Radboud University Nijmegen

Master Thesis

NewHope for ARM

An Efficient Implementation of the Post-Quantum Ephemeral Key Exchange NewHope for the ARMv6-M Architecture

Author:
Philipp Jakubeit

Supervisor:
Dr. Peter Schwabe

A thesis submitted in fulfillment of the requirements for the degree of MSc Computing Science

in the

Digital Security
Institute for Computing and Information Sciences

August 6, 2016

Declaration of Authorship

I, Philipp Jakubeit, declare that this thesis titled "NewHope for ARM - An Efficient Implementation of the Post-Quantum Ephemeral Key Exchange NewHope for the ARMv6-M Architecture" and the work presented in it are my own. I confirm that:

- This work was done wholly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

"*If quantum mechanics hasn't profoundly shocked you, you haven't understood it yet.*"

Niels Bohr

RADBOUD UNIVERSITY NIJMEGEN

Abstract

Faculty of Science

Institute for Computing and Information Sciences

MSc Computing Science

NewHope for ARM
An Efficient Implementation of the Post-Quantum Ephemeral Key Exchange NewHope for the ARMv6-M Architecture

by Philipp Jakubeit

In this thesis we provide the mathematical background information and details of our ARM Cortex-M0 implementation of the post-quantum key exchange NewHope. NewHope was designed and published in 2015 by Alkim, Ducas, Pöppelmann and Schwabe [1]. It bases its security claims on lattice problems, the ring-learning-with-errors problem which is reducible to the shortest vector problem. The security claims provided by these problems are not related to the factorization problem or the discrete-logarithm problem. Therefore, Shor's algorithm does not affect them and no other quantum algorithm is known which could solve them in polynomial time. Actors like the NIST [58], the NSA [59] or the Tor project [50] have recognized the relevance of post-quantum cryptography. Key exchanges prove relevant as they form the basis of forward secrecy in a world with large-scale quantum computers at the horizon. These key exchanges must be designed and proven to be efficiently implementable. We performed an implementation of NewHope on the embedded ARMv6M architecture based Cortex-M0 and increased the cycle count for the full key exchange by 37%. The main computation of the key exchange lies in computing the multiplication of polynomials. We optimized this multiplication realized by the number theoretic transform by 54% compared to the reference implementation and by 6% to 45% compared to research in the literature.

Acknowledgements

I want to thank my supervisor Peter Schwabe, for the scientific insights and assistance he provided during the whole period of writing this thesis. I want to thank Lejla Batina for taking the time to be the second reader of my thesis. I want to thank Erdem Alkim for insights into the target architecture and for working out Keccak together. I want to thank Pol van Aubel for getting me started with microcontrollers on Arch-Linux and Joost Rijneveld for assistance in figuring out communication issues with the Cortex-M0. I want to further thank my family and friends for reading parts of my thesis, and providing valuable feedback as well as supportive and encouraging words. Thanks to all of you.

Contents

List of Figures

List of Tables

List of Abbreviations

BCNS	**B**os, **C**ostello, **N**aehrig and **S**tebila
CVP	**C**losest **V**ector **P**roblem
DFT	**D**iscrete **F**ourier **T**ransformation
DLP	**D**iscrete **L**ogarithm **P**roblem
DMA	**D**irect **M**emory **A**ccess
ECC	**E**liptic **C**urve **C**ryptography
FFT	**F**ast **F**ourier **T**ransformation
GPIO	**G**eneral **P**urpose **I**nput / **O**utput
KB	**K**ilo **B**yte
KEM	**K**ey-**E**ncapsulation **M**echanism
LWE	**L**earning **W**ith **E**rror
NP	**N**on-deterministic **P**olynomial time
NTT	**N**umber **T**heoretic **T**ransformation
P	**P**olynomial time
PA	**P**in **A**rray
RLWE	**R**ing **L**earning **W**ith **E**rror
RNG	**R**andom **N**umber **G**enerator
ROM	**R**ead-**O**nly **M**emory
ROM	**R**andom-**A**ccess **M**emory
RSA	**R**ivest, **S**hamir and **A**dleman
RXD	**R**ead **D**ata
SVP	**S**hortest **V**ector **P**roblem
TLS	**T**ransport - **L**ayer **S**ecurity
TTL	**T**ransistor - **T**ransistor **L**ogic
TXD	**T**ransmit **D**ata
UART	**U**niversal **A**synchronous **R**eceiver/**T**ransmitter
USB	**U**niversal **S**erial **B**us

Introduction

In 1994, Peter Shor published an algorithm which is known by the name of 'Shor's algorithm' [77]. It is intended to compute the prime factorization of large integers and is a so called quantum-algorithm. A quantum-algorithm is an algorithm which performs its computations on a quantum computer, a computer based on quantum-mechanical phenomena. The breakthrough Shor's algorithm provides is that it is able to compute the prime factors of an arbitrary long integer number in a reasonable amount of time, polynomial time to be explicit. This was not possible before and leads to it constituting a threat to the contemporary security backbone of the Internet.

Due to contemporary cryptography used on the Internet not being able to withstand the implications of Shor's algorithm, research is conducted on alternative approaches. One of such promising approaches are cryptographic schemes which are based on lattices. One computationally hard problem based on lattices to build cryptography on is the so called 'ring-learning-with-errors (RLWE) problem'. A theoretical concept of a key exchange was first published by Ding, Xie en Lin [18] in 2012. The specific key exchange we are investigating was published by Peikert [65] in 2014. After an instantiation performed by Bos, Costello, Naehrig and Stebila, researchers at Microsoft [7], a second, faster and more secure instantiation was published. It was named, due to the possibility of providing a secure alternative for the Internet, a NewHope [1].

Future-proof secure communication is not only relevant for personal computers and servers, but also for embedded hardware. Therefore, it is essential for cryptographic primitives to be efficiently computable on those restricted environments. We chose the low end Cortex-M0 with its ARMv6M architecture as target device to provide an efficient implementation of NewHope.

The remainder of this thesis is split into two major parts, the background context needed to understand the content of this thesis and the description of the implementation we performed. The first part, the preliminaries, are split into two subparts. The first describes the mathematical basics on which the security of the cryptographic scheme NewHope is build. The second dives into the scheme and elaborates it and relevant design decisions in detail. The second part, our implementation, presents the main contribution of this thesis. It first introduces the target architecture and continues with elaborating our implementation in detail. We start from the changes we needed to perform to get it running on the restricted hardware. We continue with the description of the specific optimization measures we performed and end in a comparison of our optimized version with the non-optimized version and comparable results from the literature.

Preliminaries

This section explains the relevant mathematical basics to understand the following sections and the essence of the research we performed. A reader who already possesses background information about the necessity of post-quantum cryptography, lattices and the hard problems forming the basis of RLWE might want to skip this section. First, basic information about complexity theory is given. The goal is to enable every reader to understand further assumptions and claims made on the basis of complexity analysis. Second, the concept of post-quantum cryptography is elaborated on, from its origins to its implications. Third, the mathematical concept of lattices in general gets introduced. Then the concepts of the shortest vector problem, the closest vector problem and the closely related learning-with-errors (LWE) problem are provided. The concept of LWE is then extended to RLWE, which is performed on lattices with a certain algebraic structure. Fourth, instantiations of RLWE are discussed and the key exchange (NewHope) used for our fast ARM implementation will be introduced. Additional information on relevant mathematical concepts regarding NewHope will also be presented in this subsection.

2.1 Complexity

In computer science complexity can be considered from two perspectives; The first is describing the running time of an algorithm, the second is describing the memory usage of an algorithm. Within this section we will focus on the first perspective, the time complexity. The symbol to represent complexity is $\mathcal{O}$. It is called 'big-O' notation and is part of the family of Landau notations, which allows a more specific complexity analysis. $f(n) = \mathcal{O}(g(n))$ is defined for the functions f and g applied on a sufficiently large n as follows [67]:

$$\exists c > 0 \, \exists k \, \forall n \geq k : \, 0 \leq f(n) \leq c \cdot g(n).$$

In words; g defines an upper bound function, which is multiplied by a constant c greater zero, for the function f evaluated for the same input n, which needs to be sufficiently large ($n > k$). This implies that from a given threshold value k on, $c \cdot g(n)$ is always greater or equal to $f(n)$. A depiction of this can be seen in Figure 2.1. In the remainder of this thesis we will describe in words that some algorithm $\mathcal{A}$ has a complexity of $\mathcal{O}$(upper bound of $\mathcal{A}$) to refer to this definition of time complexity.

One variant needs to be introduced, which is $f(n) = \tilde{\mathcal{O}}(g(n))$. It is used to omit logarithmic factors and is defined as

$$\exists x : f(n) \in \mathcal{O}(g(n) \log^x g(n)).$$

We will briefly introduce three time complexity categories by name: Sub-quadratic time is used to categorize algorithms running in greater than linear time ($\mathcal{O}(n)$), but less than quadratic time ($\mathcal{O}(n^2)$). Polynomial time is used to categorize algorithms running in $2^{\mathcal{O}(\log n)}$ and is computable in a reasonable amount of time. Sub-exponential time is used to categorize algorithms running in $\mathcal{O}(2^{n^\epsilon})$ for $0 < \epsilon < 1$ and is impractical to compute for large enough parameters n.

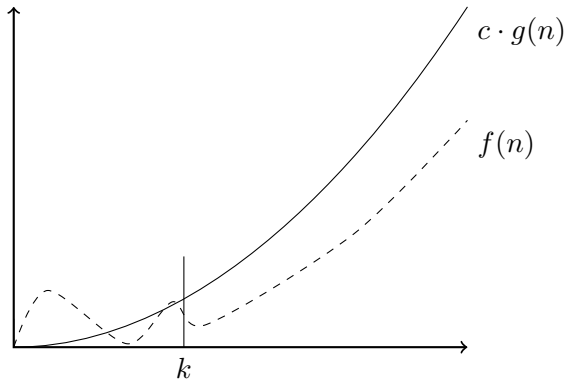

FIGURE 2.1: An example graph for the relation of the two functions f and g for which holds $f(n) = \mathcal{O}(g(n))$. Inspired by [67].

In complexity theory one distinguishes two kinds of problems: The 'search problem' and the 'decision problem'. The former is to find a solution to a certain instance of a problem. The latter is to provide a yes-or-no answer to an instance of the problem. Regarding decision problems two complexity classes can be distinguished, **P** (polynomial time) and **NP** (non-deterministic polynomial time). A problem $\mathcal{P}$ belongs to class **P** if it can be decided on a deterministic Turing machine [80] in polynomial time. The problem $\mathcal{P}$ belongs to class **NP** if the 'yes' instances can be verified by a deterministic Turing machine in polynomial time. This implies that $\mathbf{P} \in \mathbf{NP}$. Whether these two complexity classes are equal or unequal is still an unsolved problem in computer science. Figure 2.2 depicts the relation of the set of problems in either case.

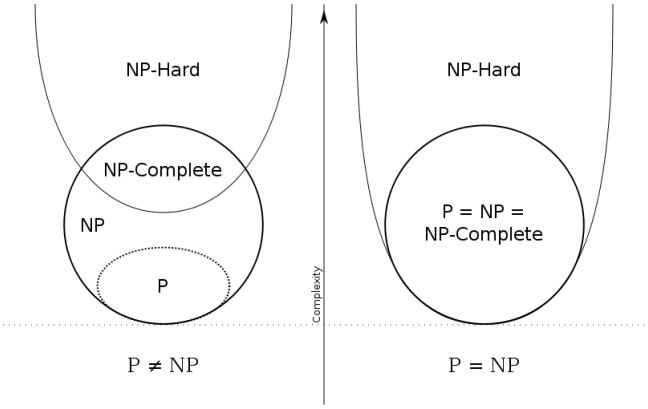

FIGURE 2.2: Visualization of the complexity problem classes **P** and **NP** based on either the equality or inequality. Taken from [34].

As can be seen in Figure 2.2 the concept of **NP**-hardness and **NP**-completeness need to be clarified. The concept of an **NP**-hard problem is that any problem in **NP** can be reduced to it in polynomial time. The concept of an **NP**-complete problem is that it is both, **NP**-hard (thus every problem can be reduced to it in polynomial time) and it is in **NP** (thus an answer can be verified in polynomial time).

2.2 Post-Quantum Cryptography

The term 'post-quantum cryptography' is a combination of words rooted in Latin (post-quantum) as well as old Greek (cryptography). The prefix 'post' can be translated to 'after', while the term 'quantum' describes the smallest quantity of radiant energy in physics. In the context of this thesis, it is used as a reference to quantum computers. These are computers which do not base their computations on binary states (e.g. realized by electrical current) as classic computers do, but are based on quantum-mechanical phenomena. In the remainder of this thesis, we will use the terms quantum and classical as antonyms, e.g. classical computers and quantum computers. The Latin part (post-quantum) provides the context for the Greek part (cryptography), a context describing a situation 'after quantum computers exist'. The Greek part is a composition of the word 'crypto' and 'graphy'. The first, 'crypto', can be translated to 'hidden' or 'concealed', while the later, 'graphy' can be translated to 'to write' or 'to represent'. Together they consequently describe the field of science which studies possibilities to hide the written or to conceal representations of information.

2.2.1 Classical Cryptography

Classical cryptography has been used for at least 4000 years [52]. Written notes date back to Mesopotamia where the 'At-bash' cipher was used and, of course, to Greek and the Roman empire, from which the 'Polybius

Square' and the 'Caesar cipher' are still known today [12]. However, the application and complexity of methods used to conceal information varied through the centuries. Until the early 20th century especially the military and diplomats made use of this science. In the second half of the 20th century business applications, mainly the financial sector, extended the circle of users. At the end of the 20th century with the Internet at the horizon, classical cryptography became part of everyday life.

Classical cryptography can be divided into two subfields, symmetric cryptography and asymmetric cryptography. The historical examples are all situated in the field of symmetric cryptography. Only advances in the field of logic made it possible for asymmetric cryptography to be thought of, as was done by William Stanley Jevons in his book 'The Principles of Science' in 1874 [38]. However, it took nearly one more century until the first asymmetric cryptographic scheme was instantiated, publicly accessible in 1976 by Diffie and Hellman [17]. Earlier the concept was proposed to the British intelligence agency in 1970 by Ellis [23], which was made public in 1997.

2.2.2 Symmetric Cryptography

Symmetric cryptography is the subfield of cryptography most people have in mind when confronted with the term cryptography. Represented in a model it can be described as two parties, Alice and Bob, who want to exchange information in a protected way. An illustration can be seen in Figure 2.3.

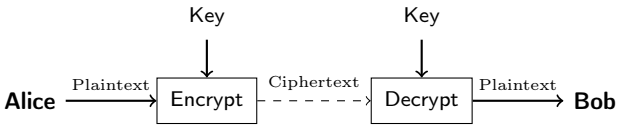

FIGURE 2.3: A brief model of symmetric cryptography.

Following from the terminology in Figure 2.3, the ciphertext is the encrypted information and is typically send via an insecure channel. Nowadays this channel could be the Internet, historically this could have been a messenger traveling through enemy territory. Both parties, Alice and Bob use the same key for encryption and for decryption. This, however, only substitutes the problem of exchanging information by exchanging the key.

2.2.3 Asymmetric Cryptography

Asymmetric cryptography is the subfield of cryptography, which can solve the problem of exchanging the key. Three major scheme categories of asymmetric cryptography can be distinguished. The category of signature schemes, public-key schemes and key-exchange schemes. Signature schemes can be understood as the category which enables correspondents to validate each others authenticity. We will not elaborate on this category as it

exceeds the scope of this research. Public-key schemes are comparable to the symmetric cryptography introduced in the previous section. They differ in assuming not one key, but a key pair. This key pair consists of a private and a public key, which are designed such that they are linked mathematically. This enables the parties to communicate in a protected way without the need to exchange a secret, because no secret key needs to be transmitted. The public key is used as the key for encryption. The only manner to decrypt the ciphertext is to use the corresponding private key. An illustration can be seen in Figure 2.4.

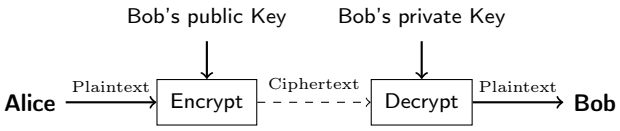

FIGURE 2.4: A brief model of a public-key scheme.

Key-exchange schemes are used to specifically solve the problem of exchanging the key. This category resides between the signatures and the public-key schemes, because it requires authenticated correspondents and actually performs the task of linking the keys mathematically. The idea is to use the asymmetric properties of the public-key schemes to encapsulate a key which can be used for further symmetrically secured communication. Therefore, this category is also referred to as 'key encapsulation'. The schemes consist of public parameters called system parameters, Alice and Bob both compute private secrets and compute their own public key with the system parameters and their corresponding private secrets. Then they exchange their public keys and derive a shared secret from the correspondents public key and their own secret. An illustration can be seen in Figure 2.5.

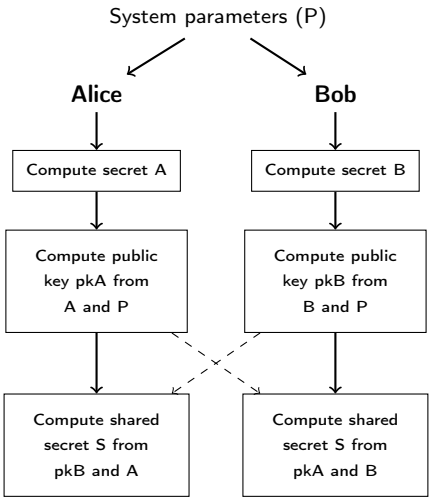

FIGURE 2.5: A brief model of a synchronous key-exchange scheme.

In the case of a key exchange we can distinguish between actively and passively secure key exchanges. The active or passive refers to the capabilities

of the assumed adversary. A passive adversary can record and analyze messages which are send via an insecure channel, whereas an active adversary is also assumed to be able to modify or inject messages [62]. An actively secure key exchange makes its security claims with regards to an active adversary, while a passively secure key exchange makes its security claims only with regards to a passive adversary. A second way to distinguish key exchanges further regards their synchronicity. Either the computation of the public keys can be performed by both participants synchronously as depicted in Figure 2.5 or the computation of the private keys depend on each other. We will discuss this asynchronous key exchange in more detail in Section 2.4. A third way to distinguish key exchanges even further is the reusability of the system parameters and the derived shared secret.

The construction of asymmetric-cryptographic schemes introduces a new attack vector, compared to symmetric cryptography, namely the computational effort needed to reconstruct the private key or private secret from the publicly known information, e.g. the system parameters and the public keys. To protect asymmetric-cryptographic schemes against this attack vector they rely on the complexity of mathematical problems. At the moment, the most commonly used schemes are based on either the factorization problem or the discrete-logarithm problem.

The factorization problem, on which RSA-like asymmetric cryptographic schemes are based [74] relies on the fact that multiplying two integers is easy while finding the prime factors of a product is computationally hard. More formally, multiplying two integers of size n can be realized in sub-quadratic time (e.g. by the Karatsuba algorithm [40] in $\mathcal{O}(n^{\log 3})$ or the Schönhage-Strassen algorithm [75] in $\mathcal{O}(n \log n \log\log n)$). The fastest classical algorithm to find the prime factors of a given integer product, after years of research, is the number field sieve with a complexity of about $\mathcal{O}(e^{1.9(\log n)^{\frac{1}{3}}(\log\log n)^{\frac{2}{3}}})$ [42]. Thus prime factorization is still sub-exponential in n and therefore impractical to compute if n is large enough.

The discrete-logarithm problem (DLP) relies on the fact that exponentiation is easy while finding the logarithm is computationally hard. More formally exponentiation of an integer base of size n with an exponent k is equivalent to k multiplications, thus it consists of k-times the complexity of one multiplication. The fastest classical algorithm to find the logarithm of a given integer product is, after years of research, also the number field sieve with its complexity of about $\mathcal{O}(e^{1.9(\log n)^{\frac{1}{3}}(\log\log n)^{\frac{2}{3}}})$ [42]. Finding the discrete logarithm is therefore as well sub-exponential in n and consequently also impractical to compute for a large enough parameter n. However, if the DLP is defined on different groups, the complexity increases. An example are elliptic-curve-based asymmetric-cryptographic schemes [56]. The best known algorithm to solve the DLP defined over elliptic curves is the Pollard-ρ algorithm [68] with a time complexity of $\mathcal{O}(n^{\frac{1}{2}})$ [8], for n being the group size.

2.2.4 Quantum Computation

Quantum computers make use of quantum-mechanical phenomena. Classical computers operate on so called bits. A bit is the measurement of information as Shannon described it [76]. One bit can take the state 0 or 1 and thus classical computers perform their computations in the Boolean domain. In the case of quantum computers the computations are performed on so-called quantum-bits or qubits for short. We will use the term 'qubits' in the remainder of this thesis. Qubits can also take the (quantum) states 0 or 1. However, qubits can also take the state of a superposition. Superposition is the principle that additions of quantum states form a new valid quantum state. Qubits can be represented in the Dirac notation [19] as $|0\rangle$ and $|1\rangle$ for the case that a conversion to the Boolean domain always returns the 0 bit, or the 1 bit respectively. A superposition would be denoted as $|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$, for α and β being probability amplitudes, which would be represented by complex numbers.

Quantum computation is a concept which is known to the literature since 1980, when Benioff and Feynman made the first publications in this field of research [4, 27]. Even though the new field of science concerned with quantum computations came a long way since then, it still remains hard to construct a universal quantum computer. A wide variety of claims are made regarding the amount of qubits and the size of numbers factored, as the concept of factorization is relevant to cryptanalysis which was pointed out in the previous chapter. The company D-Wave, receiving investments from Google and the NASA, claims to have built a quantum computer consisting of 1000 qubits [22]. Dattani and Bryan claim to have factored 56153 with only 4 qubits [15] and Xu et al. [83] claim to have factored 143. However, these factorizations are based on either side effects or a method called 'adiabatic quantum computation', which does not provide mathematical guarantees if the numbers increase. In the case of D-wave the problem is that their so called 'quantum computer' is not universal and even fails to produce a quantum speed-up if the size of the problem increases [10]. The largest number factored on a universal quantum computer is 21, by Martín-López, Laing, Lawson, Alvarez, Zhou and O'Brien [49] who additionally reduce the qubits required by $\frac{2}{3}$. The reduction of qubits was realized by recycling the qubits. This recycling of qubits points into a possible direction where fewer qubits are needed (only $\log n+1$ qubits are required). However, factorization performed on universal quantum computers is still an open problem for reasonably large numbers. Researchers from IBM estimate the arrival of large scale quantum computers within the next 15 years [81].

It is unclear how far the research genuinely is, since the development of large scale quantum computers will have a severe impact on contemporary cryptography as we will point out in Section 2.2.4.1. A document published in the context of the Snowden revelations shows that the NSA classifies the information about "Non-technical details (e.g., scheduling) regarding NSA-conducted or -sponsored classified quantum computer research" as confidential and technical details as well as "the existence of a specific classified quantum computer research project" as secret [60]. Additionally,

the option of higher classification is explicitly mentioned regarding specific projects.

2.2.4.1 Quantum Cryptanalysis

The reason for the strict classification of information regarding quantum-computer research can be explained to a certain degree by the impact quantum computers have on contemporary cryptography. The two most well known names related to quantum algorithms are Shor and Grover, who both developed theoretical quantum algorithms.

Shor developed an algorithm known as Shor's algorithm in 1994 [77]. It is a so called quantum algorithm, indicating that it is designed for a quantum computer. It combines a classical and a quantum part and makes use of the quantum Fourier transform to extract the required information from the superposition of quantum states. Shor's algorithm is designed to perform prime factorization. The groundbreaking aspect of Shor's algorithm is that it runs in polynomial time, to be explicit it can factor a number of size n into its prime factors in $\mathcal{O}((\log n)^2(\log\log n)(\log\log\log n))$. Shor [77] elaborates that his algorithm can also be used to efficiently solve the discrete logarithm.

Grover published an algorithm known as Grover's algorithm in his paper 'A Fast Quantum Mechanical Algorithm for Database Search', in 1996 [32]. It was intended, as the title of Grovers paper suggests, to search databases. Classical algorithms need at least $\mathcal{O}(n)$ to search disordered domains of size n, whereas Grovers quantum algorithms could perform the task in $\mathcal{O}(n^{\frac{1}{2}})$. It provides therefore only a quadratic speedup, but will none the less affect cryptographic primitives.

The subfield of symmetric cryptography is threatened only mildly by the quantum algorithms known so far. Due to finding the key being a search problem, Grovers algorithm can be used for quadratic speedups. This implies the need of the security parameters to be adapted accordingly. Papers as [39, 44] performed research on the quantum resistance of symmetric cryptographic schemes, however, they suggest that more research is required in this field of study.

The subfield of asymmetric cryptography faces more severe consequences due to the possibility of quantum algorithms used in cryptanalysis. In April 2016 Dridi and Alghassi used a quantum computer to factor all bi-primes up to 200 099 [20], which is still far away from being able to break real world applications of cryptographic primitives based on the factorization problem, but it illustrates that the scientific research is closing in on the problem. Most asymmetric cryptography on the Internet used today is either based on the problem of prime factorization (e.g. RSA, Rabin) or the discrete-logarithm problem (e.g. Diffie-Hellman, Elliptic Curves), which both can be computed in polynomial time by quantum computers. Those primitives form the backbone of secure and private communication of the Internet today. Proos and Zalka [71] showed that the number of qubits needed to break RSA are twice the amount of the key size, thus

to break RSA-2048 an amount of 4096 qubits would be needed. To break elliptic curves approximately six times the amount of the key size is needed, thus to break ECC-256 an amount of 1500 qubits would be needed.

2.2.4.2 Post-Quantum Primitives

"Is cryptography dead?", this question was asked by Bernstein at the beginning of the book Post-Quantum Cryptography [6]. The book also provides the answer; no it is not. Only certain primitives are threatened by the development of quantum computers and the quantum algorithms known today, e.g. the previously mentioned primitives based on integer factorization and the discrete-logarithm problem. As pointed out earlier in this section, most symmetric cryptographic primitives do survive a potential development of a large scale quantum computer by adapting their parameters. Further promising primitives, are

Hash based cryptography e.g. Merkle's hash trees [53],

Code based cryptography e.g. McEliece's hidden Goppa code [51],

Multivariate quadratic equation based cryptography e.g. Patarin's HFE [64],

Supersingular isogenies e.g. Feo's, Jao's and Plût's proposal [16] ,and

Lattice based cryptography e.g. NTRU [35] or RLWE based approaches.

Within this thesis we focus on the last mentioned primitive, lattice based cryptography based on RLWE.

2.3 Lattices

A lattice is a set of vectors which could be visualized as a periodically spaced array of points in the two-dimensional plane (see Figure 2.6).

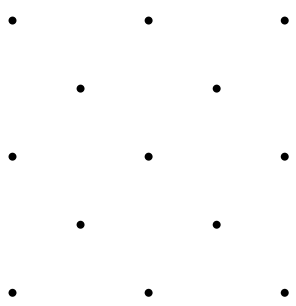

FIGURE 2.6: A part of a lattice in the two-dimensional plane.

Figure 2.6 shows an example of a lattice in the two-dimensional plain $\mathbb{R}^2$. In general, a lattice is defined as being a discrete additive subgroup in $\mathbb{R}^n$.

It can be generated from a basis $B = \{\mathbf{b}_0, \ldots, \mathbf{b}_{n-1}\}$, where $\mathbf{b}_i$ are linear independent vectors in $\mathbb{R}^n$, for $i \in (0, \ldots, n-1)$. A lattice $\mathcal{L}(B)$ is the following set of vectors:

$$\mathcal{L}(\mathbf{b}_0, \ldots, \mathbf{b}_{n-1}) = \left\{ \sum_{i=0}^{n-1} a_i \mathbf{b}_i \middle| a_i \in \mathbb{Z} \right\}$$

Different bases can define the same lattice. In Figure 2.7 two different Bases can be seen. The basis $B_0 = \{(0,1), (\frac{1}{2}, \frac{1}{2})\}$ is depicted with dashed lines. The second basis $B_1 = \{(-\frac{1}{2}, \frac{1}{2})), (-\frac{1}{2}, -\frac{1}{2})\}$ is depicted with continuous lines.

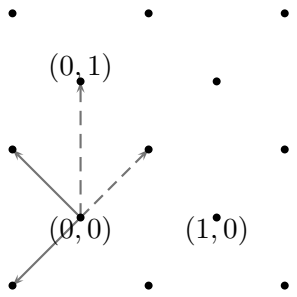

FIGURE 2.7: Two bases of the lattice, B_0 and B_1.

It is crucial to emphasize that the same lattice can be defined by various bases. This fact provides the logical precondition for the first of the problems presented in the following.

2.3.1 Lattice Problems

Various problems can be defined in the scope of lattices, however, we will only focus on four of them, being the relevant problems for the remainder of this thesis. We will focus on the shortest vector problem (SVP) and the closely related closest vector problem (CVP) of which we additionally define gap versions and approximation versions. We then narrow down on the LWE problem and the RLWE problem.

2.3.1.1 Shortest Vector Problem (SVP)

The SVP was formulated by Dirichlet in 1842; it is the search problem to find a shortest non zero vector $\mathbf{s}$ in the lattice $\mathcal{L}(B)$ given a basis B, i.e. a vector $\mathbf{s} \in \mathcal{L}(B) \backslash 0$ such that $\forall \mathbf{t} \in \mathcal{L}(B) \backslash 0$, $\|\mathbf{s}\| \leq \|\mathbf{t}\|$ [55]. For $\|\cdot\|$ being the ℓ_2 norm defined for a vector $\mathbf{x} \in \mathbb{R}^n$ as $\|\mathbf{x}\| = (\sum_{i=1}^{n} x_i^2)^{\frac{1}{2}}$.

The decision version of the SVP problem is named GapSVP and translates to asking the question whether a vector $\mathbf{s}$ exists that is shorter than a parameter d. It gets the basis B and the maximum length d as parameters and returns a binary answer. It return 'yes' if $\|\mathbf{s}\| \leq d$ or 'no' if $\|\mathbf{s}\| >$

$g(n) \cdot d$ [31]. In all other cases the algorithm returns an error. The function g defined as a function of the dimension is called the gap-function and provides the name of the decision version.

The α-approximation version of the SVP and the GapSVP is expressed by indexing the problem with an α (e.g. SVP_α, GapSVP_α) and describes the search or decision version of the problem, which only needs to be solved approximately by a factor of α.

2.3.1.2 Closest Vector Problem (CVP)

The CVP is the search problem to find the closest lattice point $\mathbf{z}$ of the lattice $\mathcal{L}(B)$ to a non zero input vector $\mathbf{v}$, given vector $\mathbf{v}$ and the basis B. We define the distance function $dist(\mathbf{v}, \mathbf{z})$ to be the Eucildean distance between the vectors $\mathbf{v} \in \mathbb{R}^n$ and $\mathbf{z} \in \mathcal{L}(B)$ [78]. From this we can define the closest vector to $\mathbf{v}$ regarding the lattice $\mathcal{L}(B)$ to be:

$$dist(\mathbf{v}, \mathcal{L}(B)) = \min_{\mathbf{z} \in \mathcal{L}(B)} \{dist(\mathbf{v}, \mathbf{z})\}.$$

A visualization can be seen in Figure 2.8.

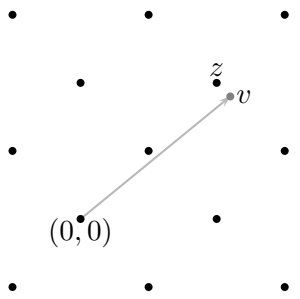

FIGURE 2.8: An example of a CVP with $\mathbf{z} = (1, 1)$ and $\mathbf{v} = (1.1, 0.9)$.

The decision version of the CVP problem is named GapCVP and translates to asking the question whether a vector $\mathbf{v}$ has a distance to the lattice $\mathcal{L}(B)$ that is shorter than a parameter d. It gets the basis B, the vector $\mathbf{v}$ and the maximum length d as parameters and returns a binary answer. It return 'yes' if $dist(\mathbf{v}, \mathcal{L}(B)) \leq d$ or 'no' if $dist(\mathbf{v}, \mathcal{L}(B)) > g(n) \cdot d$ [31]. In all other cases the algorithm returns an error. The function g defined as a function of the dimension is called the gap-function and also provides the name of the decision version for CVP.

The α-approximation version of the CVP and the GapCVP are again expressed by indexing the problem with an α (e.g. CVP_α, GapCVP_α) and also describes the search or decision version of the problem, which only needs to be solved approximately by a factor of α.

2.3.1.3 Complexity of SVP and CVP

In 1981 van Emde Boas [24] proposed a conjecture that the GapSVP belongs to the class of **NP**-hard problems. This conjecture stayed unproven till Ajtai proved that GapSVP is **NP**-hard to solve exactly for randomized reductions, in 1997. If it is sufficient to approximate the GapSVP then this does not hold true anymore. Lenstra et al. [41] showed with the famous *LLL* algorithm that it is possible to approximate GapSVP_α in polynomial time within a factor $\alpha = \mathcal{O}((\frac{2}{\sqrt{3}})^n)$. Goldreich and Goldwasser [31] went further claiming that the SVP is unlikely to be **NP**-hard for factors $\alpha = \mathcal{O}(\sqrt{\frac{n}{\log n}})$. However, Micciancio showed in [54] that GapSVP_α stays **NP**-hard for $\alpha \leq \sqrt{2}$.

In 1981 van Emde Boas [24] further proved that the GapCVP belongs to the class of **NP**-hard problems. Regarding the complexity analysis of the GapCVP_α, the numbers from GapSVP_α can be mapped to the GapCVP_α, because Goldreich et al. [30] showed that any algorithm efficiently approximating GapCVP_α also efficiently approximates GapSVP_α (for the same factor α and with the same computational effort). However, it is unclear whether the opposite holds true. Micciancio and Feige showed in [26] that even with pre-processing GapCVP_α stays **NP**-hard for $\alpha < \sqrt{\frac{5}{3}}$.

2.3.1.4 Learning-With-Errors (LWE) Problem

In 2005 Regev published a paper with the title 'On Lattices, Learning with Errors, Random Linear Codes, and Cryptography' [72] . It introduces the concept of the LWE problem for lattices, which is a generalization of the learning-parity-with-noise (LPN) problem. In his paper Regev also provided a quantum proof that solving the LWE problem implies an efficient algorithm to solve GapSVP_α for $\alpha = n^{1.5}$. The parameters of this scheme are n, which defines the security and the modulus $q \geq 2$. If the modulus is chosen to be equal to two and the error is chosen to be Bernoulli noise, the LWE problem becomes an instance of the LPN problem. Regev also suggests an asymmetric cryptographic scheme based on his contributions. Ding, Xie en Lin published a key-exchange scheme based on LWE [18]. Peikert presented an even simpler cryptosystem with tighter security claims based on his proof that certain variants of the SVP problem can be classically reduced to LWE problems [66]. As pointed out in Section 2.1, two categories exist for computational problems like the LWE problem, a search version and a decision version.

Search Version of LWE. In the search version of the LWE problem pairs of vectors $(\mathbf{a}_i, \mathbf{b}_i)$ are given, for which holds that $\mathbf{a}_i \leftarrow \mathbb{Z}_q^n$ and $\mathbf{b}_i = \langle \mathbf{s}, \mathbf{a}_i \rangle + e_i$. The task is to find the secret vector $\mathbf{s} \in \mathbb{Z}_q^n$. The error e_i is chosen from a probability distribution χ defined on $\mathbb{Z}$. The $\leftarrow$ operand denotes that the vector $\mathbf{a}_i$ is set to an uniformly random sample from $\mathbb{Z}_q^n$. The idea is to determine $\mathbf{s}$ from several samples of the following form:

$$\begin{aligned}\mathbf{a}_1 \in \mathbb{Z}_q^n,\ \mathbf{b}_1 =& \langle \mathbf{s}, \mathbf{a}_1 \rangle + e_1 \\ \mathbf{a}_2 \in \mathbb{Z}_q^n,\ \mathbf{b}_2 =& \langle \mathbf{s}, \mathbf{a}_2 \rangle + e_2 \\ &\vdots\end{aligned}$$

Without the error e_i being added to the inner product of $\mathbf{s}$ and $\mathbf{a}_i$, the task could be efficiently solved by Gaussian elimination. Since there is an error added, Gaussian elimination increases the problem to unmanageable levels, since it takes linear combinations of n equations [73].

Decision Version of LWE. In the decision version of the LWE problem the task is to distinguish between two pairs $(\mathbf{a}_i, \mathbf{b}_i)$ and $(\bar{\mathbf{a}_i}, \bar{\mathbf{b}_i})$ for which holds that $\mathbf{a}_i \in \mathbb{Z}_q^n$ and $\mathbf{b}_i = \langle \mathbf{s}, \mathbf{a}_i \rangle + e_i$, and $\bar{\mathbf{a}_i}, \bar{\mathbf{b}_i} \in \mathbb{Z}_q^n$. In words; Decide for a given pair if the second vector is created from an inner product of the first vector with some secret vector added to some error, or if the second vector is uniformly random.

Regev proved in [72] that GapSVP_α can be reduced to LWE for $\alpha = \tilde{\mathcal{O}}(\frac{n}{a})$ in the worst case. Regarding this reduction the chosen parameters are two integers n and p, and $a \in (0, 1)$ such that $ap > 2\sqrt{n}$. Regev sets the parameters for his cryptographic application to $p = \mathcal{O}(n)$ and $a = \frac{1}{\sqrt{n}\log^2 n}$. This implies, that solving an instance of the LWE problem is at least as hard to solve, in terms of complexity, as it is to solve a $\text{GapSVP}_{\tilde{\mathcal{O}}(\frac{n}{a})}$ instance.

When used for cryptographic primitives the LWE problem is, however, inefficient. The major inefficiency is the key size. An instance of the LWE problem with parameters comparable to the security of AES-128 would require a key size larger than one megabyte [46].

2.3.1.5 Ring-Learning-With-Errors (RLWE) Problem

To make the concept of the LWE problem practical for cryptographic application, the problem of the key size needs to be solved. RLWE provides a solution to this at the price of introducing more structure. The key size gets shrunk by a factor of at least 200 and is about two to five kilobytes (depending on the choice of parameters). Thereby the lower bound of 2KB is 4 times larger than a key for RSA-4096. As additional benefit, the matrix multiplication gets faster due to the ring structure of the problem.

The introduction of more structure changes the security claims made by the reductions mentioned in the previous section. However, the best known attacks against RLWE do not exploit the ring structure [1]. Peikert, Regev and Lyubashevsky additionally provide a reduction for RLWE resulting in 'strong hardness claims' [48].

Subsequently to the explanation of how the inefficiency of the LWE problem gets solved and to the short summary of its hardness, we will continue with the definition of RLWE based on the elaborations of Regev [73]. The

quotient ring $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^n + 1)$ needs to be introduced, which can be read as the ring of polynomials modulo $X^n + 1$ with each coefficient being reduced modulo q. The samples provided for the RLWE problem are of the form $(\mathbf{a}, \mathbf{b} = \langle \mathbf{s}, \mathbf{a} \rangle + e) \in \mathcal{R}_q \times \mathcal{R}_q$ for $\mathbf{a} \in \mathcal{R}_q$ being chosen uniformly, $\mathbf{s} \in \mathcal{R}_q$ being a fixed secret and e being chosen independently from some error distribution.

2.3.1.6 RLWE based Key-Encapsulation Mechanism (KEM)

On the basis of the concept of the RLWE problem, Peikert proposed an ephemeral and passively-secure key-encapsulation mechanism (KEM) in his paper 'Lattice cryptography for the Internet' [65]. Based on the newly introduced error-reconciliation method, one component in $\mathcal{R}_q$ could be replaced by a more compact component in $\mathcal{R}_2$. The underlying insight is that no explicitly chosen key needs to be transmitted, but just an approximation which allows both parties to derive the same, secure ephemeral session key by error-reconciliation. The attributes of the key exchange, 'passively' and 'ephemeral' categorizes the key exchange with regards to the first and third criteria to distinguish a key exchange we described in Section 2.2.3. The key exchange makes its claims only about passive adversaries and the key material must not be reused. The second way to distinguish a key exchange we described in Section 2.2.3 is its synchronicity. The KEM proposed by Peikert is an asynchronous key exchange as both parties cannot compute their secrets on their own but depend on the computations of one another. An illustration of the adapted model for an asynchronous key exchange can be seen in Figure 2.9.

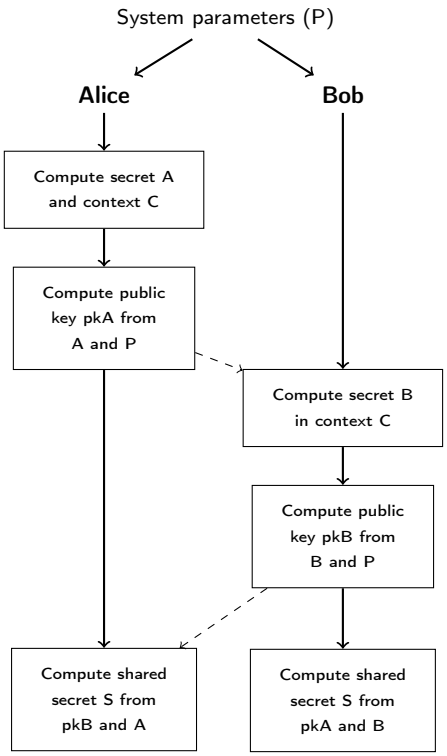

FIGURE 2.9: A brief model of an asynchronous key-exchange scheme.

In the case of the KEM by Peikert the system parameters are q, n, and χ. Alice's public key is $\mathbf{b} = \mathbf{as} + \mathbf{e}$ and Bob's public key is $\mathbf{u} = \mathbf{as}' + \mathbf{e}'$ combined with the reconciliation information $\mathbf{v}'$ as can be seen in Protocol 1.

Parameters: q, n, χ KEM.Setup(): $a \xleftarrow{\$} \mathcal{R}_q$		
Alice (server)		**Bob (client)**
KEM.Gen(a):		KEM.Encaps(a, b):
$\mathbf{s}, \mathbf{e} \xleftarrow{\$} \chi$		$\mathbf{s}', \mathbf{e}', \mathbf{e}'' \xleftarrow{\$} \chi$
$\mathbf{b} \leftarrow \mathbf{as} + \mathbf{e}$	$\xrightarrow{\mathbf{b}}$	$\mathbf{u} \leftarrow \mathbf{as}' + \mathbf{e}'$
		$\mathbf{v} \leftarrow \mathbf{bs}' + \mathbf{e}''$
		$\hat{\mathbf{v}} \xleftarrow{\$} dbl(v)$
KEM.Decaps($\mathbf{s}(\mathbf{u}, \mathbf{v}')$):	$\xleftarrow{\mathbf{u},\mathbf{v}'}$	$\mathbf{v}' = \langle \hat{\mathbf{v}} \rangle_2$
$\mu \leftarrow rec(\mathbf{us}, \mathbf{v}'))$		$\mu \leftarrow \lfloor \hat{\mathbf{v}} \rceil_2$

PROTOCOL 1: Peikerts KEM as published in [65]. The round function $\lfloor \cdot \rceil_2$, the cross-round function $\langle \cdot \rangle_2$, the randomized function $dbl(\cdot)$ and the reconciliation function $rec(\cdot, \cdot)$ are used according to the specifications in the original paper.

As pointed out in the introduction of this section the result of the combination of the private and public information of this scheme is only an approximation of the ephemeral key. For Alice it is the ring element $us = ass' + e's$

and for Bob it is the ring element $v = bs' + e'' = ass' + es' + e''$. With the help of the reconciliation information send by Bob, Alice and Bob are able to use the error-reconciliation method described in [65]. By applying the error reconciliation on their ring element, both are able to derive the ephemeral session key μ (being the shared secret S).

The $\xleftarrow{\$}$ infix operator used in Protocol 1 must be distinguished for three cases depending on the second operand to understand the protocol and to keep this thesis consistent with the NewHope paper [1]. The first is the uniform choice of coefficients for the second operand being R_q. The interpretation of $\mathbf{a} \xleftarrow{\$} \mathcal{R}_q$ is that all coefficients of the first operand $\mathbf{a}$ are chosen uniformly from R_q. The second is the case where the second operand is the error distribution χ. The interpretation of $x \xleftarrow{\$} \chi$ is that the first operand x gets sampled as $x \in \mathcal{R}$ according to χ. The third is the case where the second operand is a probabilistic algorithm $\mathcal{A}$. The interpretation of $y \xleftarrow{\$} \mathcal{A}$ is that the output of $\mathcal{A}$ gets assigned to the first operand y and $\mathcal{A}$ is running with randomly chosen coins. We see the occurrence of all three cases in Protocol 1. In the setup phase the coefficients of $\mathbf{a}$ are chosen uniformly from R_q, Alice and Bob both sample the errors from χ, and the output of the algorithm $dbl(\cdot)$ running with randomly chosen coins is assigned to $\hat{\mathbf{v}}$.

To put all three criteria to categorize a key exchange together, the KEM proposed by Peikert is passively secure, asynchronous and does not allow the reuse of the exchanged key. If instantiated correctly the major benefit of this key exchange is that it is a post-quantum key exchange and thus guarantees forward secrecy for communication protected against adversaries being equipped with storing capacities today and quantum computers in the future. For that reason instantiations of this KEM are discussed as concepts for a hybrid handshake in the Tor project [79]. However, Peikert does not provide specific recommendations for the parameter choices nor an actual implementation. Therefore, the next step is to have a look into actual instantiations.

2.4 Instantiations of the RLWE Problem based KEM

Within this subsection we will introduce two instantiations of Peikert's KEM based on the RLWE problem. The first instantiation was published in 2015 by Bos, Costello, Naehrig and Stebila [7]. We chose, again for consistency with the original NewHope paper, to name the instantiation accordingly to the first letters of the authors, BCNS. The second instantiation was first published in 2015 and revised in 2016 by Alkim, Ducas, Pöppelmann and Schwabe [1] under the name NewHope. This second instantiation is also the main focus of this thesis as we implemented it on the Cortex-M0.

2.4.1 BCNS

Bos, Costello, Naehrig and Stebila published the first instantiation of Peikert's RLWE-based passively secure KEM in 2015 [7]. The goal was to provide a 128 bit security level. The parameters they chose were $n = 1024$ and $q = 2^{32} - 1$. As discrete Gaussian error distribution they chose $\chi = D_{\mathbb{Z},\sigma}$ for $\sigma = \frac{8}{\sqrt{2\pi}} \approx 3.192$. Since the instantiation is based on the KEM proposed by Peikert, the ephemeral session key is only received after error correction. This results in a probability of $2^{-2^{17}}$ for the key not being equal for Alice and Bob, the participants of the key agreement. The messages send between Alice and Bob are at least $\log_2(q)n = 32$ kilobytes for the message $\mathbf{b}$, a ring element, which Alice sends to Bob and 33 kilobytes for the answer $(\mathbf{u}, \mathbf{v}')$ consisting of the ring element $\mathbf{u}$ and $\mathbf{v}'$ being in $\mathcal{R}_2$. The cycle counts on an Intel Haswell architecture as presented by [1] are $\approx 2\,958\,995$ for the server side and $\approx 3\,995\,977$ for the client side of the key exchange.

2.4.2 NewHope

Alkim, Ducas, Pöppelmann and Schwabe published the second instantiation of Peikert's RLWE-based passively secure KEM in 2015 (after BCNS) and a revised version in 2016 [1]. Compared to the BCNS approach it provides faster computation, stronger security claims, and a noteable decrease in key and message size. For the detailed security claims we refer to [1], where the authors provided a detailed security analysis of all known and presupposed quantum algorithms for solving the underlying SVP. The claim the authors make is to provide a security level of '128 bits of post-quantum security with a comfortable margin' [1]. As an additional security enhancement the authors of NewHope decided to generate the system parameter $\mathbf{a}$ freshly in every run of the key exchange. This choice prevents backdoors and all-of-the-price-of-one attacks for the price of a slight computational overhead. The parameters they chose were $n = 1024$ and $q = 12289 < 2^{14}$, which obviously improves the efficiency, but also the security. As error distribution they did not chose the discrete Gaussian distribution, but decided to use a centered binomial distribution Ψ_k for $k = 16$. We will elaborate on the error distribution in the following subsection. Since the instantiation is also based on the KEM proposed by Peikert, the ephemeral session key is only received after error correction. This results in a probability of less than 2^{-60} for the key being not equal for Alice and Bob, the participants of the key agreement.

Furthermore, the authors of NewHope changed the reconciliation method and proposed a new error-correction approach. The insight that led to the changes made was that if 256 bits need to be decoded into $n = 1024$ coefficients, four coefficients can be used to decode one bit. This yields an increased error-resilience which allowed them to use larger noise and thus increase the overall security.

Additionally, they chose to specify encodings of the polynomials in the domain of the Number Theoretic Transform (NTT), which eliminates some

of the NTT computations necessary. We will discuss the error-reconciliation and the NTT in detail in the following sections. The messages send between Alice and Bob are 1824 bytes for the message $\mathbf{b}$, an element in the NTT domain with the encoded seed for $\mathbf{a}$, which Alice sends to Bob and 2 kilobytes for the answer $(\hat{\mathbf{u}}, \mathbf{r})$ consisting of $\hat{\mathbf{u}}$ in the NTT domain with the encoded $\mathbf{r}$ being in $\mathcal{R}_4$. The cycle counts are $\approx 358\,234$ for the server side and $\approx 402\,058$ for the client side of the key exchange. These cycle counts are on the same architecture for the server side computations by a factor 8.26 faster and for the client side computations by a factor of 9.94 faster compared to the BCNS instantiation. The scheme of the NewHope key exchange can be seen in Protocol 2.

Parameters: $q = 12289 < 2^{14},\ n = 1024$ Error distribution: Ψ_{16}		
Alice (server)		**Bob (client)**
$seed \xleftarrow{\$} \mathsf{SHA3\text{-}256}(\{0,\ldots,255\}^{32})$		
$\hat{\mathbf{a}} \leftarrow \mathsf{Parse}(\mathsf{SHAKE\text{-}128}(seed))$		
$\mathbf{s}, \mathbf{e} \xleftarrow{\$} \psi_{16}^n$		$\mathbf{s}', \mathbf{e}', \mathbf{e}'' \xleftarrow{\$} \psi_{16}^n$
$\hat{\mathbf{s}} \leftarrow \mathsf{NTT}(\mathbf{s})$		
$\hat{\mathbf{b}} \leftarrow \hat{\mathbf{a}} \circ \hat{\mathbf{s}} + \mathsf{NTT}(\mathbf{e})$	$\xrightarrow[1824\text{ Bytes}]{m_a = \mathsf{encodeA}(seed, \hat{\mathbf{b}})}$	$(\hat{\mathbf{b}}, seed) \leftarrow \mathsf{decodeA}(m_a)$
		$\hat{\mathbf{a}} \leftarrow \mathsf{Parse}(\mathsf{SHAKE\text{-}128}(seed))$
		$\hat{\mathbf{t}} \leftarrow \mathsf{NTT}(\mathbf{s}')$
		$\hat{\mathbf{u}} \leftarrow \hat{\mathbf{a}} \circ \hat{\mathbf{t}} + \mathsf{NTT}(\mathbf{e}')$
		$\mathbf{v} \leftarrow \mathsf{NTT}^{-1}(\hat{\mathbf{b}} \circ \hat{\mathbf{t}}) + \mathbf{e}''$
$(\hat{\mathbf{u}}, \mathbf{r}) \leftarrow \mathsf{decodeB}(m_b)$	$\xleftarrow[2048\text{ Bytes}]{m_b = \mathsf{encodeB}(\hat{\mathbf{u}}, \mathbf{r})}$	$\mathbf{r} \xleftarrow{\$} \mathsf{HelpRec}(\mathbf{v})$
$\mathbf{v}' \leftarrow \mathsf{NTT}^{-1}(\hat{\mathbf{u}} \circ \hat{\mathbf{s}})$		$\nu \leftarrow \mathsf{Rec}(\mathbf{v}, \mathbf{r})$
$\nu \leftarrow \mathsf{Rec}(\mathbf{v}', \mathbf{r})$		$\mu \leftarrow \mathsf{SHA3\text{-}256}(\nu)$
$\mu \leftarrow \mathsf{SHA3\text{-}256}(\nu)$		

Protocol 2: The NewHope protocol including NTT and NTT^{-1} computations and sizes of exchanged messages; $\circ$ denotes pointwise multiplication. (Inspired by [1])

2.4.2.1 NewHope Error Distribution

As described above do the authors of NewHope use the centered binomial distribution Ψ_{16} as error distribution. The centered binomial distribution Ψ_k has a mean of $\mu = 0$ and a variance of $\sigma^2 = \frac{k}{2}$. For the parameter $k = 16$ this results in a standard deviation of $\sigma = \sqrt{8} \approx 2.83$. The authors of NewHope chose this error distribution to provide a faster sampling rate and to protect the error sampling against timing attacks.

However, original reductions performed by Regev [72] for LWE and Regev, Peikert and Lyubashevsky [48] for RLWE, state the security guarantees only for continuous Gaussian distributions. The authors of NewHope claim in Theorem 4.1 [1] that this is not an issue and additionally provide a proof in their appendix. The first aspect to notice is that to recover

the correct key μ the pre-hashed key ν is required, which can be seen in the last row of Protocol 2. Their proven theorem then states that any algorithm which would succeed in recovering ν with probability p would also succeed against an idealized version of the protocol with a discrete Gaussian error distribution with probability $q = \frac{p^{\frac{9}{8}}}{26}$. The attentive reader might have spotted that we switched to a discrete Gaussian distribution while the security reductions are only performed on continuous Gaussian distributions. However, in [9] Brakerski extended the security reductions to discrete Gaussian distributions.

2.4.2.2 NewHope Error-Recovery Mechanism

The error-recovery mechanism and the analog error-reconciliation approach chosen by the designers of NewHope is based on the fact that solving a CVP is easy for low dimensional lattices. It can be interpreted as Fuzzy Extractor, using helper data $\mathbf{r}$ to retrieve the same ν from slightly different $\mathbf{v}$ and $\mathbf{v}'$. The reconciliation method from NewHope uses dimension $d = 4$, thus the lattice $\hat{D}_4$ with basis $\mathbf{B} = (\mathbf{u}_0, \mathbf{u}_1, \mathbf{u}_2, \mathbf{g})$ for $\mathbf{u}_i$ being the canonical basis vectors of $\mathbb{Z}^4$ (e.g. $\mathbf{u}_0 = (1,0,0,0), \mathbf{u}_1 = (0,1,0,0), \mathbf{u}_1 = (0,0,1,0)$) and $\mathbf{g} = (\frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2})$. The Voronoi cells $\mathscr{V}$ of this lattice are 24-cells, called icositetrachorons [43]. A target point gets mapped to a cell by the closest vector algorithm, which solves the $CVP_{\hat{D}_4}$ efficiently. A detailed explanation, taken from [1] can be seen in Algorithm 1, where the rounding function is defined as $\lfloor x \rceil = \lfloor x + \frac{1}{2} \rfloor$ for $x \in \mathbb{R}$.

Algorithm 1 Solving $\text{CVP}_{\hat{D}_4}(\mathbf{v} \in \mathbb{R}^4)$

Ensure: An integer vector $\mathbf{z}$ such that $\mathbf{Bz}$ is a closest vector to $\mathbf{v}$: $\mathbf{v} - \mathbf{Bz} \in \mathscr{V}$

1: **if** $(\|\mathbf{v} - \lfloor \mathbf{v} \rceil\|_1) < 1$ **then**
2: **return** $(\lfloor v_0 \rceil, \lfloor v_1 \rceil, \lfloor v_2 \rceil, 0)^t + \lfloor v_3 \rceil \cdot (-1,-1,-1,2)^t$
3: **else**
4: **return** $(\lfloor v_0 \rfloor, \lfloor v_1 \rfloor, \lfloor v_2 \rfloor, 1)^t + \lfloor v_3 \rfloor \cdot (-1,-1,-1,2)^t$
5: **end if**

Having defined Algorithm 1 the function producing the helper data for the error-reconciliation, called HelpRec can be defined as,

$$\text{HelpRec}(\mathbf{v}, b) = \text{CVP}_{\hat{D}_4}\left(\frac{2^r}{q}(\mathbf{v} + b\mathbf{g})\right) \mod 2^r,$$

for $b \in \{0,1\}$ being a random bit, resulting in the algorithm running with a fairly chosen coin, and r being the discretezation value, which we will explain below and can be assumed to be $r = 2$. For dimension $d = 4$ this implies that $r \cdot d = 8$ bits of reconciliation information need to be transmitted per key bit.

To calculate the reconciliation a simplified algorithm needs to be introduced, which returns the result modulo $\mathbb{Z}^4$ and is called *Decode*. The definition taken from [1] can be seen in Algorithm 2.

Algorithm 2 *Decode*($\mathbf{x} \in \mathbb{R}^4/\mathbb{Z}^4$)

Ensure: A bit k such that $k\mathbf{g}$ is a closest vector to $\mathbf{x}+\mathbb{Z}^4$: $\mathbf{x}-k\mathbf{g} \in \mathscr{V}+\mathbb{Z}^4$

1: $\mathbf{v} = \mathbf{x} - \lfloor \mathbf{x} \rceil$
2: **return** 0 if $\|\mathbf{v}\|_1 \leq 1$ and 1 otherwise

Having defined Algorithm 2 the reconciliation function producing the ephemeral key before hashing ν from a vector $\mathbf{x}$ and a reconciliation vector $\mathbf{r} \in \{0,1,2,3\}^4$, called Rec can be defined as,

$$\mathsf{Rec}(\mathbf{x}, \mathbf{r}) = \mathit{Decode}(\frac{1}{q}\mathbf{x} - \frac{1}{2^r}\mathbf{B}\mathbf{r}).$$

To reduce a 1024 coefficient polynomial in $\mathcal{R}_q$ to a vector in $\hat{D}_4$ the authors of NewHope take 4 coefficients at a time creating 256 different vectors in $\hat{D}_4$.

The Two-Dimensional Depiction. The original NewHope paper elaborates the concepts of error-reconciliation on the example of the 2-dimensional lattice $\hat{D}_2$ for illustrative purposes, since it can be drawn. The lattice $\hat{D}_2$ has the basis $\{(0,1), (\frac{1}{2}, \frac{1}{2})\}$ and the Voronoi cells become diamonds for this lattice dimension. A fragment of this lattice can be seen in Figure 2.10, where the dashed lines depict the boundaries for the reconciliation vector $\mathbf{v} \in [0,1)^2 \subset \mathbb{R}$.

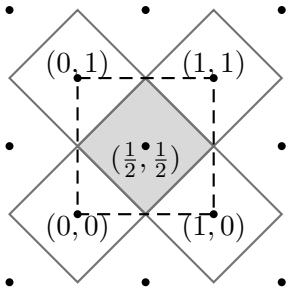

Figure 2.10: A part of the lattice $\hat{D}_2$. Taken from [1].

Since both parties of the key exchange, Alice and Bob have a slightly different vector, $\mathbf{v}_{Alice}$ and $\mathbf{v}_{Bob}$ the idea of the error-reconciliation used by the authors of NewHope is to let one of the participants, in this case Bob, send the distance between his vector and the center of the Voronoi cell as helper data or reconciliation vector $\mathbf{r}$. Alice then simply adds this vector $\mathbf{r}$ to her vector $\mathbf{v}_{Alice}$ to move towards the center of the correct Voronoi cell. This is required to produce the same output bit. It is set to 1 if the vector is in the gray Voronoi cell ($\mathbf{v} = (\frac{1}{2}, \frac{1}{2})$) or set to 0 if the vector is in the white Voronoi cell ($\mathbf{v} \in \{(0,0), (0,1), (1,0), (1,1)\}$).

However, this would require to send a full additional vector, hence did the authors of NewHope decided to use the concept of r-bit discretezation.

The concept relies on the idea of splitting each Voronoi cell into 2^{dr} subcells and sending only the information in which subcell a vector $\mathbf{v}_{Bob}$ lies. A depiction can be seen in Figure 2.11.

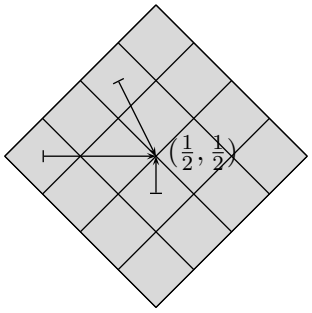

FIGURE 2.11: Discretezation of the gray Voronoi cell $(\frac{1}{2}, \frac{1}{2})$ into 2^{dr} sub-cells for the 2-dimensional lattice with a discretezation value of $r = 2$. Taken from [1].

The last aspect of the error reconciliation we need to discuss briefly, is the bias of the possible vectors. The problem is that we have q possible vectors, since q is a prime greater two it is odd and thus produces a small bias for each key bit. To resolve this issue, the authors of NEWHOPE add the term $b\mathbf{g}$ in the HelpRec function. In words, this adds the vector $(\frac{1}{2q}, \frac{1}{2q})$ with a probability $\frac{1}{2}$. The result of this has just a marginally effect on most outcomes, however, it does move $\mathbf{v}$ with probability $\frac{1}{2}$ to another Voronoi cell, which eliminates the bias. In the NEWHOPE paper [1], this concept is called blurring and a depiction can be seen in Figure 2.12.

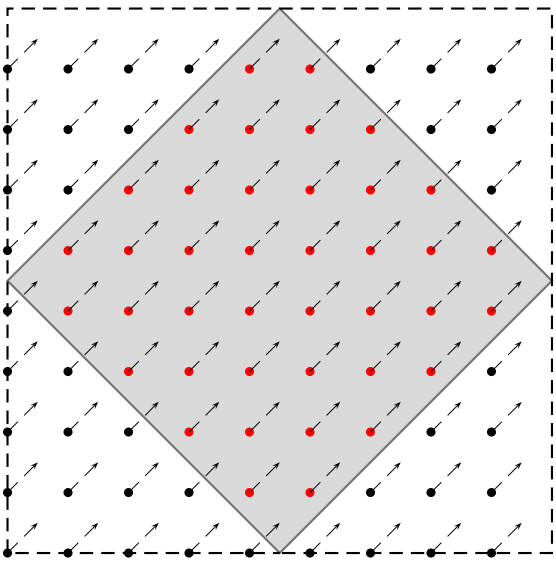

FIGURE 2.12: Possible values per vector $\mathbf{v}$ in $\mathbb{Z}_9 \times \mathbb{Z}_9$. Taken from [1].

2.4.2.3 NewHope Number Theoretic Transform

In this section we describe the NTT in detail. We will start with the most general form, the Discrete Fourier Transform (DFT) and narrow down the scope on the NTT and algorithmic optimization measures. Explicitly, we will discuss the Fast Fourier Transform (FFT) and the negative-wrapped negacyclic convolution. With these concepts as background information, we are able to describe the in-place NTT algorithm based on Gentlemen-Sande [29] butterfly operations used in the reference implementation of NewHope [1].

Discrete Fourier Transform (DFT). Following loosely the elaborations of Fürer [28], the n-point DFT can be described as linear mapping of a vector $\mathbf{x} = (x_0, x_1, \ldots, x_{n-1})$ to another vector $\mathbf{y} = (y_0, y_1, \ldots, y_{n-1})$ by $\mathbf{b} = \omega^{jk}\mathbf{a}$ for $j \geq 0$, $k \leq n-1$ with a primitive n-th root of unity ω. This mapping can be defined by

$$y_k = \sum_{j=0}^{n-1} x_j \cdot \omega^{jk}, \quad \text{for } k \in \{0, \ldots, n-1\}.$$

An n-th root of unity ω is characterized by satisfying

$$\omega^n = 1.$$

If n is also the smallest integer of $k \in \{1, \ldots, n\}$ for which holds that $\omega^k = 1$, ω is called a primitive n-th root of unity. Being a primitive n-th root of unity grants several properties from which we will introduce two:

Reduction Property If ψ is a $2n$-th root of unity, then $\psi^2 = \omega$ is an n-th root of unity.

Inverse Property $\omega^{-1} = \omega^{n-1}$.

We will relate to these properties in the following to elaborate the workings of certain constructions. For now we take a look at the DFT, which can be described as matrix multiplication, due to it being a linear mapping by

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \ldots & 1 \\ 1 & \omega & \omega^2 & \ldots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \ldots & \omega^{2(n-1)} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \ldots & \omega^{(n-1)(n-1)} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{n-1} \end{bmatrix}.$$

The inverse of the DFT, DFT^{-1}, is defined for cases in which n^{-1} the multiplicative inverse of n exists by

$$x_j = n^{-1} \sum_{k=0}^{n-1} y_k \cdot \omega^{-jk}, \quad \text{for } j \in \{0, \ldots, n-1\}.$$

Similarly to the DFT the DFT^{-1} can be described as matrix multiplication by

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{n-1} \end{bmatrix} = n^{-1} \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega^{-1} & \omega^{-2} & \dots & \omega^{-(n-1)} \\ 1 & \omega^{-2} & \omega^{-4} & \dots & \omega^{-2(n-1)} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & \omega^{-(n-1)} & \omega^{-2(n-1)} & \dots & \omega^{-(n-1)(n-1)} \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{bmatrix}.$$

From the inverse property it can be seen in the matrix representations that the inverted ω's are simply calculated by restructuring the matrix of the DFT (taking its complex conjugate transpose). For example, we can see that the second column of the DFT matrix is exactly the same as the transposed last row of the DFT^{-1} due to the inverse property.

Another fundamental aspect of the DFT is that the vector $\mathbf{x}$ can be interpreted as polynomial

$$p_x(z) = x_0 + x_1 z + x_2 z^2 + \cdots + x_{n-1} z^{n-1}.$$

If we write the sum from the DFT definition above as

$$y_k = x_0 + x_1 \omega^k + x_2 \omega^{2k} + \cdots + x_{n-1} \omega^{(n-1)k}$$

it can easily be seen that y_k is the evaluation of the polynomial $p_x(\omega^k)$. This shows that the DFT relates the coefficients to the values of a polynomial.

If we consider the interpolation theorem in Theorem 2.4.1 as described by [25], we can see that four simple steps are required to multiply two polynomials with the help of a DFT.

Theorem 2.4.1 The Interpolation Theorem for Polynomials. *Given a set of n points in the plane, $S = (x_0, y_0), (x_1, y_1), (x_2, y_2), \dots, (x_{n-1}, y_{n-1})$, such that the x_i's are all distinct, there is a unique $n-1$ polynomial $p(x)$ with $p(x_i) = y_i$, for $i \in \{0, 1, \dots, n-1\}$.*

As example we assume two polynomials p_a and p_b of size n. We pad both polynomials to size $2n$ to match the size of the resulting product polynomial $p_c = p_a \cdot p_b$. We will represent the polynomials by their corresponding vectors $\mathbf{a}$, $\mathbf{b}$ and $\mathbf{c}$. The four steps required for multiplying two polynomials with the help of a DFT are the following:

1. Evaluate the polynomial p_a in $2n$ points by applying the DFT.
2. Evaluate the polynomial p_b in the same $2n$ points by also applying the DFT.
3. Compute the $2n$ products of the evaluated points.
4. Perform an DFT^{-1} to compute p_c's coefficients from the values of the polynomial.

The points derived at the third step represent the final polynomial according to Theorem 2.4.1. The fourth step is thus needed to transform the

values to the coefficients of the resulting product polynomial. A product polynomial $p_c = p_a \cdot p_b$ can thus be computed by $\mathbf{c} = DFT^{-1}(DFT(\mathbf{a}) \circ DFT(\mathbf{b}))$ with $\circ$ denoting the pointwise multiplication. In the remainder of this thesis we will use the subscript 'multiplication' to differentiate the DFT and the $\text{DFT}_{\text{multiplication}}$.

NTT. The NTT is a specialization of the DFT defined for vectors with its elements being in the finite field. One such finite field is the quotient ring $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^n + 1)$ we introduced in Section 2.3.1.5. Multiplying two polynomials inside the quotient ring can therefore be computed as $\mathbf{c} = \mathsf{NTT}^{-1}(\mathsf{NTT}(\mathbf{a}) \circ \mathsf{NTT}(\mathbf{b}))$ for $\mathbf{a}, \mathbf{b} \in \mathcal{R}_q$. Note that a primitive n-th root of unity is guaranteed to exist if n divides $q - 1$. This is the reason for the parameter choice for NewHope, as 12289 is the smallest prime which satisfies $q \equiv 1 \mod n$ for $n = 1024$. If $\sum_{i=0}^{n-1} \omega^{ik} = 0$, ω is called a principal root of unity. Every primitive root of unity defined over a finite field is also a principal root of unity. This grants us the additional **Reflective Property** for principle n-th roots of unity, which states that $\omega^{k+\frac{n}{2}} = -\omega^k$.

Algorithmic Optimization Measures. Two optimization techniques of the NTT need to be defined, the FFT and the negative-wrapped nega-cyclic convolution.

The FFT is an algorithm to compute the DFT efficiently and can also be used to compute specializations of the DFT such as the introduced NTT. In the literature the FFT for a sequence of arbitrary length was published by Cooley and Tukey in 1965 [13], but was discovered several times before. The earliest discovery dates back to Gauss in 1809 [33]. The idea is to divide an n-point DFT, for n being divisible by 2, repeatedly by 2 until only 2-point transforms are left. The amount of divisions required is $\log n$ and defines the amount of levels the FFT needs to perform. Each division by 2 is realized by distinguishing the even and the odd elements. The first division of the DFT can therefore be rewritten as

$$\sum_{j=0}^{n-1} x_j \cdot \omega^{jk} = \sum_{m=0}^{\frac{n}{2}-1} x_{2m} \cdot \omega^{2mk} + \sum_{m=0}^{\frac{n}{2}-1} x_{2m+1} \cdot \omega^{(2m+1)k}.$$

This shows that we can rewrite the n-point DFT by two $\frac{n}{2}$-point transforms. The variable k still ranges over n values, however, only $\frac{n}{2}$ values of the parity divided sums are required to be computed as both are periodic in k with a period of $\frac{n}{2}$ as can be seen from the reflective property. This rewriting can be performed recursively $\log n$ times until only 2-point transforms are left. Different approaches can be chosen to compute the 2-point transforms. The authors of NewHope chose for an algorithm consisting of Gentlemen-Sande butterfly operations [29] which combine the two coefficients x_i and x_j in the following way:

$$\begin{bmatrix} x'_i \\ x'_j \end{bmatrix} \leftarrow \begin{bmatrix} x_i + x_j \\ \omega^t(x_i - x_j) \end{bmatrix}$$

The indices i and j are depended on the level $l \in \{0, \ldots, \log n\}$ which defines the distance $d = 1 \ll l$ with '$\ll$' representing a binary left shift. The start position of the range of the index i needs to be calculated per level as $start \in \{0, \ldots, d\}$. For each element in $start$ the index of i ranges from the value in $start$ to n with a step size of $2d$. The index j is simply $j = i+d$. The twiddle factor ω^t gets determined by the exponent $t \in \{1, \ldots, \frac{n}{2}\}$. The actual C code defining and looping over the indices can be found in Listing 3 in Section 3.4.1.

The origin of the name 'butterfly' can be understood with some creativity from the Figure 2.13, which depicts the way x_i and x_j are combined to produce the results x'_i and x'_j.

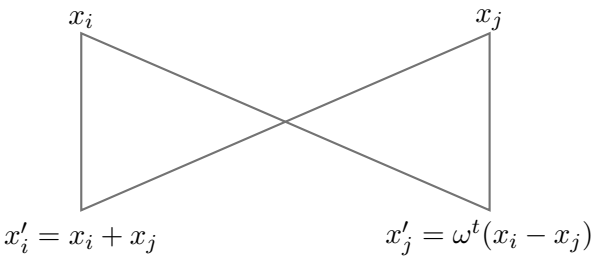

FIGURE 2.13: A depiction of the butterfly operation on two coefficients x_i and x_j.

The performance gain achieved by the FFT is that it can be computed in $\mathcal{O}(n \log n)$ whereas the naive approach to compute the DFT (or NTT) has complexity $\mathcal{O}(n^2)$. Due to the FFT splitting the input according to its parity the bits get reversed, at least if the FFT is calculated in-place. We will illustrate that with a short example. A vector $\mathbf{x} = (x_0, x_1, x_2, x_3)$ gets split into the two vectors $\mathbf{x}_{\text{even}} = (x_0, x_2)$ and $\mathbf{x}_{\text{odd}} = (x_1, x_3)$. The new bit ordering is therefore (x_0, x_2, x_1, x_3). This ordering, however, is a bit reversal of the original values as 01_2 becomes 10_2 and vice versa (00_2 and 11_2 stay the same after reordering). Therefore, such an FFT takes either bits in normal order and returns its output bits in reversed order or it takes bits in reversed order and returns its output bits in normal order.

The product polynomial p_c needs to be reduced such that $\mathbf{c} \in \mathcal{R}_q$ becomes a ring element. The negative-wrapped negacyclic convolution deals with the problem of reducing the product polynomial. From the reversal of the reduction property we derive that we can choose ψ to be a primitive $2n$-th root of unity in $\mathbb{Z}_q$ if we chose it to be $\psi = \sqrt{\omega}$. When we multiply the vectors $\mathbf{a}$ and $\mathbf{b}$ by the powers of ψ we receive two new vectors $\mathbf{a}_\psi = (a_0, a_1\psi, a_2\psi^2, \ldots, a_{n-1}\psi^{n-1})$ and $\mathbf{b}_\psi = (b_0, b_1\psi, b_2\psi^2, \ldots, b_{n-1}\psi^{n-1})$. If the resulting polynomial is multiplied by the inverted powers of ψ, we get the negative-wrapped negacyclic convolution. This optimization technique does not require the padding of the input polynomials p_a and p_b and reduces

the result $\mathbf{c}$ implicitly modulo X^n+1. We could therefore perform the multiplication of two polynomials p_a and p_b of size n and their corresponding vectors $\mathbf{a}, \mathbf{b} \in \mathcal{R}_q$ such that p_c is also of size n and $\mathbf{c} \in \mathcal{R}_q$ by computing $\mathbf{c} = (1, \psi^{-1}, \psi^{-2}, \ldots, \psi^{-(n-1)}) \circ \mathsf{NTT}^{-1}(\mathsf{NTT}(\mathbf{a}_\psi) \circ \mathsf{NTT}(\mathbf{b}_\psi))$.

The NewHope In-Place NTT. From this section forward we will refer to the FFT optimized NTT multiplication including the negative-wrapped negacyclic convolutions as $\mathsf{NTT}_{\text{multiplication}}$. The transforms will be denoted by NTT and NTT^{-1} respectively. In the case of NewHope for $n = 1024$ we deduce from the details provided above that $\log n = 10$ levels are required, each performing $\frac{n}{2} = 512$ Gentlemen-Sande butterfly operations. Each Gentlemen-Sande butterfly operation is performing its computations on two coefficients with an increasing step size per level. Therefore, also $\frac{n}{2}$ powers of ω are required. The primitive n-th root of unity is fixed by the authors of NewHope to $\omega = 49$ and consequently $\psi = \sqrt{49} = 7$. These values fix also their inverses $\omega^{-1} = 49^{-1} \mod q = 1254$ and $\psi^{-1} = 7^{-1} \mod q = 8778$. The only thing left to calculate is the scalar $n^{-1} \mod q = 12277$ to be able to compute the NTT^{-1}. Due to the choice of Gentlemen-Sande butterfly operations, the NTT and NTT^{-1} both expect bit-reversed inputs and produce normal-bit-ordered outputs.

2.4.2.4 Optimization Measures for the NTT

In the following we present four optimization techniques for calculating the $\mathsf{NTT}_{\text{multiplication}}$ which are already in place due to the decisions made by the authors of NewHope. These descriptions of the four optimization techniques will be completed with respective mathematical concepts needed to understand each technique.

The first optimization applied by the authors of NewHope is that the bit reversal required before the computation of each NTT is omitted, because all inputs are randomly generated and therefore assumed to be in bit-reversed order. The bit reversal cannot be omitted for the NTT^{-1} and needs to be computed before each occurrence.

The second optimization applied is that the reduction required after addition is replaced by a 'short Barrett reduction' [3]. The main idea behind the 'short' approach is the realization that it is not required to fully reduce the coefficient modulo q, but it is sufficient to reduce any 16-bit unsigned integer to an integer congruent modulo q of at most 14-bits [1]. Since all reductions must be performed modulo q the pre-computation necessary for the Barrett reduction can be hard-coded as parameter.

Barrett Reduction. The Barrett reduction was introduced by Barrett in 1987 [3] for implementing RSA efficiently on a standard digital signal processor. It computes $r = x \mod m$ for given x and m. Additionally it needs a pre-computation, which makes it only efficient for many reductions with a single modulus m. The pre-computation required is $\mu = \frac{b^{2k}}{m}$. By computing μ the costly and otherwise needed division is not required

anymore and can be replaced by a multiplication. The Barrett reduction can be seen in Algorithm 3 taken from [52], which assumes $b > 3$.

Algorithm 3 Barrett reduction

INPUT: positive integers $x = (x_{2k-1} \dots x_1 x_0)_b$, $m = (m_{k-1} \dots m_1 m_0)_b$ (with $m_{k-1} \neq 0$), and $\mu = \frac{b^{2k}}{m}$.
Output: $r = x \mod m$.

1. $q_1 \leftarrow \lfloor \frac{x}{b^{k-1}} \rfloor$, $q_2 \leftarrow q_1 \cdot \mu$, $q_3 \leftarrow \lfloor \frac{q_2}{b^{k+1}} \rfloor$
2. $r_1 \leftarrow x \mod b^{k+1}$, $r_2 \leftarrow q_3 \cdot m \mod b^{k+1}$, $r \leftarrow r_1 - r_2$.
3. If $r < 0$ then $r \leftarrow r + b^{k+1}$.
4. While $r > m$ do: $r \leftarrow r - m$.
5. Return r.

The third optimization applied by the authors of NewHope is to use the Barrett reduction only every second (odd) level, which they call 'lazy'. The insight which lead to this optimization is that per addition a maximum of one carry bit occurs. For 14-bit inputs this leads to a situation where the output after addition of an even level is at most 15-bit. Since the Barrett reduction can handle up to 16-bit inputs it suffices to perform it only every second level. The addition performed in an odd level could lead to results of at most 16-bit, which can still be handled by the Barrett reduction.

The fourth optimization applied is that the Montgomery representation [57] is used within the $\mathsf{NTT}_{\text{multiplication}}$. Each arithmetic operation is performed in $\mathbb{Z}_q$, therefore the Montgomery constant needs to be $> 2^{14}$. The authors of NewHope chose it to be $R = 2^{18}$. By not storing the precomputed powers of ω but their Montgomery representation ($2^{18}\omega \mod q$) the result of a multiplication with a coefficient can be reduced by the fast Montgomery reduction. Again, the randomly chosen input polynomials are assumed to be already in the Montgomery domain. The implementation of NewHope went even further by not completely reducing modulo q, but just reducing it until the result is at most 14 bits, which is sufficient for any unsigned integer in $\{0, \dots, 2^{32} - q(R-1) - 1\}$ [1]. However, this specific implementation does not work correctly for any 32-bit integer, e.g. the addition returns 0 instead of 4095 when asked to reduce the input $2^{32} - q(R-1) = 1073491969$. This is no problem, due to the possible range of input values as elaborated in [1].

Montgomery reduction. The Montgomery reduction was introduced by Montgomery in 1985 [57]. It is a method of reducing T modulo m, a positive integer. Another integer $R > m$ is needed, for which $gcd(m, R) = 1$. Montgomery described a method for computing $TR^{-1} \mod m$ without using the standard reduction algorithm. If R is chosen correctly, the Montgomery reduction can be computed very efficiently. The algorithm of the Montgomery reduction as presented in [52] can be seen in Algorithm 4.

Algorithm 4 Montgomery reduction

INPUT: integers $m = (m_{n-1} \dots m_1 m_0)_b$ with $gcd(m, b) = 1$, $R = b^n$, $m' = -m^{-1} \mod b$, and $T = (t_{2n-1} \dots t_1 t_0)_b < mR$.
Output: $TR^{-1} \mod m$.

1. $A \leftarrow T$.
2. For i from 0 to $(n-1)$ do the following:
   2.1 $u_i \leftarrow a_i m' \mod b$.
   2.2 $A \leftarrow A + u_i m b^i$.
3. $A \leftarrow \frac{A}{b^n}$.
4. If $A \leq m$ then $A \leftarrow A - m$.
3. Return A.

With this we conclude the preliminary section and hope to have provided the reader with sufficient background information to dive into the implementation details of our ARM optimized implementation of the post-quantum key exchange NewHope we will discuss in detail in the next section.

NewHope for the ARMv6-M architecture

In this section we present an efficient implementation of the NewHope key exchange on the ARMv6-M architecture. We took the reference C implementation provided by the authors of NewHope, examined the computational effort per building block, and carefully optimized them in assembly. The architectural characteristics and the optimization of the NewHope building blocks, which are described in the following subsections, form the main contribution of this Thesis.

First, we provide a short introduction to the target microcontroller and elaborate on relevant characteristics of the ARMv6-M architecture in detail. Second, we provide a summary of the core building blocks from an implementation perspective. Third, we provide a summary of essential changes to fit NewHope on the Cortex-M0 and how we realized communication. Fourth, we describe optimization techniques per building block, which are based on the structure presented Section 3.2.

3.1 The ARM Cortex-M0 (ARMv6-M)

As representative of the ARM Cortex-M family we chose the Cortex-M0 as target platform. It is the smallest processor provided by ARM [2]. It is based on the ARMv6-M architecture which makes use of a combined instruction set. It fully supports Thumb instructions and additionally provides support for a subset of the Thumb 2 instruction set. The Cortex-M0 provides 13 general purpose registers, however, only Thumb 2 instructions can make use of all of them. Thumb instructions can only make use of the first 8 registers. It additionally provides three special registers, the stack pointer `r13`, the link register `r14`, and the program counter `r15`. Each of these registers is 32-bit wide, the general word size of the Cortex-M0. The words are represented in little-endian. It allows for memory operations to load and store N registers by using the `STM` and the `LDM` instructions instead of the `STR` and the `LDR` instructions. The first take $1 + N$ cycles, while the later take 2 cycles. Further, do the first two instructions increase the pointer by default by N times 'word length in byte', which saves one more `add` instruction. Another aspect of the Cortex-M0 is that it does not provide a random number generator (RNG).

We implemented and ran our tests and cycle counts on an STM32F0 Discovery board. This board contains a STM32F051R8T6 microcontroller, which we ran at 48MHz. It provides 8KB of RAM and 64KB of flash memory. Connectivity is partly provided by the USB cable. We used it to

flash the microcontroller. To communicate with the board, a USB to TTL serial adapter is used. The ground and 3.3 Volt power cables are connected to the corresponding GPIO pins, RXD is connected to PA2 and TXD is connected to PA3. The 5 Volt cable stays unconnected.

3.2 NewHope Building Blocks

From a perspective which takes the implementation into account, one can distinguish between the following building blocks of NewHope: The generation of the public parameter $\mathbf{a}$, the sampling of noise, the computation of the NTT and the NTT^{-1} including bit reversal, the multiplication of the coefficients, the pointwise multiplication and addition, the error reconciliation and the help vector generation for the reconciliation, and hashing. In Table 3.2 the frequency per function can be seen per participant of the key exchange. We will make a change in notation at this point consistently with the change from the the overall model perspective to the actual implementation perspective. We will, in the remainder, distinguish between the server side and the client side of the key exchange, contrary to the so far used participants of the key exchange, Alice and Bob. Please note that that the server side corresponds to Alice as she initializes the key exchange due to NewHope being designed to fit into the TLS protocol.

Table 3.2: Operation counts on the client and the server side of the NewHope key exchange.

Operation	**Server-side**	**Client-side**
Generating the public parameter $\mathbf{a}$;	1	1
Sampling noise polynomials;	2	3
Computing the NTT;	2	2
Computing the NTT^{-1} with bit reversal;	1	1
Multiplying the coefficients with ψ (respectively ψ^{-1});	3	3
Computing the polynomial addition;	2	2
Computing the pointwise multiplication;	1	2
Computing the help vector $\mathbf{r}$ for error reconciliation;	0	1
Computing the error reconciliation Rec;	1	1
Hashing.	2	1

These building blocks can be grouped together according to the following five concepts:

$\mathsf{NTT}_{\text{multiplication}}$,

Error Reconciliation,

Polynomial Arithmetic,

ChaCha20, and

Keccak.

We ordered the categories according to relevance due to cycle count and frequency of the operation counts, except for the Keccak related functions, because we make use of an already available assembly implementation provided by Noekeon [14]. Regarding the categorization, we have the following structure:

The first category contains building blocks that are required to compute the $\mathsf{NTT}_{\text{multiplication}}$. This category includes the NTT, the NTT^{-1}, the multiplication of the coefficients with ψ before the NTT and with ψ^{-1} after the NTT^{-1}, and the bit reversal. The NTT and NTT^{-1} functions differ only by their second parameter, which are the powers of ω for the NTT and the powers of ω^{-1} for the NTT^{-1}. The bit reversal function reverses the bits of the polynomial with the help of a look up table before the NTT^{-1}. We will categorize the pointwise multiplication, which belongs to the $\mathsf{NTT}_{\text{multiplication}}$, not here as it will fit better into the category of polynomial arithmetic during our elaborations of the applied optimization techniques.

The second category contains building blocks that are used for **error reconciliation**. This category includes the generation of the help vector and the error reconciliation itself. Both functions operate on another representation of the polynomial. As described in Section 2.4.2.2 the key agreement is performed on the lattice $\hat{D}_4$. The two functions are intertwined due to their definition. The help vector generation takes a polynomial and noise as input and returns the help-vector $\mathbf{r}$. The reconciliation function computes a key from such a help-vector and an approximate polynomial. The formal definitions can also be found in Section 2.4.2.2.

The third category contains building blocks that perform **polynomial arithmetic**. This category includes the polynomial addition and the pointwise multiplication. The polynomial addition and pointwise multiplication do exactly what one would expect. The polynomial addition performs n additions of the coefficients and the pointwise multiplication performs n multiplications of the coefficients.

The fourth category contains a building block that makes use of the **ChaCha20** stream cipher. This category includes the error sampling from the centered binomial distribution Ψ_{16}. The sampling is implemented by a call to the ChaCha20 stream cipher. The parameters are a random seed, an 8-byte nonce, which gets varied, and a buffer object which gets filled with the coefficients.

The fifth category contains building blocks that are based on **Keccak**. This category includes the generation of the public parameter $\mathbf{a}$, which is realized by the extendable output function SHAKE-128, and the hashing of the 32-byte value ν, which is realized by the secure hash function SHA3-256.

3.3 Adapting NewHope for the Cortex-M0

Two aspects need to be discussed before the optimization per building block can be introduced. The first is to fit the NewHope key exchange onto the Cortex-M0 with its architectural constraints. The second is to establish communication with the Cortex-M0. Both concepts are straight forward; First the building blocks of the key exchange need to fit on the Cortex-M0 to be computable on the device. Second the communication is needed to derive the results of the exchange's computations and to allow the concept of exchange in general.

3.3.1 Architectural Constraints

The reference implementation of NewHope is implemented according to the structure which can be derived from the description of NewHope in Protocol 2. This also relates to naming of variables and the use of multiple instances instead of reuse of variables.

The first aspect we had to adapt was that the Cortex-M0 can only provide either side of the key exchange, the functionality of the client side **or** the functionality of the server side. We implemented both participants of the key exchange in files named accordingly `client.c` and `server.c`.

The second aspect we had to adapt, derived from the choices in the reference implementation, was to restructure the server side computations and the client side computations. The reference implementation makes use of five polynomials for the key generation ($\mathbf{a}, \mathbf{s}, \mathbf{b}, \mathbf{e}$, and $\mathbf{r}$), four polynomials for the final server computations ($\mathbf{v}, \mathbf{b}, \mathbf{s}$, and $\mathbf{t}$) and eight during client-side computations ($\mathbf{b}, \mathbf{a}, \mathbf{s}', \mathbf{e}', \mathbf{e}'', \mathbf{t}, \mathbf{u}$, and $\mathbf{v}$). Each polynomial is named accordingly to its usage in the top level description of Protocol 2, except $\mathbf{t}$, which is a temporary polynomial. Each of these polynomials is represented as an array of its coefficients. We know that $n = 1024$ and that each coefficient is smaller than $q = 12289$, thus has at most 14 bits. Data types need to be expressed in multiples of bytes on the Cortex-M0. This knowledge combined implies that the most straight forward way to store one polynomial is to have an array of length 1024 with a 16-bit (2-byte) data type, an unsigned 16-bit integer `uint16_t`. This in turn implies that each polynomial is represented by 2048 byte, thus 2KB. The final server computations and the client side computations additionally have a 2KB character array as one of their inputs. The architecture of the Cortex-M0, however, only provides 8KB of RAM. The initialization of the Cortex-M0 already requires memory and we also need some memory next to the representations of polynomials. This implies that we have a maximum of three polynomials, which can be used in parallel.

The change we performed is to adapt the data types used as parameters according to our needs. The reference implementation makes use of character arrays of 2048 byte as parameters to perform byte-wise communication later on. We changed the type of these parameters to represent a polynomial, internally defined as struct and named `poly`, the above mentioned 1024 element array of `uint16_t` types. By applying this change we generate an uninitialized polynomial already in the function parameters. We could afterwards make use of this polynomial and restructure the code of the client side and the code of the server side such that they only make use of three polynomials. This could be realized straight forward for the server side. For the computations on the client side, we were required to restructure the call of subfunctions and to perform a swap of values at the end which gives a slight overhead. Despite that, these implementation changes enabled us to call the API functions provided by the reference implementation with our adaptations, such that both the client side and the server side of the NewHope key exchange can be computed on the Cortex-M0.

A drawback of the change of the parameter type is that the decoding and encoding operations defined in the reference implementation do not work for the new data type. We were therefore obliged to adapt those functions accordingly. Next to the encoding and decoding of additional information, the seed for $\mathbf{a}$ in the case of message A and the help vector $\mathbf{r}$ for message B, the functions make use of two subfunctions which calculate a polynomial from a byte array `poly_frombytes` and vice versa `poly_tobytes`. Inside these functions the bits are shifted to the low end if we assume a little-endian representation. We therefore replaced those two functions with a `densify` and an `amplify` function. Both keep the bit order of the reference implementations `poly_frombytes` and `poly_tobytes`, however, the `densify` and `amplify` functions operate on the same polynomial instead of transferring the information into another data type, the character array.

The third aspect we had to adapt was to replace the call to the ChaCha20 stream cipher. In the reference implementation, the generation of noise polynomials takes a 32-bit seed and extends it by calling the ChaCha20 stream cipher with those seeds into a buffer of $4n = 4096$ bytes. The generation of noise polynomials is called on either side, the client or the server, from a context in which three polynomials already exist in memory. This implies that there is no space left for a 4KB buffer to be filled by the ChaCha20 stream cipher. We solved this issue by calling the ChaCha20 function four times in a row with a buffer of 1KB, which fits into the memory at this point of execution. However, since we call it with the same seed, some source of entropy is needed to provide different outputs. The reference implementation makes use of one single byte to produce different noise polynomials from one noise seed by changing the first byte of the 8-byte nonce of the ChaCha20 stream cipher. We decided to incorporate that concept and perform an additional change in the second byte of the nonce. As value, we simply use the counter of our loop performing the consecutive calls to the ChaCha20 stream cipher.

The fourth aspect we had to adapt was to replace the generation of the

seeds for the generation of the public parameter **a** as well as the generation of the noise polynomials. As pointed out above, the Cortex-M0 does not provide an RNG. Further, no default approach to generate randomness exists on the Cortex-M0. The reference implementation simply queries `/dev/random`, which of course is not available on the embedded Cortex-M0. We chose to allow a context specific implementation by providing an easy-to-replace function in our C code `randombytes` inside the `randombytes.c` file. If the key exchanged should be used in practice it is *crucial* to replace our deterministic `randombytes` function by a true RNG.

3.3.2 Communication

We make use of two different concepts for the communication. We perform communication via the universal asynchronous receiver/transmitter (UART) and via direct memory access (DMA). We started by using UART communication with the libraries needed to establish the communication from [37]. In the final version, we make only use of these libraries and communication technique for speed and memory calculations. The example code for communication we provide for the host, is realized via DMA on the Cortex-M0 and makes use of the open cm3 library from [45]. This enables the Cortex-M0 to perform computations while data is transferred. For the host side, we provide two Python scripts. One to respond to the Cortex-M0 initialized to perform as server side of the key exchange and a second to respond to the Cortex-M0 initialized to perform as client side of the key exchange. In Table 3.3 can be seen which messages are expected, for either the client or the server side of the key exchange. All values which are sent and received are in bytes. A guard parameter is used, which has the decimal value 10. It is used to separate between the independent messages which need to be transferred.

TABLE 3.3: Communication pattern for client and server side of the NEWHOPE key exchange on the Cortex-M0.

Client-side	**Size**	**Server-side**	**Size**
guard	1	guard	1
receive message A	1824	send message A	1824
guard	1	guard	1
send message B	2048	receive message B	2048
guard	1	guard	1
(response to key check	5	response to key check	5)

The last row of Table 3.3 is put into brackets, because the key check performed is only valid for the deterministic implementation we provide as proof of concept with this work. In any real-world instantiation of our Cortex-M0 code, this key check would be canceled and instead the usage of the just agreed ephemeral key could be started.

3.4 Optimization Details per Building Block

In this section, we give a detailed explanation of the optimization techniques we applied. We structure it according to the categories we introduced in Section 3.2. A general optimization technique we applied is to make use of the benefits we could achieve from the word size of the processor. Each of the coefficients used to represent a polynomial for the NewHope key exchange is 16-bit, thus one half-word. We could decrease the amount of memory operations needed by loading full-words and making use of the multiple load and store instructions. We do, however, have to pay some overhead for loading full-word representations. If we want to perform other than logical operations, we need to split the 32-bit value in one register into two 16-bit values in two registers. In Listing 1 the example of two full-word loads can be seen. In Listing 1a the load is realized by using the multiple load `LDM` instruction, in Listing 1b the load is realized by using four load register `LDR` instructions. The general observation we can take is that it is more efficient to reduce the amount of memory operations to its minimum.

Listing 1 Comparison of full-word and half-word loads for 64-bit.

(a) Usage of full-word loads and required splitting (7 cycles).

```
load_fullwords,ri:

LDM ri, [rx0,rx1]  (3)
UXTH rfree0, rx0   (1)
LSR rx0, #16       (1)
UXTH rfree1, rx1   (1)
LSR rx1, #16       (1)
```

(b) Usage of half-word load register instructions (8 cycles).

```
load_halfwords,ri:

LDRH rx0, {ri}       (2)
LDRH rfree0, {ri,#2} (2)
LDRH rx1, {ri,#4}    (2)
LDRH rfree1, {ri,#6} (2)
```

Next to the savings achieved by loading full-words in a consecutive manner we could also benefit from loading single full-words. If we assume a loop that iterates over an array of 16-bit values, we can decrease the amount of memory operations. We achieve this by loading two 16-bit values at once and perform the computations of the body of the loop on both values.

3.4.1 NTT

The NTT is the most frequently called building block of the NewHope key exchange, as can be seen in Table 3.2. Furthermore, it does have the largest cycle count of all building blocks from the key exchange. In the C reference implementation provided by the authors of NewHope [1] the NTT is implemented as can be seen in Listing 3. It consists effectively of three nested for-loops. The first is counting the levels from 0 to 10. The second is counting the distance and the third the step size. Inside of these three nested loops, a Gentleman-Sande butterfly operation [29], summarized in Listing 2 and explained in Section 2.4.2.3, is performed. In the case of the C reference implementation the levels are split into even and odd levels, the lazy Gentleman-Sande butterfly operation, used in the odd levels, is summarized in Listing 4 and explained in Section 2.4.2.4.

We further benefit from the efficiency considerations described in Section 2.4.2.4, which the authors of NewHope took into account. The NTTs

Listing 2 =Gentleman-Sande butterfly - all variables are `uint16_t`.

```
W = omega[(1024 + j)/(2*distance)];
temp = a[j];
a[j] = bar((temp + a[j + distance]));
a[j + distance] = mon((W * ((uint32_t)temp + 3*PARAM_Q - a[j + distance])));
```

Listing 3 =Structure of the NTT in the C reference implementation.

```
for(i=0;i<10;i+=2)
  // Even level
  distance = (1<<i);
  for(start = 0; start < distance;start++)
    for(j=start;j<1023;j+=2*distance)
      do lazy butterfly
  // Odd level
  distance <<= 1;
  for(start = 0; start < distance;start++)
    for(j=start;j<1023;j+=2*distance)
     do butterfly
```

are faster due to not computing the bit reversal. The application of the Barrett reduction [3] and the application of the Montgomery reduction in step three and four of the butterfly operations [57] instead of the usage of a straight forward modulo operator saves a lot of cycles. On the Cortex-M0 a reduction by the modulo operator '%' specified by the C language would take 210 cycles. Our assembly implementation of the Barrett reduction can be seen in Listing 5d and takes 5 cycles. Our assembly implementation of the Montgomery reduction can be seen in Listing 5c and takes only 6 cycles. The efficiency consideration to perform a lazy butterfly operation every second level saves even more cycles by omitting half of the the otherwise required Barrett reductions. We make use of this in our assembly implementation, but decided to present the C implementations due to code size and clarity (the lazy butterfly operation in Listing 4 and the butterfly operation in Listing 2). With these optimization techniques in place we had a solid base to build our architecture optimized assembly code on.

The first change we made was the amount of butterfly and lazy butterfly operations we perform inside the nested loops. We perform two butterfly operations after one another. This can be translated to the C code in Listing 3 that the 'start' loop-counter would need to be increased by 2 instead of 1. This yields us the benefit of the 32-bit word size of the target architecture in combination with the 16-bit size of the coefficients of the polynomial, the elements of array a. We can load two values $a[j]$ and $a[j+1]$ simultaneously for the cost of one load, by using the `LDR` instruction instead of using the `LDRH` instruction twice. The same holds for the store instruction `STR` which can be used instead of two `STRH` instructions. Note that it is not efficient in the context of the `NTT` to perform multiple loads/stores (`LDM`, `STM`) due to the varying step size.

The second change we implemented was to unroll all levels. For the C implementation this is equivalent to removing the outermost loop and appending the 10 levels after one another. This decreases the cycle count due to omitting branch instructions of the outermost for-loop in the trade-off for code size.

Listing 4 =Lazy Gentleman-Sande butterfly - all variables are `uint16_t`.

```
W = omega[(1024 + j)/(2*distance)];
temp = a[j];
a[j] = (temp + a[j + distance]); // Omit reduction (be lazy)
a[j + distance] = mon((W * ((uint32_t)temp + 3*PARAM_Q - a[j + distance])));
```

Listing 5 Optimized reduction routines.

(c) Montgomery reduction (for $rQ = 12289$ and $rlog = 2^{18}$).

```
mon, rIn:
 SUB rTmp, rQ,#2
 MUL rTmp, rIn
 AND rTmp, rlog
 MUL rTmp, rQ
 ADD rIn, rTmp
 LSR rIn, #18
```

(d) Short Barrett reduction (for $rQ = 12289$).

```
bar, rIn:
 LSL rTmp, rIn, #2
 ADD rTmp, rIn
 LSR rTmp, #16
 MUL rTmp, rQ
 SUB rIn, rTmp
```

The third change we performed was to merge level 0 and level 1. The idea behind this is to reduce the amount of load and store instructions needed. Level 0 applies the butterfly operation on two consecutive elements. Level 1 applies the lazy butterfly operation on two elements with a distance of two. Our first change implies that two elements are loaded simultaneously. Therefore, $a[0]$,$a[1]$,$a[2]$,$a[3]$ are loaded into the registers in the first iteration. This enabled us to perform level 0 twice. First on $a[0]$ and $a[1]$ and second on $a[2]$ and $a[3]$. After these computations we perform level 1 twice on the same elements. First on $a[0]$ and $a[2]$ and second on $a[1]$ and $a[3]$. After these computations we store the computed values, which consist of the computations of the first iteration of level 0 and the computations of the first iteration of level 1. This concept produces a slight overhead, because we need to recalculate the correct ω for both levels. However, it saves double loading and storing of the values, which results in an overall cycle count optimized version compared to non merged level 0 and level 1. We also tried to apply this concept to the other levels. However, we are confronted with architectural constraints. As mentioned earlier, we need to recalculate the ωs which generates some overhead, because we do not have enough registers to hold all values in parallel. In the case of higher levels the computations are not performed on consecutive elements, which lets the overhead get out of proportion. For example, the first iteration of level 3 would need to load $a[0]$, $a[1]$ and $a[8]$, $a[9]$. Level 4, however, would additionally need $a[16]$, $a[17]$. The only thing achieved by this structure is that one load instruction can be omitted ($a[0]$, $a[1]$ can be used for both levels). However, this does not save enough cycles compared to the overhead created by the required recalculations of omega and calculating the right offsets with the limited register capabilities of the target architecture.

The fourth change we performed was to minimize register reordering. As pointed out earlier, the target architecture has only 8 full-purpose registers and 5 usable high registers. For the NTT we make use of the high registers `r8` to `r12` and additionally make use of the link register `r14`, which can be done since we do not have internal function calls and our assembly function takes care of storing and restoring its value. We went through

our NTT code and optimized it such that constants and loop-counter are placed in high registers where possible. We still need to shift some register contents, however, we keep that to the minimum and it enables us to make an optimal use of the capabilities of the limited target architecture.

3.4.1.1 Multiplying the Coefficients

Additional to the computations performed in the core function of the NTT we also need to perform a multiplication of the coefficient with the precomputed ψs before each call to the NTT and a multiplication of the coefficients with the precomputed ψ^{-1}s after each call to the NTT^{-1}. We implemented the multiplication of the coefficients in assembly to benefit from the Cortex-M0's 32-bit word size. Extending this architectural benefit we make use of the fact that the multiplication of the coefficients with the precomputed coefficients is a simple operation and does not need many registers. Therefore, we were able to load 4 coefficients at once and also store them. With this we decreased the amount of loads and stores required.

3.4.1.2 Bit Reversal

In the case of the bit reversal we do not provide an assembly version. A more detailed explanation of how this could be achieved in an environment with a larger flash, can be found in Section 4.1. The major problem why we could not apply techniques similar to the optimization techniques used for the combination of two polynomial is that the elements read from the lookup table are not consecutive. This implies that we cannot benefit from the architecture by loading full-word sized entries. What we could do was to take the natural boundaries into account and do not loop over the last 33 elements, which do not get changed.

3.4.2 Error Reconciliation

The error reconciliation mechanism used in the NewHope key exchange makes use of two functions, the generation of the help vector for reconciliation and the reconciliation function itself. We will explain in the following two accordingly named sections the implementation details of our optimized assembly versions. The authors of NewHope claim in the first publication of the key exchange [1] that the error reconciliation method used "can be implemented in constant time using only integer arithmetic - which is important on constrained devices without a floating-point unit". With the Cortex-M0 at hand we face exactly the constraint of not having a floating-point unit and could therefore benefit from this design decision.

3.4.2.1 Generation of the Help Vector

From an implementation point of view the generation of the help vector can be compartmentalized into the following components.

First, the 32-byte seed is provided as parameter to the ChaCha20 stream cipher, together with the 8-byte nonce described in 3.3.1 to sample an array of 256 random bits b. For further detail on the ChaCha20 building block please refer to the corresponding Section 3.4.4, for a reminder of the usage of the random bit b please refer to Section 2.4.2.2.

Second, the help reconciliation function loops over the random-bit array consisting of 32 character entries. The reference implementation therefore loops 256 times, because every bit of each character needs to be processed. This approach has some redundancy, which we removed in our assembly implementation.

Third, the remaining body of the loop is computing the $CVP_{\hat{D}_4}$ algorithm described in Algorithm 1. In the reference implementation this is realized by calling a function `f` which performs the first two steps and parts of the third step of $CVP_{\hat{D}_4}$. `k` is then set to either 0 or 1 and the last two steps of the $CVP_{\hat{D}_4}$ are computed.

As hinted at in our description of the second component, the reference implementation of the loop does contain redundant logic. We approached this by restructuring the loop. We do not loop over 256 elements anymore, but we perform the same call eight consecutive times for the 8 different 32-bit entries of the array of random bits. We chose to switch to loading eight times 32-bit, contrary to the C implementation which has a thirty-two element array of 8-bit. By this, we achieved the benefit from the decrease of memory operations needed due to the architecture's word size. We continued by performing four times the same computations per byte of the loaded random bits. Each of these computations in turn gets split into four internal computations. Each of these internal computations, called `oneinternal` in our assembly file takes the two least significant bits of the 32-bit random bits loaded from memory and shifts the other random bits two to the right. In Listing 6, the macro can be seen. Overall it is called 128 times and reduces therefore the amount of calls, compared to the loop. It stores the relevant registers and calls a function which processes the internal computations of the help reconciliation function, those described in the third component.

Listing 6 Structure of the `oneinternal` macro.

```
oneinternal rrandom32
    MOV rhelp,#3
    AND rhelp,rrandom32
    LSR rrandom32,#2
    PUSH {rarraypointer0,rarraypointer1,rarraypointer2,rrandom}
    MOV rarraypointer1,rhelp
    BL asm_helprec_internal
    POP {rarraypointer0,rarraypointer1,rarraypointer2,rrandom}
    ADD rarraypointer0,#4
    ADD rarraypointer2,#4
```

The assembly function `asm_helprec_internal` performs the computations of the $CVP_{\hat{D}_4}$ algorithm on two entries opposed to the definition and the reference implementation, where only one entry is processed per iteration. This is the reason why we extract two of the random bits for each call. Inside the `asm_helprec_internal` function this yields the benefit of the 32-bit architecture. It enables us to load and store two 16-bit coefficients with one memory operation. Additionally, it decreases the amount of calls to `asm_helprec_internal` we need to perform. This saves overhead for the function call. We would rather omit calling the function completely and replace it by a macro, which is not possible due to the architectural constraints, flash size in this case.

3.4.2.2 Reconciliation

From an implementation point of view the generation of the help vector can be compartmentalized into the following components.

First, the 32-byte key is initialized to zero.

Second, the reconciliation function defines a loop which iterates 256 times. During each iteration it computes the input to the `Decode` function, which we described in Section 2.4.2.2. To achieve this, the values from the two input polynomials are loaded and multiplied by multiples of q to satisfy the parameters of $Decode(\frac{1}{q}x - \frac{1}{2^r}B\mathbf{r})$ by using integer arithmetic.

Third, the remaining body of the loop sets the 256 key bits of the `Decode` function's output per iteration of the loop.

Again the reference implementation relies on redundant logic. We took a similar approach as we already did for the help reconciliation. We restructured the loop such that we do not loop over 256 elements anymore, but we perform the same call eight consecutive times for the 8 different 32-bit entries of the `key` array. We again chose to switch to loading eight times 32-bit, contrary to the C implementation, to benefit from the decrease of memory operations needed due to the architectures word size. We continued by performing four calls to the `asm_8bits_key` function per byte of the `key`, which still need to be determined. Each of these calls produces 8-bit of the `key` which is shifted x bits to the left for $x \in \{0, 8, 16, 24\}$ and combined afterwards by a logical `or`. With this we only perform one store operation per full-word of `key` bits to reduce the amount of memory operations to the minimum. Inside the `asm_8bits_key` function we, therefore, perform eight times the computations of one loop iteration of the reference implementation. Performing the computations an even number of times again allows us to load full-word entries from memory, which in turn decrease the amount of memory operations needed. Inside each of these instances the body of the loop in the reference implementation gets executed. We load the coefficients, multiply them by multiples of q and input them to the `Decode` function. The `Decode` function in turn calls a subfunction `g` on all four input elements, together forming a vector in $\hat{D}_4$. The distance is computed and the most significant bit extracted and returned as entry

of the key. We need to perform a call to the `asm_8bits_key` function, because otherwise we would again face the flash size constraint of the target architecture.

3.4.3 Polynomial Arithmetic

The polynomial arithmetic is used to combine two polynomials. The two representatives of combining two polynomials are polynomial addition and pointwise multiplication. We will discuss the optimization techniques applied in the following subsection.

3.4.3.1 Polynomial Addition and Pointwise Multiplication

Polynomial addition as well as pointwise multiplication are straight-forward concepts. A simple loop iterates over the 1024 entries of the array representing the coefficients of two polynomials and either performing an addition or a multiplication. We decided to rewrite both functions such that they make optimal usage of the target architecture, by making use of the 32-bit word size. We load and store two consecutive coefficients of the polynomial and apply the calculations needed on each half-word. By doing so, we only call half of the iterations of the main loop. In the case of the addition, we even have enough registers to load and store 4 coefficients with one `LDM` or one `STM` instruction. In the case of the pointwise multiplication, we were only able to load and store 2 coefficients simultaneously, because we need to reduce the product after multiplication.

3.4.4 ChaCha20

The ChaCha20 stream cipher [5] is a standalone stream cipher presented by Bernstein as a variant of the Salsa stream cipher in 2008. It is used for noise generation in the NewHope key exchange. All noise polynomials get generated from a seed and a nonce. The fair coin behavior of the help reconciliation algorithm is also based on a call to the ChaCha20 stream cipher.

We based our architecture-specific implementation on the ChaCha20 implementation by Neikes and Samwel [61], which was specifically designed for the Cortex-M0. They optimized the core functionality of the ChaCha20 stream cipher in assembly, which fits well into our approach. However, we still optimized two minor aspect.

The first aspect we optimized is based on the circumstance that the ChaCha20 core functionality operates on variables stored in little-endian. The reference implementations from the authors of NewHope [1] and from Neikes and Samwel [61] have two subfunctions which provide the ability to store and load into the little-endian representation. As pointed out in Section 3.1 the default endianess of the Cortex-M0 is already little-endian. This architectural benefit allowed us to omit these two functions.

The second aspect we optimized was to replace the two loops in the ChaCha20 body with a call to our substituting assembly function. It is no huge cycle-count saving we derived, but the assembly version is much more elegant as can be seen in Listing 7e compared to Listing 7f.

Listing 7 Reference and assembly implementation of two loops inside the ChaCha20 stream cipher.

(e) Assembly implementation of the two loops.

```
LDM rn!,{rtemp0,rtemp1}
STM rin!,{rtemp0,temp1}
EOR rtemp0,rtemp0
EOR rtemp!,rtemp1
STM rin!,{rtemp0,rtemp1}
```

(f) Reference implementation of the two loops, with 'in' and 'n' being unsigned characters.

```
for (i = 0;i < 8;++i) in[i] = n[i];
for (i = 8;i < 16;++i) in[i] = 0;
```

3.4.5 Keccak

The assembly implementations of Keccak are taken from the Noekeon repository [14]. We decided to do so because the development team behind the Keccak family provides a version of their primitives for our target architecture. At least that is what was claimed. After a mail correspondence with the developer, an updated code version was provided which actually compiled on the Cortex-M0. The next problem we faced was that we do not have full block inputs for the computations of SHAKE-128 for the generation of $\mathbf{a}$ and SHA3-256 for hashing the ephemeral key ν. This prohibits the usage of the predefined Squeeze and Absorb functions provided for the ARMv6M architecture. However, the header file inside their code packages subdirectory `SnP/SnP.h` provides detailed information on the inner workings. This enabled us to call their assembly subfunctions to fulfill our needs. For absorbing we just call `KeccakF1600_StateXORBytes` with the correct parameters. For squeezing we need to make two function calls, a first one to `KeccakF1600_StatePermute` and a second to `KeccakF1600_StateExtractBytes`. This suffices for SHA3-256 since we do not absorb or squeeze more than one full block. However, the situation for SHAKE-128 is different. The way NewHope makes use of this function is either to squeeze one or four blocks. Therefore, we distinguished between those cases and provide two fixed implementations. One function calling the pair of assembly functions needed for squeezing once and a second function calling the pair of assembly functions needed for squeezing four times in a row. By doing so we were able to replace the computational costly functions from the reference implementation specified in the `fips.c` file by assembly functions provided by Noekeon [14].

3.5 Results

We compiled our implementation with `arm-none-eabi-gcc` version 5.2.0 and `-Ofast` as compiler flag. By applying our assembly-optimized implementation and using the fast and efficient implementations of ChaCha20

and Keccak described in the previous section we gain significant speed-ups, on the Cortex-M0, compared to the adapted C version of NewHope, described in Section 3.3.1. These speed comparisons are performed by measuring cycles, which we implemented on the Cortex-M0 (according to [37]) by making use of the system timer SysTick. It is set to 12 000 000 in the beginning and counted down while the function is called, of which the cycle count needs to be determined. The value of the system timer after the execution of the function in question gets subtracted from the value of the system timer before the call to the function. Additionally, the value gets decreased by 2, because calling no function yields a cycle count of 2.

In Table 3.4 we present the cycle counts per building block (summarized and explained in Section 3.2) for the adapted C version which is able to run on the Cortex-M0 and for our assembly optimized implementation. In a third column we present the percentage we saved by our implementation compared to the adapted C version. We further present the overall cycle counts needed for the client side computations, the server side computations, and the key exchange in total together with their percentages, representing the overall speed-up we gained. Please note that the major cycle count achievement of the generation of **a** and of the noise sampling is a based on Keccak and ChaCha20 and is thus achieved by reusing and optimizing the work of the before mentioned ARMv6-M implementation of Keccak [14] and ChaCha20 [61].

Table 3.4: Cycle counts per operation, together with overall performance, of the NewHope key exchange. Additionally, the percentage of improvement is presented.

Operation	**C on M0**	**Our implementation**	**Improvement**
Generating **a**;	527 293	328 789	37.65 %
Sampling noise ;	234 024	208 693	10.82 %
Multiplying coefficients ;	34 842	15 665	55.04 %
NTT;	327 004	148 517	54.58%
Bit reversal;	19 385	18 888	2.56%
Polynomial addition;	33 822	13 860	59.02%
Pointwise multiplication;	58 406	25 652	56.08%
HelpRec;	79 663	68 170	14.43%
Rec;	69 137	46 945	32.10%
SHA3-256.	29 032	23 999	17.34%
Client Side	2 723 353	1 760 837	35.34%
Server Side	2 433 238	1 467 769	39.70 %
NewHope	5 156 591	3 228 606	37.39%

Additional to this internal comparison, we are thus comparing our own results with the adapted C reference implementation, it would be interesting to perform an external comparison, thus comparing our results with results achieved by other researchers in this field. In the literature various implementations of lattice-based cryptography on embedded microcontrollers

can be found (e.g. [70] targeting the AVR architecture, or [69] targeting FPGAs). To perform a direct and fair comparison is nearly impossible due to many, often unsolvable constraints. Different architectures are used, the implemented schemes vary and the security levels differ (no lattice-based key exchange in the literature is as conservative in the choice of parameters, and thus secure, as NewHope).

When this thesis was written, we could identify two papers that optimize lattice-based cryptography on ARM Cortex-M based microcontrollers. De Clercq, Roy, Vercauteren, and Verbauwhede optimized RLWE-based encryption on the Cortex-M4F microcontroller in [11] and Oder, Pöppelmann, and Güneysu optimized the Bliss signature scheme in [63] also on the Cortex-M4F microcontroller. The Bliss signature scheme is presented in [21] by Ducas, Durmus, Lepoint, and Lyubashevsky.

Regarding the architecture, the Cortex-M0 (elaborated in Section 3.1) is inferior to the Cortex-M4F. The 'F' specifies the fact that a floating point unit is present. The Cortex-M4F microcontroller forms the upper end of the Cortex-M family. The Cortex-M4F is based on the ARMv7-M architecture and makes use of the 32-bit Thumb 2 instruction set, which allows it to make full usage of the 13 general purpose registers and perform operations like shifting a register in the same cycle as adding by extending the `add` instruction. Additionally, store instructions on the Cortex-M4F require only 1 cycle because the address generation is performed in the initial cycle and the actual storing of data is performed while the next instruction is executed.

Regarding the varying schemes implemented, the widely accepted and often applied technique to compare different schemes is to compare the performance of subroutines. In the case of RLWE-based cryptography the most relevant subroutines to compare are noise sampling and the $\mathsf{NTT}_{\text{multiplication}}$, which are the two most costly operations. A comparison of the error sampling with our results can be seen in Table 3.5.

Table 3.5: Performance comparison of the error sampling.

	Noise sampling[a]
Cortex-M0 **(ours)**	270
Cortex-M4F [11]	28.5
Cortex-M4F [63]	2 066

[a] Cycle counts for sampling one coefficient.

From Table 3.5 we can derive that our error sampling algorithm based on ChaCha20 is 86.93% faster than the speed-optimized Gaussian sampling based on the Ziggurat algorithm implemented by [63]. However, it is 89.63% slower compared to the Gaussian sampling based on the Knuth-Yao algorithm implemented by [11]. We could only realize error sampling without a required floating point unit on the Cortex-M0, which is the first obvious reason for being slower than the Knuth-Yao algorithm implemented by [11]. A second reason is that the sampling algorithm used by NewHope, unlike the Knuth-Yao sampler, runs in constant time and is

thus inherently protected against timing attacks. Therefore, the decreased performance on the Cortex-M0 is a price to pay for compatibility with significantly increased timing-attack-protected sampling performance on large processors with caches. More detail on this can be found in [1].

The implementations presented in the papers [11, 63] perform the NTT on 512-coefficient polynomials with the same modulus $q = 12289$ that we used. The usage of the parameter $n = 512$, compared to our case where $n = 1024$, implies that we need to scale the cycle counts. As explained in the Section 2.4.2.3, the NTT computations consist essentially of $\log_2 n$ levels of Gentleman-Sande butterfly operations. Each level performs its computations on $\frac{n}{2}$ elements. The usage of twice as many coefficients implies that the elements per level double and that an additional level is required during the computation of the NTT. For comparison this has the consequence that we need to scale the results of Clercq et al. [11] and Oder et al. [63] by a factor of $\frac{20}{9} \approx 2.22$ to match the same dimensions we used. A comparison of the $\mathsf{NTT}_{\text{multiplication}}$ is provided in Table 3.6.

TABLE 3.6: Performance comparison of the $\mathsf{NTT}_{\text{multiplication}}$.

	$\mathsf{NTT}_{\text{multiplication}}$
Cortex-M0 **(ours)**	537 086
Cortex-M4F [11]	528 451[b]
Cortex-M4F [63]	1 130 276[b]

[b] Number scaled from dimension 512 to dimension 1024 by multiplication with $\frac{20}{9}$.

From Table 3.6 we can derive that the cycle counts we achieve on the Cortex-M0 to compute the $\mathsf{NTT}_{\text{multiplication}}$ are 53% faster than the implementation by [63] and 2% slower than the implementation by [11]. The reason for being 2% slower can be found in architectural constraints regarding the instruction set and parallelization. The cycle deficit lies in the bit reversal, the polynomial multiplication and the multiplication with coefficients. As we followed a strict compartmentalization, the memory addressing of the same elements (for the NTT, both multiplications and bit reversal, if the NTT^{-1} is performed) proof to have a higher impact on the performance of the Cortex-M0 than it has on the more advance Cortex-M4F (e.g. storing takes only 1 cycle).

If we compare the cycle counts of the NTT directly, however, we receive an overall faster result as can be seen in Table 3.7.

TABLE 3.7: Performance comparison of the NTT.

	NTT
Cortex-M0 **(ours)**	148 517
Cortex-M4F [11]	157 977[b]
Cortex-M4F [63]	272 486[b]

[b] Number scaled from dimension 512 to dimension 1024 by multiplication with $\frac{20}{9}$.

With respect to the NTT the cycle counts we achieve on the Cortex-M0 are 6% faster than the results on the Cortex-M4F from [11] and 45% faster than

the results on the Cortex-M4F from [63]. This shows that the optimization measures applied by us enable inferior hardware to perform the best results in calculating an NTT on ARM Cortex-M processors.

Conclusion

With this thesis we have provided a fast and efficient implementation of the ephemeral key exchange NewHope published by [1]. We have started with a general overview of the terminology of post-quantum cryptography and narrowed down our focus on lattice-based cryptography and the corresponding problems. From there we derived the passively secure KEM proposed by [65] and discussed its instantiations, from which the second is NewHope. Subsequently we elaborated NewHope in detail and presented our ARMv6-M optimized implementation on the Cortex-M0.

We achieved an overall performance gain for the key exchange of 37.39% compared to the C reference implementation provided by the authors of NewHope [1], which we adapted to make it computable on the target device. We achieved a performance gain of 35.34% for the client side computations and a performance gain of 39.70% for the server side computations. As pointed out in Section 3.2, the main computational effort from the NewHope key exchange lies in the computation of the NTT. We focused especially on this function and were able to achieve a 54.58% faster implementation compared to the C reference implementation. Compared to the literature we achieve an at least 6% faster implementation even on slower Cortex-M processors as pointed out in Section 3.5. With those cycle counts we provide a solid base for a potential application of the NewHope key exchange on small embedded ARM processors using the ARMv6-M architecture.

4.1 Reflection

The implementation of NewHope we performed on the Cortex-M0 was focused on efficiency. However, we sometimes chose to not only look for cycle counts but to take the best cycle count we can get without using to much space in the ROM. The most suitable example is the case of the bit reversal, which could be faster by unrolling it. However, the required space needed in the ROM would be out of proportion. A similar situation can be sketched for the error reconciliation. Instead of calling our assembly function `asm_8bits_key` it could be realized by a macro on devices with larger ROMs and in situations where pure cycle count measures are the determining factor.

We also chose to not stick to the assembly implementation of the decoding and the encoding of the polynomials to be send from server to client and vice versa. With the current settings chosen by the authors of NewHope it only saves 700 cycles compared to the C version. This was different for

previous encoding schemes with different error distributions published earlier by the authors of NewHope (e.g. in the first release of the NewHope paper the error distribution Ψ_8 was chosen and the decoding and encoding were realized differently, which granted us cycle counts which varied by several thousands comparing the C reference implementation and our former assembly version).

In general we chose for lower cycle counts, even if it means to discard previously written code. We had, for example, an assembly implementation of the ChaCha20 stream cipher. We unrolled the last iteration of the loop and performed the addition and storing of the results immediately to remove 48 memory operations. However, when we compared it to the implementation by Neikes and Samwel [61] we chose for efficiency instead of sticking to our code.

4.2 Future Work

During the implementation for this thesis and the process of writing it we encountered several aspects which we could not find time for or which extend the scope of our research. We elaborate the aspects we encountered, which could be realized by future research, in the following:

The first aspect we already mentioned is the lack of an RNG on the Cortex-M0. Future research could look into ways of retrieving randomness on the Cortex-M0 itself (e.g by using memory boot up patterns as was done for the Cortex-M3 by [82]) or compare external sources of randomness to find the most appropriate.

The second aspect is the memory-speed trade-off we discussed earlier in another context. In the case of a future work suggestion, one could remove all or parts of the precomputed constants to compute the NTT and the NTT^{-1}, thus the ωs and ψs. In one implementation we realized during the development of our final code we were able to omit the 1KB of bit reversed ψs in the Montgomery domain `psis_bitrev_montgomery` by increasing the cycle count only by 700 cycles. Similar approaches could be realized by reordering the output of the NTT^{-1} to omit the inverted ωs and computing parts of the inverted ψs or even all constants on the fly.

The third aspect is that closely before we finalized this thesis Longa and Naehrig from the research and development team of Microsoft published a paper [47] which improves the NTT of the NewHope key exchange. They have provided a new reduction method, which does not require the conversion to the Montgomery domain and omits not just half but all of the Barrett reductions. This is achieved by increasing the size of the datatype for the coefficients and making it signed. Each coefficient would be of type `int32_t`. This of course rules out this optimization technique on memory-constrained devices such as the Cortex-M0. The problem is that we could fit only 1 polynomial with 1024 coefficients each being a full-word in size into the RAM. Other versions of the Cortex-M0 with more RAM could, however, benefit from the optimization technique. We were not able to compare the NTT directly, since it is realized by Cooley-Tukey butterfly

operations instead of Gentleman-Sande butterfly operations as used in our implementation. However, the NTT^{-1} algorithm proposed by [47] is comparable to our implementation as it also uses Gentleman-Sande butterfly operations. Due to using their new reduction approach the bit reversal required before the NTT^{-1} was also adapted. We performed a 'quick and dirty' implementation of Algorithm 4 from [47]. We implemented the bit reversal in assembly, replaced the Montgomery reduction by the K-RED reduction in all levels, removed the Barrett reduction from all levels and perform one single K-RED-2x reduction. With those changes in place we predict that the modifications proposed by [47] should also make a difference on architectures with enough RAM in the magnitude of 10k cycles for the NTT^{-1} including bit reversal.

The second and third aspect show that faster implementations are possible, however, they will come for the price of larger memory requirements, either ROM or RAM. The implementation we provide therefore stays the fastest implementation of a lattice-based post-quantum key exchange on constrained hardware due to its low memory requirement.

Appendix A

Executing NewHope on the Cortex-M0

Within this appendix we provide a guideline on how to build and run our fast NewHope implementation on the STM32F0Discovery development board. As we started our journey with the elaborations on [36], we will follow the presented build instructions loosely. Most operating system should be able to cross compile our code. In the following guideline we assume a Debian based operating system. The commands which install system wide packages must be ran as privileged user (root). On the hardware side a STM32F0Discovery development board, the corresponding USB cable, a USB-TTL converter and the corresponding cables are required.

Installing the basic tools. First we need to install the low level virtual machine and GNU compiler collection for ARM.

```
apt-get install llvm gcc-arm-none-eabi
```

Developing for 32-bit. Second we need to install the 32-bit development libraries for AMD64 (if we cross compile from this architecture).

```
apt-get install libc6-dev-i386
```

STMicroelectronics ST-link Tools Third we need to install the ST-link tools for the Development board.

Installing Dependencies. We need to install the USB programming library development files.

```
apt-get install libusb-1.0-0-dev
```

Cloning, building and installing the ST-link tools. We need to clone the ST-link tools from its GIT repository, build and install them afterwards.

```
git clone https://github.com/texane/stlink.git
cd stlink
./autogen.sh
./configure
make && make install
```

Connecting the STM32F0Discovery board. The converter gets connected as described in Section 3.1. The ground and 3.3 Volt power cables are connected to the corresponding GPIO pins, RXD is connected to PA2 and TXD is connected to PA3. The 5 Volt cable stays unconnected. The final setup can be seen on Figure A.1.

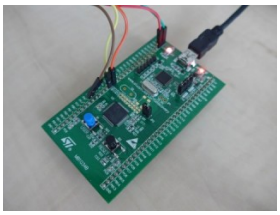

Figure A.1: Connections of the USB-TTL adapter to the development board. Taken from [36]

Clone and build NewHope for the Cortex-M0. We need to clone the source code of our NewHope implementation and build it.

```
git clone https://bitbucket.org/phibyte/new-hope-arm.git
cd new-hope-arm
make
```

Running the benchmarks. We need to call one of the following scripts to load either the memory or the speed test onto the Cortex-M0.

```
./[mem,speed].sh
```

To get the ROM size required by our implementation we need to query the archive size.

```
./arm-none-eabi-size -t libnewhope.a
```

Running NewHope. We need to execute the Python script, to interact with either the client or the server side, on the host computer and load the code via a corresponding script onto the Cortex-M0.

```
python Host\ Side/test_[client,server].py
./[client,server].sh
```

After performing these steps, the Python script will execute, communicate with the Cortex-M0 and print a positive result. If any communication issues occur, the 'reset key' on the development board needs to be pressed. It will reset the client (or server) side loaded onto the Cortex-M0 and the Python script can begin to interact from the start.

Bibliography

[1] E. Alkim, L. Ducas, T. Pöppelmann, and P. Schwabe. "Post-quantum key exchange – a new hope". In: *to appear at USENIX*. https://cryptojedi.org/papers/newhope-20160328.pdf. 2016 (cit. on pp. v, 1, 15, 18–24, 28, 29, 37, 40, 43, 47, 49).

[2] ARM Ltd. *Cortex-M Series*. www.arm.com/products/processors/cortex-m/, (accessed 2015-12-10). 2015 (cit. on p. 31).

[3] P. Barrett. "Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor". In: *Advances in Cryptology – CRYPTO '86*. Vol. 263. https://choucroutage.com/Papers/SideChannelAttacks/crypto-1986-barrett.pdf. 1987, pp. 311–323 (cit. on pp. 28, 38).

[4] P. Benioff. "The computer as a physical system: A microscopic quantum mechanical Hamiltonian model of computers as represented by Turing machines". In: *Journal of Statistical Physics* 22.5 (1980). https://www.researchgate.net/publication/226754042_Benioff_PA_The_computer_as_a_physical_system_a_microscopic_quantum_mechanical_hamiltonian_model_of_computers_as_represented_by_Turing_machines_J_Stat_Phys_225_563-591, pp. 563–591 (cit. on p. 9).

[5] D. J. Bernstein. "ChaCha, a variant of Salsa20". In: *Workshop Record of SASC 2008: The State of the Art of Stream Ciphers*. http://cr.yp.to/chacha/chacha-20080128.pdf. 2008 (cit. on p. 43).

[6] D. J. Bernstein, J. Buchmann, and E. Dahmen. *Post Quantum Cryptography*. http://www.e-reading.club/bookreader.php/135832/Post_Quantum_Cryptography.pdf. Springer Science & Business Media, 2008 (cit. on p. 11).

[7] J. W. Bos, C. Costello, M. Naehrig, and D. Stebila. "Post-quantum key exchange for the TLS protocol from the ring learning with errors problem". In: *2015 IEEE Symposium on Security and Privacy*. https://eprint.iacr.org/2014/599.pdf. 2015, pp. 553–570 (cit. on pp. 1, 18, 19).

[8] J. W. Bos, A. Dudeanu, and D. Jetchev. "Collision bounds for the additive Pollard rho algorithm for solving discrete logarithms". In: *Journal of Mathematical Cryptology* 8.1 (2014). https://eprint.iacr.org/2012/087.pdf, pp. 71–92 (cit. on p. 8).

[9] Z. Brakerski, A. Langlois, C. Peikert, O. Regev, and D. Stehlé. "Classical Hardness of Learning with Errors". In: *Proceedings of the Forty-fifth Annual ACM Symposium on Theory of Computing*. http://arxiv.org/pdf/1306.0281.pdf. 2013, pp. 575–584 (cit. on p. 21).

[10] A. Cho. "Quantum or not, controversial computer yields no speedup". In: *Science* 344.6190 (2014). http://katzgraber.org/currents/media/press/2014-06-science_long.PDF, pp. 1330–1331 (cit. on p. 9).

[11] R. de Clercq, S. S. Roy, F. Vercauteren, and I. Verbauwhede. "Efficient software implementation of ring-LWE encryption". In: *Design, Automation & Test in Europe Conference & Exhibition, DATE 2015*. https://eprint.iacr.org/2014/725.pdf. 2015, pp. 339–344 (cit. on pp. 46, 47).

[12] F. Cohen. *A Short History of Cryptography*. http://web.itu.edu.tr/~orssi/dersler/cryptography/Chap2-1.pdf, (accessed 2016-04-03). 1990 (cit. on p. 6).

[13] J. W. Cooley and J. W. Tukey. "An algorithm for the machine calculation of complex Fourier series". In: *Mathematics of Computation* 19.90 (1965). https://web.stanford.edu/class/cme324/classics/cooley-tukey.pdf, pp. 297–301 (cit. on p. 26).

[14] J. Daemen, M. Peeters, G. V. Assche, and V. Rijmen. *Keccak Code Package*. https://github.com/gvanas/KeccakCodePackage/, commit c004b5d3fc9ddc5ae784fbe9a6eb2f3e09811c5f. 2016 (cit. on pp. 33, 44, 45).

[15] N. S. Dattani and N. Bryans. "Quantum factorization of 56153 with only 4 qubits". In: *arXiv preprint arXiv:1411.6758* (2014). https://arxiv.org/pdf/1411.6758v3.pdf (cit. on p. 9).

[16] L. De Feo, D. Jao, and J. Plût. "Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies". In: *Journal of Mathematical Cryptology* 8.3 (2014). https://eprint.iacr.org/2011/506.pdf, pp. 209–247 (cit. on p. 11).

[17] W. Diffie and M. Hellman. "New directions in cryptography". In: *IEEE Transactions on Information Theory* 22.6 (1976). https://ee.stanford.edu/~hellman/publications/24.pdf, pp. 644–654 (cit. on p. 6).

[18] J. Ding, X. Xie, and X. Lin. "A Simple provably secure key exchange scheme based on the learning with errors problem." In: *IACR Cryptology ePrint Archive* 2012.1 (2012). https://eprint.iacr.org/2012/688.pdf, p. 688 (cit. on pp. 1, 14).

[19] P. A. M. Dirac. "A new notation for quantum mechanics". In: *Mathematical Proceedings of the Cambridge Philosophical Society* 35.3 (). http://www.ifsc.usp.br/~lattice/wp-content/uploads/2014/02/Dirac_notation.pdf, pp. 416–418 (cit. on p. 9).

[20] R. Dridi and H. Alghassi. "Prime factorization using quantum annealing and computational algebraic geometry". In: *arXiv preprint arXiv:1604.05796* (2016). https://arxiv.org/pdf/1604.05796.pdf (cit. on p. 10).

[21] L. Ducas, A. Durmus, T. Lepoint, and V. Lyubashevsky. "Lattice signatures and bimodal Gaussians". In: *Advances in Cryptology – CRYPTO 2013*. Vol. 8042. https://eprint.iacr.org/2013/383.pdf. 2013, pp. 40–56 (cit. on p. 46).

[22] D-Wave. *D-Wave Systems Breaks the 1000 Qubit Quantum Computing Barrier*. http://www.dwavesys.com/press-releases/d-wave-systems-breaks-1000-qubit-quantum-computing-barrier, (accessed 2016-04-03). 2015 (cit. on p. 9).

[23] J. H. Ellis. "The possibility of secure non-secret digital encryption". In: *UK Communications Electronics Security Group* (1970). http://cryptocellar.org/cesg/possnse.pdf (cit. on p. 6).

[24] P. van Emde Boas. *Another NP-complete partition problem and the complexity of computing short vectors in a lattice*. https://staff.fnwi.uva.nl/p.vanemdeboas/vectors/page1.html. Universiteit van Amsterdam. Mathematisch Instituut, 1981 (cit. on p. 14).

[25] A. Emerencia. *Multiplying huge integers using Fourier transforms*. http://www.cs.rug.nl/~ando/pdfs/Ando_Emerencia_multiplying_huge_integers_using_fourier_transforms_paper.pdf, (accessed 2016-04-23). 2007 (cit. on p. 25).

[26] U. Feige and D. Micciancio. "The inapproximability of lattice and coding problems with preprocessing". In: *Journal of Computer and System Sciences* 69.1 (2004). https://cseweb.ucsd.edu/~daniele/papers/GapCVPP.pdf, pp. 45–67 (cit. on p. 14).

[27] R. P. Feynman. "Simulating physics with computers". In: *International Journal of Theoretical Physics* 21.6 (1982). https://www.cs.berkeley.edu/~christos/classics/Feynman.pdf, pp. 467–488 (cit. on p. 9).

[28] M. Fürer. "Faster integer multiplication". In: *SIAM Journal on Computing* 39.3 (2009). http://wwwmath.uni-muenster.de/u/cl/WS2007-8/mult.pdf, pp. 979–1005 (cit. on p. 24).

[29] W. M. Gentleman and G. Sande. "Fast Fourier Transforms: for fun and profit". In: *Fall Joint Computer Conference*. Vol. 29. http://cis.rit.edu/class/simg716/FFT_Fun_Profit.pdf. 1966, pp. 563–578 (cit. on pp. 24, 26, 37).

[30] O. Goldreich, D. Micciancio, S. Safra, and J.-P. Seifert. "Approximating shortest lattice vectors is not harder than approximating closest lattice vectors". In: *Information Processing Letters* 71.2 (1999). https://cseweb.ucsd.edu/~daniele/papers/GMSS.pdf, pp. 55–61 (cit. on p. 14).

[31] O. Goldreich and S. Goldwasser. "On the Limits of Non-approximability of Lattice Problems". In: *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*. http://www.wisdom.weizmann.ac.il/~oded/PSX/lp.pdf. 1998, pp. 1–9 (cit. on pp. 13, 14).

[32] L. K. Grover. "A Fast Quantum Mechanical Algorithm for Database Search". In: *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*. https://arxiv.org/pdf/quant-ph/9605043.pdf. 1996, pp. 212–219 (cit. on p. 10).

[33] M. Heideman, D. Johnson, and C. Burrus. "Gauss and the history of the Fast Fourier Transform". In: *IEEE ASSP Magazine* 1.4 (Oct. 1984). http://ocw.nctu.edu.tw/course/fourier/supplement/heideman-johnson-etal1985.pdf, pp. 14–21 (cit. on p. 26).

[34] L. N. Hoang. *P versus NP: A Crucial Open Problem*. http://www.science4all.org/article/pnp/, (accessed 2016-04-03). 2013 (cit. on p. 5).

[35] J. Hoffstein, J. Pipher, and J. H. Silverman. In: *Algorithmic Number Theory: Third International Symposiun, ANTS-III Portland*. http://grouper.ieee.org/groups/1363/lattPK/submissions/ntru.pdf (cit. on p. 11).

[36] M. Hutter and P. Schwabe. *Curve25519 for ARM Cortex-M0*. http://munacl.cryptojedi.org/curve25519-cortexm0.shtml, (accessed 2016-04-03). 2013 (cit. on pp. 53, 54).

[37] M. Hutter and P. Schwabe. *Curve25519 for ARM Cortex-M0*. http://munacl.cryptojedi.org/data/curve25519-cortexm0-20150813.tar.bz2, (accessed 2015-12-10). 2016 (cit. on pp. 36, 45).

[38] W. S. Jevons. *The Principles of Science*. https://archive.org/details/theprinciplesof00jevoiala, (accessed 2016-04-03). 1874 (cit. on p. 6).

[39] M. Kaplan. "Quantum attacks against iterated block ciphers". In: *arXiv preprint arXiv:1410.1434* (2014). https://arxiv.org/pdf/1410.1434.pdf (cit. on p. 10).

[40] A. Karatsuba and Y. Ofman. "Multiplication of multidigit numbers on automata". In: *Soviet physics doklady*. Vol. 7. https://www.researchgate.net/profile/Anatolii_Karatsuba/publication/234346907_Multiplication_of_Multidigit_Numbers_on_Automata/links/00b495357e64391356000000.pdf?origin=publication_detail. 1963, pp. 595–596 (cit. on p. 8).

[41] A. K. Lenstra, H. W. Lenstra, and L. Lovász. "Factoring polynomials with rational coefficients". In: *Mathematische Annalen* 261.4 (1982). http://infoscience.epfl.ch/record/164484/files/nscan4.PDF, pp. 515–534 (cit. on p. 14).

[42] A. K. Lenstra, H. W. Lenstra, M. S. Manasse, and J. M. Pollard. *The development of the number field sieve*. https://infoscience.epfl.ch/record/164684/files/164684.pdf. Springer, 1993 (cit. on p. 8).

[43] J. Leys, E. Ghys, and A. Alvarez. *Dimensions*. http://www.dimensions-math.org/, (accessed 2016-04-03). 2010 (cit. on p. 21).

[44] H. Li and L. Yang. "Quantum differential cryptanalysis to the block ciphers". In: *Applications and Techniques in Information Security*. https://arxiv.org/pdf/1511.08800.pdf. 2015, pp. 44–51 (cit. on p. 10).

[45] Libopencm3. *Open-Source ARM Cortex M microcontroller library*. https://github.com/libopencm3/libopencm3, commit 492a943b7e448469cd8e88f60fda41ef46aa3b2a. 2016 (cit. on p. 36).

[46] R. Lindner and C. Peikert. "Better Key Sizes (and Attacks) for LWE-Based Encryption". In: *Topics in Cryptology - CT-RSA 2011*. https://web.eecs.umich.edu/~cpeikert/pubs/lwe-analysis.pdf. 2011, pp. 319–339 (cit. on p. 15).

[47] P. Longa and M. Naehrig. *Speeding up the Number Theoretic Transform for Faster Ideal Lattice-Based Cryptography*. https://eprint.iacr.org/2016/504.pdf, (accessed 2015-07-05). 2016 (cit. on pp. 50, 51).

[48] V. Lyubashevsky, C. Peikert, and O. Regev. "On Ideal Lattices and Learning with Errors over Rings". In: *Advances in Cryptology – EUROCRYPT 2010*. http://www.di.ens.fr/~lyubash/papers/ringLWE.pdf. 2010, pp. 1–23 (cit. on pp. 15, 20).

[49] E. Martín-López et al. "Experimental realization of Shor's quantum factoring algorithm using qubit recycling". In: *Nature Photonics* 6.11 (2012). https://arxiv.org/pdf/1111.4147.pdf, pp. 773–776 (cit. on p. 9).

[50] N. Mathewson. *Cryptographic directions in Tor*. https://people.torproject.org/~nickm/slides/nickm-rwc-presentation.pdf, (accessed 2016-06-20). 2016 (cit. on p. v).

[51] R. McEliece. "A public-key cryptosystem based on algebraic". In: *Coding Thv* 4244.1 (1978). http://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19780016269.pdf, pp. 114–116 (cit. on p. 11).

[52] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. http://cacr.uwaterloo.ca/hac/. CRC press, 20011 (cit. on pp. 5, 29).

[53] R. C. Merkle. "A certified digital signature". In: *Advances in Cryptology – CRYPTO' 89 Proceedings*. https://discovery.csc.ncsu.edu/Courses/csc774-F11/reading-assignments/Merkle-Tree.pdf. 1990, pp. 218–238 (cit. on p. 11).

[54] D. Micciancio. "The shortest vector in a lattice is hard to approximate to within some constant". In: *SIAM Journal on Computing* 30.6 (2001). https://cseweb.ucsd.edu/~daniele/papers/SVP.pdf, pp. 2008–2035 (cit. on p. 14).

[55] D. Micciancio and P. Voulgaris. "Faster exponential time algorithms for the shortest vector problem". In: *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*. https://cseweb.ucsd.edu/~daniele/papers/Sieve.pdf. 2010, pp. 1468–1480 (cit. on p. 12).

[56] V. S. Miller. "Use of Elliptic Curves in cryptography". In: *Advances in Cryptology – CRYPTO '85 Proceedings*. https://www.docdroid.net/58Q3zwX/miller-ecc-lncs85.pdf.html. 1986, pp. 417–426 (cit. on p. 8).

[57] P. L. Montgomery. "Modular multiplication without trial division". In: *Mathematics of Computation* 44.170 (1985). http://www.ams.org/journals/mcom/1985-44-170/S0025-5718-1985-0777282-X/S0025-5718-1985-0777282-X.pdf, pp. 519–521 (cit. on pp. 29, 38).

[58] National Institute of Standards and Technology. *Workshop on Cybersecurity in a Post-Quantum World*. http://www.nist.gov/itl/csd/ct/post-quantum-crypto-workshop-2015.cfm, (accessed 2016-03-11). 2015 (cit. on p. v).

[59] National Security Agency. *NSA Suite B Cryptography*. https://www.nsa.gov/ia/programs/suiteb_cryptography/, (accessed 2016-03-11) (cit. on p. v).

[60] National Security Agency Central Security Service. *Classification guide for NSA/CSS quantum computing research 10-25*. https://snowdenarchive.cjfe.org/greenstone/collect/snowden1/index/assoc/HASHd3c4.dir/doc.pdf, (accessed 2016-01-04). 2011 (cit. on p. 9).

[61] M. Neikes and N. Samwel. *ARM implementation of the ChaCha20 block cipher*. https://gitlab.science.ru.nl/mneikes/arm-chacha20, commit 40fd6708. 2016 (cit. on pp. 43, 45, 50).

[62] K. Neupane. "Two-party key establishment: From passive to active security without introducing new assumptions". In: *Groups-Complexity-Cryptology* 4.1 (2012), pp. 1–17 (cit. on p. 8).

[63] T. Oder, T. Poppelmann, and T. Guneysu. "Beyond ECDSA and RSA: Lattice-based digital signatures on constrained devices". In: *Design Automation Conference (DAC)*. https://www.sha.rub.de/media/attachments/files/2014/06/bliss_arm.pdf. 2014, pp. 1–6 (cit. on pp. 46–48).

[64] J. Patarin. "Hidden Field Equation and Isomorphism of Polynomials". In: *Advances in Cryptology – EUROCRYPT '96: International Conference on the Theory and Application of Cryptographic Techniques*. http://www.minrank.org/hfe.pdf. 1996, pp. 33–48 (cit. on p. 11).

[65] C. Peikert. "Lattice cryptography for the Internet". In: *Post-Quantum Cryptography*. https://web.eecs.umich.edu/~cpeikert/pubs/suite.pdf. 2014, pp. 197–219 (cit. on pp. 1, 16–18, 49).

[66] C. Peikert. "Public-key cryptosystems from the worst-case shortest vector problem". In: *Proceedings of the forty-first annual ACM symposium on Theory of computing*. http://drops.dagstuhl.de/opus/volltexte/2009/1892/pdf/08491.PeikertChris.Paper.1892.pdf. 2009, pp. 333–342 (cit. on p. 14).

[67] V. Pieterse and P. E. Black. *big-O notation*. http://www.nist.gov/dads/HTML/bigOnotation.html, (accessed 2016-03-11). 2012 (cit. on pp. 3, 4).

[68] J. M. Pollard. "A monte carlo method for factorization". In: *BIT Numerical Mathematics* 15.3 (1975). https://www.cs.cmu.edu/afs/cs.cmu.edu/Web/People/avrim/451f11/lectures/lect1122_Pollard.pdf, pp. 331–334 (cit. on p. 8).

[69] T. Pöppelmann and T. Güneysu. "Towards Practical Lattice-Based Public-Key Encryption on Reconfigurable Hardware". In: *Selected Areas in Cryptography – SAC 2013*. Vol. 8282. https://www.ei.rub.de/media/sh/veroeffentlichungen/2013/08/14/lwe_encrypt.pdf. 2013, pp. 68–85 (cit. on p. 46).

[70] T. Pöppelmann, T. Oder, and T. Güneysu. "High-Performance ideal lattice-based cryptography on 8-bit ATxmega microcontrollers". In: *Progress in Cryptology – LATINCRYPT 2015: 4th International Conference on Cryptology and Information Security in Latin America*. https://eprint.iacr.org/2015/382.pdf. 2015, pp. 346–365 (cit. on p. 46).

[71] J. Proos and C. Zalka. "Shor's discrete logarithm quantum algorithm for elliptic curves". In: *arXiv preprint quant-ph/0301141* (2003). https://arxiv.org/pdf/quant-ph/0301141v2.pdf (cit. on p. 10).

[72] O. Regev. "On Lattices, Learning with Errors, Random Linear Codes, and Cryptography". In: *Proceedings of the Thirty-seventh Annual ACM Symposium on Theory of Computing*. http://www.cims.nyu.edu/~regev/papers/qcrypto.pdf. 2005, pp. 84–93 (cit. on pp. 14, 15, 20).

[73] O. Regev. "The learning with errors problem". In: *Invited survey in CCC* (2010). http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.205.2622&rep=rep1&type=pdf (cit. on p. 15).

[74] R. L. Rivest, A. Shamir, and L. Adleman. "A method for obtaining digital signatures and public-key cryptosystems". In: *Communications of the ACM* 21.2 (1978). http://web.mit.edu/6.857/OldStuff/Fall03/ref/rivest78method.pdf, pp. 120–126 (cit. on p. 8).

[75] A. Schönhage and V. Strassen. "Fast multiplication of large numbers". In: *Computing* 7.3 (1971). http://moonflare.com/misc/Schnelle%20Multiplikation%20gro%DFer%20Zahlen.html, pp. 281–292 (cit. on p. 8).

[76] C. E. Shannon. "A mathematical theory of communication". In: *The Bell System Technical Journal* 27.3 (1948). http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6773024, pp. 379–423 (cit. on p. 9).

[77] P. W. Shor. "Polynomial-time algorithms for prime factorization and aiscrete logarithms on a quantum computer". In: *SIAM Journal on Computing* 26.2 (1997). https://arxiv.org/pdf/quant-ph/9508027v2.pdf, pp. 1484–1509 (cit. on pp. 1, 10).

[78] C.-L. Tian, W. Wei, and D.-D. Lin. "Solving closest vector instances using an approximate shortest independent vectors oracle". In: *Journal of Computer Science and Technology* 30.6 (2015). https://eprint.iacr.org/2014/545.pdf, pp. 1370–1377 (cit. on p. 13).

[79] *Tor Project: Anonymity Online*. https://www.torproject.org/, (accessed 2016-01-04) (cit. on p. 18).

[80] A. M. Turing. "On computable numbers, with an application to the Entscheidungsproblem". In: *Journal of Math* 58.5 (1936). https://www.cs.virginia.edu/~robins/Turing_Paper_1936.pdf, pp. 345–363 (cit. on p. 4).

[81] J. Utsler. *Quantum Computing Might Be Closer Than Previously Thought*. http://www.ibmsystemsmag.com/mainframe/trends/IBM-Research/quantum_computing/, (accessed 2016-03-03). 2013 (cit. on p. 9).

[82] A. Van Herrewege, V. van der Leest, A. Schaller, S. Katzenbeisser, and I. Verbauwhede. "Secure prng seeding on commercial off-the-shelf microcontrollers". In: *Proceedings of the 3rd international workshop on Trustworthy embedded devices*. https://eprint.iacr.org/2013/304.pdf. ACM. 2013, pp. 55–64 (cit. on p. 50).

[83] N. Xu et al. "Quantum Factorization of 143 on a Dipolar-Coupling Nuclear Magnetic Resonance System". In: *Phys. Rev. Lett.* 108.13 (Mar. 2012). https://arxiv.org/pdf/1111.3726.pdf, p. 130501 (cit. on p. 9).