It is important to note that integers can be expressed as polynomials when a base is given. Take the number 2143 as an example. 2143 can be expressed as $2 \cdot 10^3 + 1 \cdot 10^2 + 4 \cdot 10^1 + 3 \cdot 10^0$ which is the polynomial $a(x) = 2x^3 + x^2 + 4x + 3$ evaluated at base 10 ($a(10) = 2143$). Given a polynomial it is thus also possible to express it as an integer. As a result of this fact, multiplication algorithms for integers can also work for our polynomial multiplication in `NTRU`. When describing the multiplication algorithms used in the implementation of `NTRU` they will assume we are multiplying polynomials since we have no need for conversion from and to integers.

Schoolbook multiplication

The schoolbook method, also known as long multiplication, for multiplying polynomials is the most common one and simply multiplies each term of one polynomial with all terms of the other polynomial. That means that for the multiplication of polynomials with n terms there will be n^2 multiplications. Besides the multiplications there are also some additions ($(n-1)^2$ to be exact) as some terms in the resulting polynomial may come from multiple multiplications. For example, $2x^5$ may come from the multiplication of $x^3 \cdot x^2$ and $x^2 \cdot x^3$. The resulting polynomial has $2n-1$ terms and the coefficients of those term may exceed the modulus. It is thus required to apply a polynomial reduction at the end in order for this to work in `NTRU` (and the coefficients must also be reduced). An example of the schoolbook method:

$$\begin{aligned}(10x^2+3x+1)(15x^2+x+1) =& 10x^2 \cdot 15x^2 + 10x^2 \cdot x + 10x^2 \cdot 1 + \\ & 3x \cdot 15x^2 + 3x \cdot x + 3x \cdot 1 + \\ & 1 \cdot 15x^2 + 1 \cdot x + 1 \cdot 1 \\ =& 150x^4 + 10x^3 + 10x^2 + 45x^3 + \\ & 3x^2 + 3x + 15x^2 + x + 1 \\ =& 150x^4 + 55x^3 + 28x^2 + 4x + 1\end{aligned}$$

While the complexity of the schoolbook method is $\mathcal{O}(n^2)$ it is still a relevant algorithm as there is no overhead/setup cost. This is especially relevant when multiplying smaller polynomials as some algorithms require an up-front computation negating their benefit unless the input is sufficiently large.

Karatsuba multiplication

Karatsuba's algorithm for multiplication was discovered by Karatsuba [37] and requires $n^{\log_2 3}$ multiplications. The simple case in Karatsuba is for polynomials with a degree of 1. Given the following two polynomials:

$$\begin{aligned}A(x) &= a_1x + a_0 \\ B(x) &= b_1x + b_0\end{aligned}$$

The following temporary values can be calculated:

$$\begin{aligned}D_0 &= a_0b_0 \\ D_1 &= a_1b_1 \\ D_{0,1} &= (a_0+a_1)(b_0+b_1)\end{aligned}$$

From these temporary values the resulting polynomial $C(x)$ can be calculated as follows:

$$C(x) = D_1x^2 + (D_{0,1} - D_0 - D_1)x + D_0$$

One can see that only three multiplications are required rather than four but this comes at the cost of four additions instead of one. Despite the additional additions Karatsuba's algorithm is still considered fast due to the fact that addition is linear in time while multiplication is not.

In order to apply Karatsuba on larger polynomials it is used in a recursive fashion. This is known as recursive Karatsuba and is the reason why it is considered a divide and conquer algorithm. Larger instances of the problem are split up into smaller instances and then the algorithm is applied again until the easiest case is reached (typically when schoolbook multiplication becomes the best option or when dedicated processor instructions are available).

The application of recursive Karatsuba is very similar to that of degree-1 Karatsuba. An example will illustrate the process. Imagine $A(x)$ and $B(x)$ to be degree-255 polynomials (thus 256 coefficients). The result is a polynomial $C(x)$ with degree 510 (511 coefficients). $A(x)$ and $B(x)$ are both split into upper and lower halves resulting in four 128-coefficient polynomials (similar to creating a_1, a_0, b_1, and b_0). These are called $A_u(x)$, $A_l(x)$, $B_u(x)$, and $B_l(x)$. The temporary values are now computed as:

$$\begin{aligned} D_0 &= A_l B_l \\ D_1 &= A_u B_u \\ D_{0,1} &= (A_l + A_u)(B_l + B_u) \end{aligned}$$

These multiplications are computed by using Karatsuba again but this time degree-127 polynomials are multiplied. This process is repeated until the degree is one such that degree-1 Karatsuba is applicable. In practice the degree does not go down to one as it is more efficient to use schoolbook multiplication at some point.

D_0, D_1, and $D_{0,1}$ are 256-coefficient polynomials and must be recombined to form $C(x)$. D_1 is used for the upper 256 coefficients of $C(x)$ while D_0 is used for the lower 256 coefficients. Here there is a small difference from degree-1 Karatsuba. We still compute $D_{0,1} - D_0 - D_1$ but it has to be placed in the middle such that there are 128 coefficients to the left and the right. Since there are already 256 coefficients in the middle (128 from D_1 and 128 from D_0) we must add these first before placing the last 256 coefficients in the middle. The result is a degree-511 polynomial $C(X)$ that is the multiplication of $A(x)$ and $B(x)$.

In this example the polynomials are of degree $2^i - 1$ in order to minimize the edge cases. In reality Karatsuba can be applied to polynomials on any degree but requires more effort. An excellent reference which contains many forms of Karatsuba's algorithm is the paper *Generalizations of the Karatsuba Algorithm for Efficient Implementation* by Weimerskirch and Paar [55].

Toom-Cook multiplication

Another example of a divide and conquer algorithm is Toom-Cook multiplication, discovered by Toom and improved by Cook. Toom-Cook is similar to Karatsuba in that it splits the polynomial in parts such that the multiplications are of smaller degree. In fact, Karatsuba turns out to be a special case of Toom-Cook where $k = 2$ (two parts). The complexity of Toom-Cook is described by $\Theta(c(k)n^{\log(2k-1)/\log(k)})$. The term $c(k)$ describes the time spent on additions and multiplications of small constants. One can see that an increasing k leads to smaller powers but in practice mostly Toom-Cook 3-way and 4-way are used as anything above that becomes difficult to use due to a rapidly increasing $c(k)$ [56]. Toom-Cook is typically used when the numbers/polynomials that are multiplied are so large that the speedup over Karatsuba is worth it.

The computation of Toom-Cook has five separate steps which are:

1. Splitting
2. Evaluation
3. Pointwise multiplication
4. Interpolation
5. Recomposition

The first step is converting the integers that will be multiplied into k part polynomials and the last step does the same but in reverse. Since `NTRU` fully works on polynomials there is no need for these steps. Splitting the polynomials into k parts is done in a similar fashion as in Karatsuba. If $k = 4$ and the polynomials have 256 coefficients then they are simply split up into four polynomials of 64 coefficients. So $A(x)$ would become $p(x) = m_3x^3 + m_2x^2 + m_1x + m_0$ and $B(x)$ would become $q(x) = n_3x^3 + n_2x^2 + n_1x + n_0$. Instead of integers, m_i and n_i are polynomials of 64 coefficients.

The evaluation step in Toom-Cook is fairly clever as it uses the fact that $r(x) = p(x)q(x)$ for any value of x. In our example of $k = 4$ we know that $r(x)$ (resulting polynomial) will have a degree of 6. If one has 7 (degree + 1) points on a polynomial then one can find the unique polynomial belonging to those points. So the evaluation step computes $deg(p) + deg(q) + 1$ points on $p(x)$ and $q(x)$ so that in the next step we can find $deg(p) + deg(q) + 1$ (7 when $k = 4$) points on $r(x)$. The points chosen are typically small as they are easy to compute (they contribute to the c factor mentioned before). One special point commonly used is $x = \infty$ as it is equivalent to asking for the limit of the polynomial which is always the highest coefficient. Some point evaluation examples for $p(x)$ that are not necessarily the fastest way of computing them ($q(x)$ is equivalent):

$$\begin{aligned}
p(0) &= m_3(0)^3 + m_2(0)^2 + m_1(0) + m_0 &&= m_0 \\
p(1) &= m_3(1)^3 + m_2(1)^2 + m_1(1) + m_0 &&= m_3 + m_2 + m_1 + m_0 \\
p(-2) &= m_3(-2)^3 + m_2(-2)^2 + m_1(-2) + m_0 &&= -8m_3 + 4m_2 - 2m_1 + m_0 \\
&\dots \\
p(\infty) &= \dots &&= m_3
\end{aligned}$$

Note that m_i and n_i were polynomials so those additions and multiplications with small numbers are operations on polynomials. However, they are inexpensive operations compared to multiplication. Another observation is that these point evaluations can also be viewed as a matrix-vector multiplication. This is important for the interpolation step later.

The pointwise evaluation step is fairly simple in that it computes $r(x) = p(x)q(x)$ for $deg(p) + deg(q) + 1$ points. Taking the polynomials above there would be 7 multiplications of 64-coefficient polynomials. This is the most expensive part of Toom-Cook but is still more efficient than computing the multiplication of two 256-coefficient polynomials using (recursive) Karatsuba or schoolbook multiplication. Since the multiplications of these smaller polynomials may still be relatively large it is possible to apply Toom-Cook, Karatsuba, or schoolbook multiplication again. This is essentially the same idea as recursive Karatsuba.

Interpolation is the last relevant step in Toom-Cook. Given the points on $r(x)$ it finds the coefficients of $r(x)$. To accomplish this the idea is to essentially do the evaluation step in reverse. It is for this reason that viewing evaluation as a matrix-vector multiplication is important as we can simply multiply by the inverse of the matrix to get back to our vector (which are the coefficients of our polynomial $r(x)$). The points chosen for evaluation are chosen in such a way that a matrix inverse exists. Finding $r(x)$ is thus simply a matrix multiplication. Using a matrix inverse for polynomial interpolation is just one of the many different techniques (Gaussian elimination is another). Recall that polynomials were initially split which means that the polynomials contained within $r(x)$ must be recombined to form a single polynomial. Similar to Karatsuba there will be some overlap of these parts that must be dealt with, this is normally part of the recomposition step.

3.1.5 Polynomial inversion

Multiplying a polynomial and its inverse results in 1 which effectively cancels out a polynomial. It is for this reason that inversion is used a few times throughout `NTRU`. Finding the inverse is a fairly slow operation which is why it benefits from optimizations.

In the reference implementation of `NTRU` there are two separate algorithms used for computing the inverse. The first algorithm is the "Almost Inverse Algorithm" [49] and is used for finding a polynomial close to the inverse polynomial. It is not quite the inverse (hence Almost Inverse) but each coefficient is a distance a removed from the actual value. This is sufficient for `NTRU` to work. Since the algorithm itself does not run in constant time it was adapted in the implementation of `NTRU` such that it is. The other algorithm is first computing the inverse modulo 2 (using the Almost Inverse algorithm) and then bringing it back to inverse modulo q by using a variant of Newton iteration [52].

In the optimized implementation, the Almost Inverse algorithm is no longer used so we will not discuss that algorithm here. Instead, the optimized implementation uses three distinct algorithms for inversion. The implementation of those algorithms differs per specific parameter set and as such they will be discussed later in Chapter 4.

3.1.6 Hashing

Cryptographic hashing is used in the `NTRU` KEM during the encapsulation and decapsulation process. For all parameter sets there is only one hash function and that is SHA3-256. The implementation is provided by NIST for the competition such that all submissions have access to the same SHA3-256 performance. It is undesirable that some submission performs better due to a better implementation of their hash function as the implementation of this hash function is in theory available to all submissions. Submissions are free to use an alternative implementation or other hash functions if they so desire. Providing an implementation of SHA3-256 also provides incentive to use a standard hash function rather than one built specifically for the cryptosystem. Since the hash function is provided there is little gain in looking at the implementation of the hash function. It is sufficient to know that the cryptographic hash function takes arbitrary length input and outputs 256 bits.

3.2 NTRU DPKE

3.2.1 Keypair generation

The keypair generation for the NTRU DPKE is described in Algorithm 1. The input *seed* is typically random such that the keypair generated is random. Reusing the *seed* results in the same keypair since it is deterministic. The output **h** is the public key and the tuple $(\mathbf{f},\mathbf{f}_p,\mathbf{h}_q)$ is the private key. Since the public and private key consist out of polynomials they are converted to byte arrays (known as *packing*) before being exchanged or stored. This makes it possible for implementations of NTRU to be interoperable as the internal format to store polynomials may be different across implementations.

Let us take the polynomial f as an example of the packing routine. Polynomial f lives in $S/3$ which means every coefficient has a value in $\{0, 1, 2\}$. When packing this polynomial it is possible to put 5 coefficients into a single byte. Imagine 5 coefficients all having the maximum value of 2. First we put one coefficient in a byte which now has the value of 2. Then we multiply this byte with 3 and add the next coefficient which makes for a value of $2 \cdot 3 + 2 = 8$. We multiply with 3 again and add another coefficient which has the result $8 \cdot 3 + 2 = 26$. Note that this is equivalent to $2 \cdot 9 + 2 \cdot 3 + 2 = 26$. After doing this for all 5 coefficients the byte now has the value $2 \cdot 81 + 2 \cdot 27 + 2 \cdot 9 + 2 \cdot 3 + 2 = 242$. Any combination of 5 coefficients will result in a unique byte value which is why it is possible to store 5 coefficients in a single byte. The unpacking routine works the same but in reverse. Note that the time division takes is dependent on the values which means it is not safe to use for implementation due to creating a side channel. Instead, division is performed by multiplication and logical shifts.

Algorithm 1 KeyGen'(*seed*)

```
1: $(\mathbf{f},\mathbf{g}) \leftarrow$ Sample_fg($seed$) with $\mathbf{f},\mathbf{g} \in S/3$
2: $\mathbf{f}_q \leftarrow (1/\mathbf{f}) \bmod (q, \Phi_n)$
3: $\mathbf{h} \leftarrow (3 \cdot \mathbf{g} \cdot \mathbf{f}_q) \bmod (q, \Phi_1\Phi_n)$
4: $\mathbf{h}_q \leftarrow (1/\mathbf{h}) \bmod (q, \Phi_n)$
5: $\mathbf{f}_p \leftarrow (1/\mathbf{f}) \bmod (3, \Phi_n)$
6: return $((\mathbf{f}, \mathbf{f}_p, \mathbf{h}_q), \mathbf{h})$
```

For efficiency reasons some of these operations may be swapped or computed differently. For example, $\mathbf{f}_p$ can already be computed between step 2 and 3 and then packed. Another example is not computing $\mathbf{f}_q$ and using $\mathbf{f}$ instead in step 3 and call it **h'**. Then after step 4 (in which **h'** and mod $(q, \Phi_1\Phi_n)$ were used) you first multiply $\mathbf{h'}_q$ with $\mathbf{f}$ in mod $(q, \Phi_1\Phi_n)$ in order to cancel out the previous $\mathbf{f}$. Then you multiply by $\mathbf{f}$ modulo (q, Φ_n). The result is identical to $\mathbf{h}_q$. Since $\mathbf{h'} \neq \mathbf{h}$ we also need to multiply $\mathbf{h'}_q$ with $3 \cdot \mathbf{g}$ twice such that we get **h** and not **h'**. This whole chain of operations avoids one very costly inversion for the cost of four additional multiplications.

3.2.2 Encryption

The encryption takes two inputs. The first input is the public key **h** while the second input is the plaintext (length of supported plaintext can be found in the specification [48]). Both of these are packed so they are unpacked first before being used. Public key **h** was packed into bytes and is unpacked as R/q while the plaintext was packed into bytes and is unpacked as $S/3$. The output **c** is the ciphertext in R/q and is packed into bytes.

Algorithm 2 Encrypt(**h**, (**r**, **m**))

```
1: $\mathbf{m'} \leftarrow \text{Lift}(\mathbf{m})$
2: $\mathbf{c} \leftarrow (\mathbf{r} \cdot \mathbf{h} + \mathbf{m'}) \bmod (q, \Phi_1\Phi_n)$
3: return $\mathbf{c}$
```

The Lift operation does as it suggests and takes it takes a polynomial from one domain to another domain. For `hps` Lift is simply ensuring the polynomial is in $S/3$ (which is the identity operation since unpacking already returns a polynomial in $S/3$). For `hrss` input polynomial **m** has the following operation applied: $\Phi_1 \cdot \text{S3}(\mathbf{m}/\Phi_1)$. Like before, these operations may be reordered for efficiency as long as the output **c** does not change.

3.2.3 Decryption

Decryption takes the packed private key as input which is unpacked before being used. Additionally it takes the packed ciphertext **c** as input. The output depends on whether decryption succeeds or not. If the decryption succeeds then the output is a tuple (**r**, **m**, 0) which contains the plaintext. Otherwise it is the tuple (0, 0, 1).

Algorithm 3 Decryption((**f**, $\mathbf{f}_p$, $\mathbf{h}_q$), **c**)

```
1: if $\mathbf{c} \not\equiv 0 \pmod{(q, \Phi_1)}$ return (0, 0, 1)
2: $\mathbf{a} \leftarrow (\mathbf{c} \cdot \mathbf{f}) \bmod (q, \Phi_1\Phi_n)$
3: $\mathbf{m} \leftarrow (\mathbf{a} \cdot \mathbf{f}_p) \bmod (3, \Phi_n)$
4: $\mathbf{m'} \leftarrow \text{Lift}(\mathbf{m})$
5: $\mathbf{r} \leftarrow ((\mathbf{c} - \mathbf{m'}) \cdot \mathbf{h}_q) \bmod (q, \Phi_n)$
6: if $(\mathbf{r}, \mathbf{m}) \in \mathcal{L}_r \times \mathcal{L}_m$ return (r, m, 0)
7: else return (0, 0, 1)
```

Note that the algorithm for decryption says to return whenever something goes wrong. In reality this is not the case as it leads to a timing side channel.

Imagine some adversary modifies **c** but still ensures step 1 passes. This is possible since no secret information is used for that step. During step 6 if we return immediately an adversary can tell whether **r**, **m**, or both are in the sample spaces. Allowing this timing channel to exist is giving the adversary unnecessary advantages. Thus, in practice, a *fail* flag is set and the execution is always continued until the end.

3.3 NTRU KEM

3.3.1 Keypair generation

The NTRU KEM keypair generation is essentially the same as the DPKE keypair generation except that an additional uniformly random 256-bit bitstring **s** is generated and added to the private key. This bitstring **s** is used during the decapsulation process.

Algorithm 4 KeyGen(*seed*)

```
1: $((\mathbf{f}, \mathbf{f}_p, \mathbf{h}_q), \mathbf{h}) \leftarrow$ KeyGen'($seed$)
2: $\mathbf{s} \leftarrow_{\$} \{0,1\}^{256}$
3: return $((\mathbf{f}, \mathbf{f}_p, \mathbf{h}_q, \mathbf{s}), \mathbf{h})$
```

3.3.2 Encapsulation

Different from encryption the encapsulation process only takes the public key as input as the key. The encapsulation process takes 256 uniformly random bits known as *coins* and uses it as the source for the polynomials **r** and **m**. These are then used as the "plaintext" for the encryption routine. Note that the encryption routine takes a packed plaintext as input which means (**r**, **m**) is immediately packed and unpacked; this can be avoided during implementation. The hash of the tuple (**r**, **m**) is the key that is generated in this KEM. The idea is that the other party can decrypt the ciphertext such that it receives (**r**, **m**) and then hashes it to generate the same key.

Algorithm 5 Encapsulate(**h**)

```
1: $coins \leftarrow_{\$} \{0,1\}^{256}$
2: $(\mathbf{r}, \mathbf{m}) \leftarrow$ Sample_rm($coins$) with $\mathbf{r}, \mathbf{m} \in S/3$
3: $\mathbf{c} \leftarrow$ Encrypt($\mathbf{h}$, ($\mathbf{r}$, $\mathbf{m}$))
4: $\mathbf{k} \leftarrow H_1(\mathbf{r}, \mathbf{m})$
5: return $(\mathbf{c}, \mathbf{k})$
```

3.3.3 Decapsulation

The decapsulation process decrypts the ciphertext and if it fails it returns a pseudorandom key instead of the actual key. In NTRU-HRSS-KEM an error symbol was produced but in HRSS-SXY this was changed to produce a pseudorandom key. The idea of producing a pseudorandom key instead of an error symbol is known as implicit rejection and ensures that actual usage of the key will fail.

Algorithm 6 Decapsulate(($\mathbf{f}$, $\mathbf{f}_p$, $\mathbf{h}_q$, $\mathbf{s}$), $\mathbf{h}$)

```
1: ($\mathbf{r}$, $\mathbf{m}$, $\mathit{fail}$) $\leftarrow$ Decrypt(($\mathbf{f}$, $\mathbf{f}_p$, $\mathbf{h}_q$), $\mathbf{c}$)
2: $\mathbf{k}_1 \leftarrow H_1(\mathbf{r}, \mathbf{m})$
3: $\mathbf{k}_2 \leftarrow H_2(\mathbf{s}, \mathbf{c})$
4: if $\mathit{fail} = 0$ return $\mathbf{k}_1$
5: else return $\mathbf{k}_2$
```

The parameter $\mathbf{s}$ in the private key is needed in the generation of the pseudorandom key in order to ensure that every keypair produces a different pseudorandom key for a specific ciphertext. Generating a uniformly random $\mathbf{s}$ during the keypair generation prevents having to do this every time decapsulation occurs.

Optimizing `NTRU` using AVX2

The goal of this thesis was to optimize `NTRU` using AVX2. One of the many considerations in the NIST competition is the performance of candidates. In order to get a better understanding of the real-world performance it is necessary to have optimized implementations, because reference implementations tend to not have a focus on performance. The belief was that an AVX2 implementation would be the fastest achievable software implementation of `NTRU` on recent Intel processors. An AVX-512 implementation could potentially be faster but is far from common even on the newest processors. Note that the focus was on performance and as such other implementations could do better in the categories of memory usage, binary size, and power consumption.

Since the `hrss701` parameter set came from the NTRU-HRSS-KEM submission and was already optimized using AVX2 [34], the target of this Master's thesis was initially the `hps` parameter sets. After some reconsideration (mostly wanting to see the limit of `NTRU`) the thesis restricted itself to the `hps2048509` parameter set. The optimizations performed are applicable on other parameter sets but were not implemented.

The most costly operations were suspected to be the polynomial operations and these suspicions were confirmed by profiling the execution of `NTRU` (using `perf`). The optimizations performed are discussed in the following sections. The reference implementation generates Known Answer Test (KAT) values such that other implementation can verify their behavior against the reference implementation. The KATs were used to confirm the optimized implementation of `hps2048509` behaves identical to the reference implementation.

4.1 AVX2 features and description

AVX is an instruction-set extension for x86(-64) processors which adds 256-bit vector registers and instructions to operate on such registers (AVX2 adds more instructions). These registers are called vector registers since a single instruction operates on multiple data elements in the register. This concept is known as single instruction multiple data (SIMD).

Since `NTRU` coefficients are 16 bits in size it is possible to store 16 coefficients in one AVX2 register and do a single instruction that operates on 16 coefficients at a time. An example of where this would be faster is in polynomial addition. A naive implementation would loop through all coefficients (all 509 of them) and add them one by one. Each addition requires loading a pair of coefficients from memory and storing the result back to memory. Optionally a modular reduction can be applied depending on whether the polynomial has to be in the ring again. An implementation using AVX2 would be able to load (and store) 16 coefficients at a time ($\lceil \frac{509}{16} \rceil = 32$ iterations in total) and do a `vpaddd` instruction which adds them together. This can be followed by a vector logical AND instruction to perform the modular reduction. For processors where the vector instructions are fast enough (which is practically all AVX2-capable processors) this will outperform the non-AVX2 implementation.

In reality, compilers can heavily optimize the first implementation in the example above using many different techniques such that the gap between non-AVX2 and AVX2 is much smaller (but still exists). In fact, some compilers might even generate AVX2 instructions to perform the first implementation. This is known as auto-vectorization and would be ideal. Unfortunately, auto-vectorization is unreliable and there are many cases where a compiler would not vectorize while a programmer would. It is for this reason that an AVX2 implementation is beneficial to write.

While manually writing assembly or intrinsics, a programmer must keep a few facts in mind. Firstly, some registers are callee-saved which means that at the end of a function they must be the same as before the function started. This requires these registers to be stored somewhere else temporarily (such as in memory) or to not use them.

Secondly, the ordering of instructions is important in order to avoid long dependency chains where instructions are waiting on the result of other instructions. This is less of a problem when working on out-of-order execution processors.

Thirdly, some instructions have a higher throughput than others. Throughput is how many instructions of the same kind can be executed per clock cycle. Also, some instructions have a longer latency which is the amount of cycles since the start when the result is available, this is important in dependency chains. These numbers may vary wildly per processor. An example of this are the `pext` and `pdep` instructions which perform well on Intel processors while on AMD Zen-based processors they perform significantly worse (a factor 18 difference). A programmer must be aware of this in order to produce fast assembly. The instruction tables by Fog [25] are an extremely good resource on this front.

Finally, AVX2 instructions consume a fair amount of power and thus generate a lot of heat. A processor typically compensates for this by lowering the clock speed and turning off the AVX2 part of the core when not in use [41]. This can come at a performance penalty if the programmer does not constantly feed the core with enough AVX2 instructions due to the time required for clock speeds to adjust and the AVX2 part turning on [26].

4.2 Optimizing crucial operations

For the optimizations below we typically assume that the polynomials have $n = 512$ and not $n = 509$ as it makes it easier to implement in AVX2 if we can work on a multiple of 16 coefficients (each coefficient being 16 bits). Afterwards some correction is needed to account for the fact that three additional coefficients are used. The internal representation using $n = 512$ should not affect the outcome of the computation (compared to using $n = 509$); we always work with the top coefficients set to zero.

The operations in `NTRU` were optimized until profiling showed that only the optimized implementations remain significant. At that point the only way to gain a noticeable speedup is to find other techniques to improve the performance of the optimized implementations even more.

The code of the AVX2 implementation can be found on GitHub in the `NTRU` repository [54]. It will appear in the `NTRU` package for the NIST submission at a later time.

4.2.1 Multiplication in R/q

Multiplication in R/q is the most common multiplication in `NTRU` and is the function that has received the most optimization. In this multiplication we want to multiply two 512-coefficient polynomials in R/q which will result in a single 1023-coefficient polynomial (stored as 1024 coefficients to make it easier). Instead of immediately reducing modulo 2048 on every coefficient we wait until we perform polynomial reduction first such that we only have to reduce modulo 2048 once and only on 512 coefficients and not on 1024 coefficients. The three extra coefficients are masked out at the start such that they have no influence on the final polynomial. After reduction the polynomial has 512 coefficients and not 509 so there is some correction for that as well.

The multiplication is implemented using Toom-Cook 4-way, Karatsuba multiplication, and schoolbook multiplication. These algorithms were described in detail in Section 3.1.4 so we will assume one is familiar with them. Toom-Cook 8-way was briefly considered but was not chosen as there would be too much overhead from evaluating and interpolating 15 points. We will now describe the sequence of operations that was used for the full multiplication.

Toom-Cook 4-way is applied such that there are 7 multiplications of 128-coefficient polynomials. These multiplications of 128-coefficient polynomials are further broken down with two levels of Karatsuba multiplication for a total of $7 \cdot 3 \cdot 3 = 63$ multiplications of $((512/4)/2)/2 = 32$-coefficient polynomials. The evaluation of points is combined with Karatsuba in the implementation and the 7 points used for evaluation are 0, ∞, 1, -1, 2, -2, and 3.

The 63 multiplications of 32-coefficient polynomials can be viewed as a 64 by 32 matrix. Transposing this matrix, in blocks of 16×16 because that is the maximum amount of coefficients 16 AVX2 registers will fit, allows the multiplication to be performed in a vectorized fashion. Transposing is necessary as coefficients are sequential in memory and would thus be sequentially stored in AVX2 registers while we would like each coefficient to be in a separate AVX2 register. If each coefficient is in a separate AVX2 register then it is possible to do operations on entire registers in order to perform operations for many polynomials in one instruction. Applying Toom-Cook 4-way twice instead of following it up with two levels of Karatsuba was also considered but meant that there would be 49 multiplications of 32 coefficients which would still result in a matrix of 64 by 32. In addition there would be another 7 point evaluations and more complex interpolation.

Instead of 63 multiplications of 32-coefficient polynomials, the implementation does three more levels of Karatsuba after transposing which has the result that we would be multiplying 4-coefficient polynomials. This is sufficiently small to compute using schoolbook multiplication without having to write to memory at all during the computation. Once all 4-coefficient polynomial multiplications are completed the implementation reconstructs the 256-coefficient polynomials, of which there are 7, that result from the pointwise multiplication step. Interpolation is performed by multiplying with the inverse matrix of the matrix we used when evaluating points. While this inverse matrix has many fractions implying that we need to do floating point arithmetic it is entirely possible to do this with integer arithmetic. Finding such a sequence of steps that was reasonable to implement was done in the implementation of NTRU-HRSS-KEM [34] and was reused in this implementation. During interpolation, the polynomial reduction is also applied and then finally the reduction modulo 2048 is performed. The result is a multiplication in R/q.

In order to assess how well this strategy performs we ideally want to compute a lower bound on the required CPU cycles. Computing a completely accurate lower bound is tricky in large polynomial multiplication as there are many combinations of multiplication algorithms that can be used. As mentioned above, other combinations were considered and none of them seemed better but it is by no means exhaustive and it is possible that some combination will outperform this combination *slightly*. It is possible to do multiplication using the number-theoretic transform (NTT) but due to the choice of parameters for `NTRU` (q being a power of two and the degree of the polynomials being prime) this was not beneficial [34, 13].

What *is* possible to do is calculate how close the implementation is to the lower bound of this combination of algorithms. To do this we start from the inside out as it makes reasoning easier. In the end all the multiplications boil down to $7 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 = 1701$ multiplications of 4-coefficient polynomials. Multiplying 4-coefficient polynomials requires $4^2 = 16$ multiplications of coefficients for a total of $1701 \cdot 16 = 27216$ coefficient multiplications. Since a coefficient is 16 bits and AVX2 is used, it is possible to do $256/16 = 16$ multiplications in one instruction (since 16 fit in a register). The resulting coefficients are 32 bits but we only keep the lower half so that the multiplication results fits into a single AVX2 register. This works as it is implicitly reducing modulo 2^{16} which is a multiple of q. As a result there is a theoretical minimum requirement of $27216/16 = 1701$ AVX2 `vpmullw` instructions. However, this is purely the multiplication instruction. Every multiplication requires two source registers with 16 coefficients each. To get these coefficients in registers they must be loaded from memory. Recall that the last operation is a Karatsuba multiplication which means that one load can be used for three multiplications and there are thus $1701/3 = 567$ memory loads (alignment is assumed here thus we use `vmovdqa`). The 1701 multiplications would produce $1701 \cdot 7 = 11907$ coefficients which would need $11907/16 = 744.1875$ aligned stores. Besides multiplications, loads, and stores there are also additions that must be accounted for. There are 1701 multiplications that are computed using schoolbook multiplication. Each schoolbook multiplication of 4-coefficient polynomials has $(4-1)^2 = 9$ coefficient additions. The result is a total of $1701 \cdot 9 = 15309$ coefficient additions. Due to AVX2 we can do these in batches of 16 coefficients for a total of $15309/16 = 956.8125$ `vpaddw` instructions. In addition to the coefficient additions in the schoolbook multiplication there are also additions due to Karatsuba. At the end there are 567 levels of Karatsuba where each instance has 4 additions. Two of these additions are additions of 4-coefficient polynomials while the other two are subtractions (which is essentially an addition) of 8-coefficient polynomials. We require $567 \cdot 2 \cdot 4 + 567 \cdot 2 \cdot 8 = 13608$ coefficient additions in Karatsuba. This is an additional $13608/16 = 850.5$ `vpaddw` instructions. At this point the estimation of the lower bound is detailed enough to get a view of the implementation. The fact that other Karatsuba levels also require additions and potential loads and stores is ignored as they are a minor amount of the total cycles and are for more difficult to estimate. Using the instruction tables [25] for Skylake on the calculations above results in a theoretical lower bound of $1701 \cdot 0.5 = 850.5$ cycles. This number assumes perfect instruction scheduling, that instructions never block other instructions due to not enough execution units being available, and that instructions do not depend on each other. The lowest bound is thus the longest sequence of a single instruction type because all other instructions types could have been executed in the meantime due to processors being out-of-order. In reality this will never be the case but quantifying how many instructions are blocked is nigh impossible. Assuming that instruction types do not execute out-of-order will lead to a "lower" bound that is higher than reality. Experimentally, we measured a median amount of 1418 cycles over 10000 iterations without function-call overhead. Given the fact some instructions are left out of the lower bound and that the implementation performs $64 \cdot 3 \cdot 3 \cdot 3 = 1728$ multiplications it seems the implementation is very close to the lower bound for the 64-way parallel multiplication.

Point evaluation takes about 156 cycles, transposing cost about 1222 cycles, and the interpolation and recomposition cost about 743 cycles. As a result, one multiplication in R/q costs about 3539 cycles. The cycle count mostly consists of transposes and the 64-way parallel multiplication. We have seen that the 64-way parallel multiplication is near the lower bound and can not be improved much further. Transposes are necessary when data is continuous in memory and vectorization is wanted. The implementation does a fairly standard sequence of vector unpacks followed by vector insertions and vector permutes. It is unlikely significant speedup can be gained there. As a result, if one wanted to speed up the multiplication in R/q even more it seems that a different approach must be taken.

4.2.2 Multiplication in $S/3$

For the multiplication in $S/3$ the multiplication in R/q is reused. It turns out that the coefficients of the polynomial can have a maximum value of $509 \cdot 4 = 2036$ after all the multiplications and summations. This is just below the 2048 that is used as modulus in the multiplication in R/q and thus has no effect on the values of the coefficients. After multiplying in R/q one would extract the last coefficient, double it, and then broadcast it across the vector such that each word (16 bits; the size of one coefficient) has the doubled last coefficient. This vector is then added to 16 coefficients at a time and then each coefficient is reduced modulo 3. The result is multiplication in $S/3$. The last step is equivalent to the reference implementation except vectorized using AVX2. This is an easy way of implementing multiplication in $S/3$ but it is not the fastest way. As expected, the performance of this implementation is nearly identical to that of multiplication in R/q except that about 124 cycles are added for a total of around 3663 cycles.

An alternative method that would be faster is to do Karatsuba recursively five times such that one has $3^5 = 243$ multiplications of 16-coefficient polynomials. These coefficients occupy two bits each (their values were in $\{-1, 0, 1\}$) which means a bitsliced implementation is possible for the multiplication of the 16-coefficient polynomials. In a bitsliced implementation one vector would hold one bit while another vector would hold the other bit [16]. For multiplication of the coefficients one would operate on entire AVX2 registers. Since each vector holds one bit per coefficient one can put 256 coefficients in a single AVX2 vector. Due to the coefficients being stored in 16 bits despite only occupying two bits there would have to be some pre- and post-processing going from 16-bit coefficients to 2-bit coefficients and vice versa. Additionally, some transposing would have to happen since the 2-bit coefficients would be continuous in memory. Similar to multiplication in R/q these operations would take a good chunk of the total cycle count.

Multiplication in $S/3$ using the first implementation consumes about 15% of the total time spent in decryption and as such the alternative implementation can never provide more than a 15% speedup for decryption (and that would only be the case if it cost 0 cycles). Given that a decent amount of cycles would be consumed by transposing/processing in addition to performing the actual multiplications it seems likely that the alternative implementation would only speedup decryption with five to ten percent. While faster, it is considerably more work to implement for very little gain. This is the only performance optimization known that has been left out.

4.2.3 Inversion in S/q

In order to perform fast inversion in S/q, the implementation does two separate steps. The first step is to find an inverse in $S/2$ which turns out to be very efficient. The second step is to bring the inverse in $S/2$ to an inverse in S/q by doing eight multiplications in R/q. Why and how this exactly works in described in NTRU Technical Report #014 [52]. We will focus here on the inversion in $S/2$ (and the multiplication in $S/2$ that is necessary) since multiplication in R/q was already optimized. For the inversion in $S/3$ a different algorithm is applied as this technique does not apply there. The algorithm used for inversion in $S/3$ is not faster than using this technique [14].

In order to perform the inversion in $S/2$ we start with the observation that $f^{2^{n-1}-1} \equiv 1 \pmod{(2, \Phi_n)}$ [35]. Equivalently we can say that $f^{2^{508}-2} \equiv f^{-1}$

(mod $(2, \Phi_{509})$). Implementing this exponentiation using an addition chain of 1, 2, 3, 6, 12, 15, 30, 60, 63, 126, 252, 504, 507 takes 12 multiplications in $S/2$ and 13 multi-squarings. This addition chain is the shortest addition chain possible [24] but there may be other addition chains with an equal length. The difference in performance is negligible as they tend to have about the same step size. The polynomials in `NTRU` use 16 bits per coefficient but when working in $S/2$ this is unnecessary as every coefficient is simply a bit. Polynomials in $S/2$ can thus be represented as a bitstring which is beneficial when trying to square or multiply polynomials in $S/2$. This conversion process uses the `pdep` and `pext` instructions; recent AMD processors may benefit from doing this the naive way. Note that in the implementation we actually do inversion in R/q for the efficiency reasons mentioned in the description of the `NTRU` DPKE keypair generation. However, the code is equivalent so this should not matter.

The inversion in $S/2$ consumes 2043 cycles of which 332 cycles are spent converting to and from bitstrings. Bringing the inverse from $S/2$ to R/q through the multiplications in R/q costs about 30705 cycles. We will now discuss fast multiplication and multi-squarings in $S/2$.

Multiplication in $S/2$

Multiplication in $S/2$ makes use of the `vpclmulqdq` instruction which allows multiplication of 64-bit polynomials over GF(2^k) (in our case simply GF(2)). The existence of this instruction makes it worthwhile to convert the polynomials into bitstrings as a single 64-bit polynomial multiplication can be executed in a single cycle.

The 512-coefficient polynomials (that are 512 bits long) have two levels of Karatsuba applied such that there are 9 multiplications of 128-coefficient polynomials. Then we apply one instance of schoolbook multiplication so that we end up with 36 multiplications of 64-coefficient polynomials. These 36 multiplications are computed using `vpclmulqdq`. Afterwards a polynomial reduction is performed and the last three bits are masked out (since $n = 509$ and not 512). The result is that a single multiplication in $S/2$ takes just about 47 cycles. We need 12 multiplications in $S/2$ in order to implement the exponentiation which has a total of 564 cycles.

Multi-squarings in $S/2$

It turns out that squaring followed by polynomial reduction in $S/2$ is the same as doing a bit permutation on the bits of the polynomial [34]. More interestingly, repeated squaring is the same as repeated bit permutations which in turn is equivalent to a single combined bit permutation. This fact leads to very fast multi-squarings and was already realized in the implementation of `hrss701`. During that work, a tool was written to generate the bit permutations necessary for the multi-squarings. This tool was reused here to generate bit permutations necessary for the multi-squarings in `hps2048509`. A single multi-squaring costs about 91 cycles, smaller multi-squarings cost a little less while larger multi-squarings cost a little more. Since there are 13 multi-squarings in total they cost a total of 1183 cycles.

4.2.4 Inversion in $S/3$

Briefly mentioned before, Bernstein and Yang developed a fast constant-time gcd computation and modular inversion algorithm [14]. In their paper they performed a case-study in which they applied their algorithm to `hrss701` and it outperformed the optimized Almost Inverse implementation by a factor of 1.7. Based on this result their algorithm was also implemented in `hps2048509`.

The code for `hrss701` was adapted to work in `hps2048509`. In the original code each polynomial required six AVX2 registers since every coefficient was two bits and there were 701 coefficients ($\lceil \frac{701 \cdot 2}{256} \rceil = 6$). In `hps2048509` the four polynomials required in the algorithm only require four AVX2 registers each ($\lceil \frac{509 \cdot 2}{256} \rceil = 4$). Reducing the vector sizes is somewhat tricky since the initial state changes, this requires understanding the representation of polynomials. The representation of polynomials is different from `NTRU` in that the first 64 bits store the coefficients $x^0, x^4, \ldots, x^{252}$ and the second 64 bits store the coefficients $x^1, x^5, \ldots, x^{253}$ and so on. When multiplying by x this representation has the advantage that one only needs to move quadwords (64 bits) around. Another difference is that only $2 \cdot 508 - 1 = 1015$ iterations are necessary compared to 1399 in `hrss701`. As a result the `hps2048509` version performs better than the `hrss701` version. A single inversion in $S/3$ costs 23031 cycles.

4.2.5 Optimizing R/q to $S/3$ conversion

In the decryption of `NTRU` some $\mathbf{a}$ is computed modulo $(q, \Phi_1 \Phi_n)$ and is subsequently used in a reduction modulo $(3, \Phi_n)$. $\mathbf{a}$ must thus be converted from R/q to $S/3$. In the code of `NTRU` this is done using the `poly_Rq_to_S3` function. Since decryption is a fast operation in `NTRU` this function stood out in profiling (about 8% of the total). A straightforward conversion to AVX2 was implemented that processes 16 coefficients at a time rather than one. The result is that R/q to $S/3$ conversion takes about 179 cycles or just below 1% of the decryption time.

4.2.6 Optimizing cryptographic sorting

As mentioned before, `NTRU` requires constant-time sorting during sampling for the `hps` variants. The constant-time sorting algorithm used in the optimized implementation of `hps2048509` is the AVX2 version of djbsort. This sorting algorithm relies on sorting networks for its constant-time guarantees and was initially developed for use in NTRU Prime [13]. About 1814 cycles are consumed by the `crypto_sort` function.

Results and comparison

Performance testing was performed on an Intel i5-8250u (at 3.20GHz) using gcc 9.1.0. This Intel processor uses the Kaby Lake microarchitecture which is essentially the same as the Skylake microarchitecture (no changes in instruction performance). TurboBoost and HyperThreading were disabled. Both the reference and AVX2 implementation were compiled using the O3 optimization flag. Additionally, the AVX2 implementation used `-march=native`. The median amount of cycles for each operation was measured over 10000 runs. The `NTRU` DPKE operations were not included as `NTRU` technically only specifies a KEM. The DPKE exists as a central building block in the construction of this KEM. Besides the Kaby Lake microarchitecture we have also measured two other microarchitectures. The Haswell-based Intel i7-4770k at 3.50GHz using gcc 6.3.0 and the Zen-based AMD Ryzen 5 1600 at 3.20GHz using gcc 9.1.0. Ideally the Haswell machine would use the same frequency and compiler but this was not possible. Since most of the code is handwritten AVX2 (thus no compiler influence) and the clock speed is only about 10% higher we considered this valuable enough to include. Haswell is interesting to benchmark as it is the first microarchitecture from Intel that supports AVX2 and is thus a baseline performance.

The AVX2 implementation of `hps2048509` performs quite a bit better than the reference implementation for all processors, especially when generating keypairs and decapsulating. Encapsulation is quite a bit faster but the speedup is not as large due to the reference encapsulation already being quite a bit faster than the other two. Note that the encapsulation (and key generation) in `hps2048509` requires 2413 random bytes. This is a fairly large number of random bytes and retrieving these from the operating system using the `getrandom` syscall or reading from `/dev/urandom` directly costs about 60% of the total cycle count. For the measurements here an alternative approach is taken where only 32 random bytes are retrieved which are then put through the SHAKE128 XOF to produce 2413 bytes. This reduces the cycle count by about 14000 cycles on Kaby Lake which is quite substantial. For the Zen architecture it is around 70000 cycles as the `RDRAND` instruction which the Linux kernel uses for randomness is much slower than on Intel processors. If one did not need these random bytes or a much smaller amount then encapsulation would outperform decapsulation.

From Table 5.1 we can see that the Kaby Lake architecture performs best when using AVX2 instructions whereas Haswell and Zen are about a factor 2 behind. Haswell being slower than Kaby Lake can be largely explained by the fact that Skylake (and later) has two integer vector-multiplication units whereas Haswell has only one. This has the effect that it doubles the maximum throughput for multiplications. In the table we can indeed see that multiplication in R/q and $S/3$ (both heavily using vector multiplication) are almost twice as slow in Haswell. Since inversion in R/q is dominated by the eight multiplications in R/q we see the same slowdown in inversion in R/q. Inversion in $S/3$ is faster in Kaby Lake due to another vector execution unit having the ability to perform shifts and general improvement to the vector execution units such as faster conversion instructions. The Zen architecture is much newer than Haswell and generally outperforms Haswell (and even Skylake [26]) except when working with vector instructions. Zen vector units are 128-bit wide which means 256-bit vector instructions must occupy two execution units rather than one (or take twice as long in the vector multiplication case). Additionally, it has the same amount of vector execution units as Haswell and not Skylake. These facts cause execution units to be much less often available and thus incur a performance penalty. For the same reasons that Haswell is slower than Skylake, Zen is even slower than Haswell in polynomial multiplication and inversion. Despite that, Zen still performs similar to Haswell in encapsulation and decapsulation due to other operations such as the SHAKE128 computation outperforming Haswell. In the Zen 2 architecture some 256-bit instructions do occupy only one execution unit and as a result will likely have similar performance to Skylake (or slightly slower but still faster than Haswell). The exact details about Zen 2 have not yet been published except statements that the “datapath”, floating points unit, and Load/Store units have been doubled in width (thus being 256-bit).

In order to understand what can still be optimized in the AVX2 implementation we have looked at the cycle-count breakdown of the key generation, encapsulation, and decapsulation. The key generation of the `NTRU` KEM consists out of 50.23% of polynomial multiplication in R/q and all the functions that use it (multiplication in $S/3$, multiplication in S/q, inversion in R/q) since it is difficult to profile separately, 25.27% of inversion in $S/3$, 16.32% in gathering 32 bytes of randomness and computing SHAKE128, 3.19% of sampling which includes cryptographic sorting, 2.79% of inversion in R/q (including inversion in $R/2$, multiplication in $R/2$ and the multi-squarings), and 1.38% of packing the polynomials to bytes. The other 0.82% is spent in various small functions throughout the implementation such as `memcpy`. The only part of code that is not optimized using AVX2 and is worthwhile looking at in more detail is the computation of SHAKE128. It is likely that the performance can be improved somewhat or an alternative method to gather 2413 random bytes might be possible. Packing of polynomials could also be interesting but does not seem very friendly to vectorization and would as a result *maybe* be faster.

The cycle-count breakdown for encapsulation is as follows: 56.60% in gathering 32 bytes of randomness, computation of SHAKE128, and computation of the SHA3-256 hash. It is difficult to profile these separately since both SHAKE128 and SHA3-256 use the Keccak permutation but it seems the vast majority comes from the computation of SHAKE128. 13.79% comes from multiplication in R/q and any function that relies on it, 12.38% for packing and unpacking polynomials, 11.90% for sampling (including sorting) of polynomials r and m, 2.19% for reduction modulo 3 which is used in sampling and unpacking, 1.26% for the Lift operation, and finally 1.88% for miscellaneous operations (`memcpy`, function-call overheads, and so on). There are two targets here that are potentially interesting. The first target is computation of SHAKE128 which is used for randomness; this was already discussed during key generation. The other target is the packing and unpacking of polynomials. Currently encapsulation samples polynomials and packs them only for them to be immediately unpacked by the encryption. This can be avoided by merging the two. Since this is irrelevant to AVX2 it was left out but has become much more noticeable due to decreased cycle counts for other operations.

Cycle-count breakdown for decapsulation is as follows: 51.87% is spent on polynomial multiplication in R/q and the functions that rely on it, 25.94% is spent on the SHA3-256 operation, 13.39% on the packing and unpacking of polynomials, 3.62% in the decapsulation itself for checking that r and m are in the message space, 1.39% for reduction modulo 3 in the unpacking of polynomials, 0.99% for conversion from R/q to $S/3$, 0.80% on the Lift operation, and the other 2.00% is spent on miscellaneous operations.

With the AVX2 optimizations `NTRU` requires about 29 microseconds on the Intel i5-8250u processor at 3.20GHz to generate a keypair. Encapsulation takes about 7.4 microseconds and decapsulation takes about 6.8 microseconds. These processing times are sufficiently small enough for `NTRU` to be considered a practical quantum-resistant KEM.

Table 5.1: Cycle counts of operations in `NTRU` for both the reference implementation and the AVX2 implementation as well as the speedup the AVX2 implementation provides.

		Reference cycles	AVX2 cycles	Speedup
Kaby Lake	Multiplication in R/q	304,786	3,550	85.86x
	Multiplication in $S/3$	293,455	3,677	79.81x
	Inversion in R/q	3,343,846	30,533	109.52x
	Inversion in $S/3$	1,568,070	23,039	68.06x
	R/q to $S/3$	2,225	178	12.50x
	`sample_fixed_type` (`crypto_sort`)	33,209	2,870	11.57x
	KEM keypair generation	6,164,431	91,358	67.48x
	KEM encapsulation	357,890	23,773	15.05x
	KEM decapsulation	859,044	21,870	39.28x
Haswell	Multiplication in R/q	623,296	6,876	90.65x
	Multiplication in $S/3$	625,000	7,136	87.58x
	Inversion in R/q	6,041,188	60,712	99.51x
	Inversion in $S/3$	3,049,048	49,956	61.03x
	R/q to $S/3$	4,568	352	12.98x
	`sample_fixed_type` (`crypto_sort`)	66,268	6,736	9.84x
	KEM keypair generation	12,321,490	186,308	66.14x
	KEM encapsulation	746,760	46,956	15.90x
	KEM decapsulation	1,895,960	42,204	44.92x
Zen	Multiplication in R/q	482,368	9,024	53.45x
	Multiplication in $S/3$	489,312	9,376	52.18x
	Inversion in R/q	5,355,456	112,128	47.76x
	Inversion in $S/3$	2,397,312	55,072	43.53x
	R/q to $S/3$	3,968	544	7.29x
	`sample_fixed_type` (`crypto_sort`)	56,224	7,840	7.17x
	KEM keypair generation	10,689,696	246,720	43.33x
	KEM encapsulation	659,456	44,960	14.67x
	KEM decapsulation	1,490,688	45,760	32.58x

5.1 Comparison to other submissions

In this section we will compare the NTRU KEM to several KEMs in the NIST competition. Ideally comparisons should be fair in that they all provide the same amount of security. Defining how many bits of security a KEM has is difficult in the post-quantum setting and as such NIST has defined five security categories which say something about how much computational resources are necessary to break a KEM. A KEM that lies in the first category should at least require computational resources comparable to those required for a brute-force key search on a block cipher with a 128-bit key. Security level five is the same except for a block cipher with a 256-bit key. The hps2048509 parameter set aims for a security level of one. As such, we will compare against other submissions with parameters aiming for the same security level. There may still be some variation on security but comparing only against exactly the same security is not doable. All of the comparisons will be using lattice-based cryptography such that we can compare within this subcategory of submissions.

Comparisons are made on keysizes, ciphertext sizes, cycle counts, and whether an implementation is constant-time. Knowing whether an implementation is constant-time is important as it may cause a heavy performance in order to reach. Gathering performance results is tricky due to different architectures and testing setups. For comparison we will take our NTRU Haswell results as it is the most common architecture tested.

In Table 5.2 we see that the optimized implementation of NTRU is among the fastest and the sizes of the keys are among the smallest. Combined with the fact that NTRU has had over 20 years of cryptanalysis it is a promising submission.

Table 5.2: Comparison of six lattice-based KEMs. Cycle counts were measured using an Intel i7-4770k unless states otherwise. Cycles contains key generation (**K**), encapsulation (**E**), and decapsulation (**D**). Bytes contains secret key size (**sk**), public key size (**pk**), and ciphertext size (**c**). **Scheme** is followed by the parameter set in parenthesis. **ct?** indicates whether the implementation is constant-time.

Scheme	**Type**	**ct?**	**Cycles**		**Bytes**	
`NTRU` [48] (`hps2048509`)	IND-CCA2 KEM	yes	**K:**	186,308	**sk:**	935
			E:	46,956	**pk:**	699
			D:	42,204	**c:**	699
`NewHope` [4] (`NH-512-CCA-KEM`)	IND-CCA2 KEM	yes	**K:**	68,080	**sk:**	1888
			E:	109,836	**pk:**	928
			D:	114,176	**c:**	1120
`CRYSTALS-KYBER` [17] (`KYBER512`)	IND-CCA2 KEM	yes	**K:**	33,428	**sk:**	1632[a]
			E:	49,184	**pk:**	800
			D:	40,564[a]	**c:**	736
`FrodoKEM` [3] (`FrodoKEM-640-AES`)	IND-CCA2 KEM	yes	**K:**	≈1,384,000[b]	**sk:**	19888
			E:	≈1,858,000[b]	**pk:**	9616
			D:	≈1,749,000[b]	**c:**	9720
`NTRU Prime` [13] (`sntrup4591761`)	IND-CCA2 KEM	yes	**K:**	940,852[c]	**sk:**	1600
			E:	44,788[c]	**pk:**	1218
			D:	93,856[c]	**c:**	1047
`Round5` [7] (`R5ND_1KEM_0d`)	IND-CPA KEM	yes	**K:**	≈57,600[d]	**sk:**	16
			E:	≈94,900[d]	**pk:**	634
			D:	≈45,000[d]	**c:**	682

[a] Secret key size can be reduced to just 32 bytes but at the cost of about 53% increased decapsulation time.

[b] Intel i7-6700 (Skylake) at 3.4GHz. Compare against Kaby Lake results above.

[c] Intel Xeon E3-1275 v3 (Haswell) at 3.5GHz. `ntrulpr4591761` is an alternative if key generastion is a problem but comes at the cost of increased cycle counts for encapsulation and decapsulation.

[d] MacBook Pro 15.1" with Intel i7 2.6GHz (unknown what microarchitecture).

Conclusions

In this thesis we have seen what post-quantum cryptography is and why it is necessary. Concretely, we have looked in detail at the lattice-based cryptography scheme `NTRU` which uses polynomial rings for its operation. These operations allow for construction of a secure deterministic public-key encryption scheme and a corresponding key encapsulation mechanism.

In order to get a better view of the practical performance of `NTRU` for the NIST competition we have implemented `NTRU` using AVX2. This implementation was performance focused and showed significant speedup compared to the reference implementation without, to our knowledge, sacrificing the security of the implementation. The time consumption of `NTRU` in the optimized AVX2 implementation is low enough to be considered practical and in turns shows that it is possible to use quantum-resistant cryptography using modern hardware.

6.1 Future work

In this work we have only considered the `hps2048509` parameter set and for future work it would make sense to consider the other parameter sets. Specifically `hps4096821` would be interesting to look at since the polynomials are quite a bit larger. `hps2048677` would not be as interesting since it would have quite a bit of overlap in the implementation with `hrss701` due to both having the polynomial with 704 coefficients be the closest multiple of 32 coefficients. Note that the code for polynomial inversion in $S/3$ has been generalized by Bernstein (after being manually adapted for `hps2048509`) in order to support different size polynomials.

Outside of the other parameter sets there are likely some small optimizations that can be made to `hps2048509`. One of these optimizations is the alternative implementation of multiplication in $S/3$ discussed in 4.2.2. A quick calculation shows that it will likely improve decryption cycles by about five to ten percent. Another optimization would be to merge sampling and encryption in encapsulation since encryption currently immediately unpacks the polynomials that the sampling packs. In Chapter 5 we have seen how much this can save. One last smaller optimization that is worth looking into is a more efficient method to expand from 32 bytes of randomness to 2413 bytes of randomness. Currently this is done using SHAKE128 which is already much better than retrieving 2413 random bytes from the operating system but is still a large majority of the cycle count in encapsulation. Outside of small optimizations it might also be worthwhile to look at performance specifically for AMD processors as most of the optimizations in this thesis only consider AVX2 on Intel processors.

Other possibilities for future work are focusing on memory usage, binary size, and power consumption. For memory usage it should be possible to choose coefficients with smaller sizes that are not a multiple of a byte. The binary size of the AVX2 implementation is quite a bit larger than the reference implementation. The AVX2 implementation unrolls every single loop for performance but causes the same instruction sequences to repeat. Another source of duplication is the code for polynomial reductions and modular reduction. These are needed a few times and it is beneficial for performance to have this code duplicated.

Bibliography

[1] Miklós Ajtai. "Generating hard instances of lattice problems". In: *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*. ACM. 1996, pp. 99–108.

[2] Martin Albrecht, Carlos Cid, Kenneth G. Paterson, CJ Tjhai, and Martin Tomlinson. "NTS-KEM". In: *NIST submissions* (2019). URL: `https://nts-kem.io/`.

[3] Erdem Alkim, Joppe W. Bos, Léo Ducas, Patrick Longa, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Chris Peikert, Ananth Raghunathan, Douglas Stebila, Karen Easterbrook, and Brian LaMacchia. "FrodoKEM Learning With Errors Key Encapsulation". In: *NIST submissions* (2019). URL: `https://frodokem.org/files/FrodoKEM-specification-20190702.pdf`.

[4] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. *Post-quantum key exchange - a new hope.* Cryptology ePrint Archive, Report 2015/1092. 2015. URL: `https://eprint.iacr.org/2015/1092`.

[5] Matthew Amy, Olivia Di Matteo, Vlad Gheorghiu, Michele Mosca, Alex Parent, and John Schanck. "Estimating the cost of generic quantum pre-image attacks on SHA-2 and SHA-3". In: *International Conference on Selected Areas in Cryptography*. Springer. 2016, pp. 317–337.

[6] Jean-Philippe Aumasson, Daniel J. Bernstein, Christoph Dobraunig, Maria Eichlseder, Scott Fluhrer, Stefan-Lukas Gazdag, Andreas Hülsing, Panos Kampanakis, Stefan Kölbl, Tanja Lange, Martin M. Lauridsen, Florian Mendel, Ruben Niederhagen, Christian Rechberger, Joost Rijneveld, and Peter Schwabe. "SPHINCS$^+$". In: *NIST submissions* (2019). URL: `https://sphincs.org/data/sphincs+-round2-specification.pdf`.

[7] Hayo Baan, Sauvik Bhattacharya, Scott Fluhrer, Oscar Garcia-Morchon, Thijs Laarhoven, Rachel Player, Ronald Rietman, Markku-Juhani Olavi Saarinen, Ludo Tolhuizen, Jose Luis Torre Arce, and Zhenfei Zhang. "Round5 KEM and PKE based on (Ring) Learning with Rounding". In: *NIST submissions* (2019). URL: `https://round5.org/Supporting_Documentation/Round5_Submission.pdf`.

[8] Mihir Bellare, Anand Desai, David Pointcheval, and Phillip Rogaway. "Relations among notions of security for public-key encryption schemes". In: *Annual International Cryptology Conference*. Springer. 1998, pp. 26–45.

[9] Mihir Bellare and Phillip Rogaway. "Optimal asymmetric encryption". In: *Workshop on the Theory and Application of of Cryptographic Techniques*. Springer. 1994, pp. 92–111.

[10] Daniel J Bernstein. "Post-quantum cryptography". In: *Encyclopedia of Cryptography and Security* (2011), pp. 949–950.

[11] Daniel J. Bernstein. *djbsort: Intro*. 2017. URL: `https://sorting.cr.yp.to/` (visited on 07/24/2019).

[12] Daniel J Bernstein, Tung Chou, Tanja Lange, Ingo von Maurich, Rafael Misoczki, Ruben Niederhagen, Edoardo Persichetti, Christiane Peters, Peter Schwabe, and Nicolas Sendrier. "Classic McEliece: conservative code-based cryptography". In: *NIST submissions* (2017). URL: `https://classic.mceliece.org/nist/mceliece-20171129.pdf`.

[13] Daniel J Bernstein, Chitchanok Chuengsatiansup, Tanja Lange, and Christine van Vredendaal. "NTRU Prime: reducing attack surface at low cost". In: *International Conference on Selected Areas in Cryptography*. Springer. 2017, pp. 235–260. URL: `https://ntruprime.cr.yp.to/ntruprime-20170816.pdf`.

[14] Daniel J Bernstein and Bo-Yin Yang. "Fast constant-time gcd computation and modular inversion". In: CHES2019. 2019. URL: `https://gcd.cr.yp.to/safegcd-20190413.pdf`. Forthcoming.

[15] Dan Boneh and Matt Franklin. "Identity-based encryption from the Weil pairing". In: *Annual international cryptology conference*. Springer. 2001, pp. 213–229. URL: `https://crypto.stanford.edu/~dabo/papers/bfibe.pdf`.

[16] Tomas J Boothby and Robert W Bradshaw. "Bitslicing and the Method of Four Russians over larger finite fields". In: *arXiv preprint arXiv: 0901.1413* (2009). URL: `https://arxiv.org/abs/0901.1413`.

[17] Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. "CRYSTALS-Kyber: a CCA-secure module-lattice-based KEM". In: *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2018, pp. 353–367. URL: `https://eprint.iacr.org/2017/634`.

[18] Johannes Buchmann, Erik Dahmen, and Andreas Hülsing. "XMSS - a practical forward secure signature scheme based on minimal security assumptions". In: *International Workshop on Post-Quantum Cryptography*. Springer. 2011, pp. 117–129. URL: `https://eprint.iacr.org/2011/484.pdf`.

[19] Debrup Chakraborty, Vicente Hernandez-Jimenez, and Palash Sarkar. "Another look at XCB". In: *Cryptography and Communications* 7.4 (2015), pp. 439–468. URL: `https://eprint.iacr.org/2013/823`.

[20] Ming-Shing Chen, Andreas Hülsing, Joost Rijneveld, Simona Samardjiska, and Peter Schwabe. "MQDSS specifications". In: *NIST submissions* (2019). URL: `http://mqdss.org/files/MQDSS_Ver1point1.pdf`.

[21] Ronald Cramer and Victor Shoup. "Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack". In: *SIAM Journal on Computing* 33.1 (2003), pp. 167–226. URL: `https://eprint.iacr.org/2001/108`.

[22] Whitfield Diffie and Martin Hellman. "New directions in cryptography". In: *IEEE transactions on Information Theory* 22.6 (1976), pp. 644–654. URL: `https://ee.stanford.edu/~hellman/publications/24.pdf`.

[23] Danny Dolev, Cynthia Dwork, and Moni Naor. "Nonmalleable cryptography". In: *SIAM review* 45.4 (2003), pp. 727–784.

[24] Achim Flammenkamp. *Shortest Addition Chains*. URL: `http://wwwhomes.uni-bielefeld.de/achim/addition_chain.html` (visited on 07/24/2019).

[25] Agner Fog. *Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs.* 1996-2018. URL: `https://www.agner.org/optimize/instruction_tables.pdf` (visited on 07/24/2019).

[26] Agner Fog. *The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers.* 1996-2018. URL: `https://www.agner.org/optimize/microarchitecture.pdf` (visited on 07/24/2019).

[27] David Galindo. “Boneh-Franklin identity based encryption revisited”. In: *International Colloquium on Automata, Languages, and Programming*. Springer. 2005, pp. 791–802. URL: https://eprint.iacr.org/2005/117.

[28] Oded Goldreich, Shafi Goldwasser, and Shai Halevi. “Collision-Free Hashing from Lattice Problems”. In: *Studies in Complexity and Cryptography. Miscellanea on the Interplay between Randomness and Computation*. Springer Berlin Heidelberg, 2011, pp. 30–39. ISBN: 978-3-642-22670-0. DOI: 10.1007/978-3-642-22670-0_5. URL: http://www.wisdom.weizmann.ac.il/~oded/COL/cfh.pdf.

[29] Shafi Goldwasser and Silvio Micali. “Probabilistic encryption”. In: *Journal of computer and system sciences* 28.2 (1984), pp. 270–299.

[30] Lov K Grover. “A fast quantum mechanical algorithm for database search”. In: *arXiv preprint quant-ph/9605043* (1996). URL: https://arxiv.org/abs/quant-ph/9605043.

[31] Jeffrey Hoffstein, Cong Chen, William Whyte, and Zhenfei Zhang. “NTRUEncrypt: A lattice based encryption algorithm”. In: *NIST submissions* (2017). URL: https://www.onboardsecurity.com/nist-post-quantum-crypto-submission.

[32] Jeffrey Hoffstein, Jill Pipher, and Joseph H Silverman. “NTRU: A ring-based public key cryptosystem”. In: *International Algorithmic Number Theory Symposium*. Springer. 1998, pp. 267–288.

[33] Andreas Hülsing, Joost Rijneveld, John M Schanck, and Peter Schwabe. “NTRU-HRSS-KEM”. In: *NIST submissions* (2017). URL: https://ntru-hrss.org/data/ntrukem.pdf.

[34] Andreas Hülsing, Joost Rijneveld, John Schanck, and Peter Schwabe. “High-speed key encapsulation from NTRU”. In: *International Conference on Cryptographic Hardware and Embedded Systems*. Springer. 2017, pp. 232–252. URL: https://eprint.iacr.org/2017/667.

[35] Toshiya Itoh and Shigeo Tsujii. “A fast algorithm for computing multiplicative inverses in GF(2^m) using normal bases”. In: *Information and computation* 78.3 (1988), pp. 171–177. URL: https://core.ac.uk/download/pdf/82657793.pdf.

[36] Tetsu Iwata, Keisuke Ohashi, and Kazuhiko Minematsu. “Breaking and repairing GCM security proofs”. In: *Annual Cryptology Conference*. Springer. 2012, pp. 31–49. URL: https://eprint.iacr.org/2012/438.

[37] A. Karatsuba and Yuri Petrovich Ofman. “Multiplication of Many-Digital Numbers by Automatic Computers”. In: *Proceedings of the USSR Academy of Sciences* (1963).

[38] Auguste Kerckhoffs. “La cryptographie militaire”. In: *Journal des sciences militaires* 9 (Jan. 1883), pp. 5–38. URL: `https://petitcolas.net/kerckhoffs/crypto_militaire_1_b.pdf`.

[39] Donald E Knuth. *Art of computer programming, volume 2: Seminumerical algorithms.* Addison-Wesley Professional, 2014.

[40] Arjen Klaas Lenstra, Hendrik Willem Lenstra, and László Lovász. “Factoring polynomials with rational coefficients”. In: *Mathematische Annalen* 261.4 (1982), pp. 515–534. URL: `https://www.math.leidenuniv.nl/~hwl/PUBLICATIONS/1982f/art.pdf`.

[41] Gregory Lento. *Optimizing performance with Intel Advanced Vector Extensions.* 2014. URL: `https://computing.llnl.gov/tutorials/linux_clusters/intelAVXperformanceWhitePaper.pdf` (visited on 07/24/2019).

[42] Robert J. McEliece. “A public key cryptosystem based on algebraic coding theory”. In: *Technical report, NASA* (1978). URL: `https://ipnpr.jpl.nasa.gov/progress_report2/42-44/44N.PDF`.

[43] David A McGrew and Scott R Fluhrer. “The security of the extended codebook (XCB) mode of operation”. In: *International Workshop on Selected Areas in Cryptography.* Springer. 2007, pp. 311–327.

[44] David A McGrew and John Viega. “The security and performance of the Galois/Counter Mode (GCM) of operation”. In: *International Conference on Cryptology in India.* Springer. 2004, pp. 343–355. URL: `https://eprint.iacr.org/2004/193`.

[45] PQCRYPTO. *Post-quantum cryptography for long-term security.* 2019. URL: `https://pqcrypto.eu.org/` (visited on 07/24/2019).

[46] Ronald L Rivest, Adi Shamir, and Leonard Adleman. “A method for obtaining digital signatures and public-key cryptosystems”. In: *Communications of the ACM* 21.2 (1978), pp. 120–126. URL: `https://people.csail.mit.edu/rivest/Rsapaper.pdf`.

[47] Tsunekazu Saito, Keita Xagawa, and Takashi Yamakawa. “Tightly-secure key-encapsulation mechanism in the quantum random oracle model”. In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques.* Springer. 2018, pp. 520–551. URL: `https://eprint.iacr.org/2017/1005`.

[48] John M. Schank, Cong Chen, Oussama Danba, Jeffrey Hoffstein, Andreas Hülsing, Joost Rijneveld, Peter Schwabe, William Whyte, and Zhenfei Zhang. “NTRU”. In: *NIST submissions* (2019). URL: `https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/round-2/submissions/NTRU-Round2.zip`.

[49] Richard Schroeppel, Hilarie Orman, Sean O'Malley, and Oliver Spatscheck. "Fast key exchange with elliptic curve systems". In: *Annual International Cryptology Conference*. Springer. 1995, pp. 43–56.

[50] Peter W Shor. "Algorithms for quantum computation: Discrete logarithms and factoring". In: *Proceedings 35th annual symposium on foundations of computer science*. IEEE. 1994, pp. 124–134. URL: `https://pdfs.semanticscholar.org/6902/cb196ec032852ff31cc178ca822a5f67b2f2.pdf`.

[51] Victor Shoup. "OAEP reconsidered". In: *Annual International Cryptology Conference*. Springer. 2001, pp. 239–259. URL: `https://eprint.iacr.org/2000/060`.

[52] Joseph H. Silverman. "Almost inverses and fast NTRU key creation". In: *Technical Report #014* (1999). URL: `https://assets.onboardsecurity.com/static/downloads/NTRU/resources/NTRUTech014.pdf`.

[53] National Institute for Standards and Technology. *Post-Quantum Cryptography — CSRC*. 2019. URL: `https://csrc.nist.gov/Projects/Post-Quantum-Cryptography` (visited on 07/24/2019).

[54] NTRU Team. *NTRU parameter sets for the second round of the NIST process*. 2019. URL: `https://github.com/jschanck/ntru` (visited on 07/24/2019).

[55] André Weimerskirch and Christof Paar. "Generalizations of the Karatsuba Algorithm for Efficient Implementations." In: *IACR Cryptology ePrint Archive* 2006 (2006), p. 224. URL: `https://eprint.iacr.org/2006/224`.

[56] Alberto Zanoni. "Toom-cook 8-way for long integers multiplication". In: *2009 11th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. IEEE. 2009, pp. 54–57.