Leibniz Universität Hannover
Fakultät für Elektrotechnik und Informatik
Institut für Theoretische Informatik

Bachelor Thesis

Runtime Analysis of Lattice-Based Cryptographic Algorithms

Neal Asprion
Matriculation number: 10007563

November 7, 2019

Primary Examiner:	Prof. Dr. rer. nat. Heribert Vollmer
Secondary Examiner:	Dr. rer. nat. Arne Meier
Supervised by:	Dr. rer. nat. Maurice Chandoo

Declaration of Authorship

I hereby declare that the thesis submitted is my own unaided work. All direct or indirect sources used are acknowledged as references.
This paper was not previously presented to another examination board and has not been published.

Erklärung der eigenständigen Arbeit

Hiermit versichere ich, die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegeben Quellen und Hilfsmittel verwendet zu haben.
Die Arbeit hat in gleicher oder ähnlicher Form noch keinem anderen Prüfungsamt vorgelegen.

Hannover, November 7, 2019

Neal Asprion

Contents

List of Figures

1. Introduction

Due to the extensive research in the last few years quantum computers pose one of the biggest threats to future secure online communication. As Chen et al. (2016) [Che+16] stress in their article, they can break encryption, hashing and signing protocols widely in use. Most notably RSA, Diffie-Hellman key exchange and elliptic curve cryptosystems are broken. Shor's algorithm can, on a sufficiently large quantum computer, efficiently solve the mathematical problems which these protocols are based on. It can for example easily solve the discrete log problem on elliptic curves. This means these crypto systems are no longer secure.

It is unknown when exactly such large quantum computers will be available, but to be prepared the research has to start decades before the first one is build. This is why in recent years governments started funding various post quantum cryptography projects and due to some recent breakthroughs have, according to Chen et al. (2016), even increased the funding. Scientist turn to alternatives that can withstand Shor's algorithm and have not yet been broken by any other quantum algorithm.

Lattice-based cryptography, Code-based cryptography, multivariate polynomial cryptography and Hash-based signatures are the most promising fields of research mentioned in Chen et al. (2016). All of these do require a lot more improvements and validation to be usable in practice, as they require too large keys, too much computation time to de- or encrypt or have just recently been discovered. Especially the short amount of time some algorithms are in consideration as a new cryptography standard makes them possibly vulnerable to undiscovered attacks. Good cryptography requires years of research and lots of time searching for possible exploits or new attack vectors.

1.1. Lattice-based cryptography

Out of those fields lattice-based cryptography has gathered a lot of interest, as the algorithms are simple, efficient and often highly parallelizable. The first promising lattice-based system was the Hoffstein-Pipher-Silverman public-key-encryption, also know as "NTRU". It was suggested in 1998, but has been heavily improved since then. This means promising lattice-based encryption has not been around for a long time compared to for example code-based cryptography which was introduced by McElice in 1978 as mentioned by Bernstein et al. (2009) [BBD09]. But due to it's good algorithms it can easily compete with the other systems which are

in consideration for post quantum use.

This does not mean the optimal system has already been found. Chen et al. (2016) warn that there are still no precise estimates for every analysis technique or attack vector, on how secure lattice-based encryption is.

1.2. NIST Post-Quantum Cryptography Standardization Process

The relevance of Latticed-based cryptography becomes clear when looking at an ongoing competition hosted by the *National Institute of Standards and Technology* (NIST), which aims at selecting and standardizing possible post quantum cryptography algorithms. The competition was introduced in December 2016 and is currently in its second round [Ala+19]. There are still multiple lattice-based signing and public-key encryption schemes present in the second round.

This bachelor thesis takes a look at NewHope and FrodoKEM. These systems should work on any machine with a very short execution time. They ideally are fast and memory efficient so they can even be implemented and used on any small microcontroller. The NIST has requested the scientific community to extensively test these algorithms under real world conditions especially on microcontrollers.

This work provides a short introduction to Lattice-based cryptography and the two systems. Each system is then tested and analysed for its usage of assembly operations. The paper provides an overview of the amount of operations used and suggests which operations a microcontroller should exceed at to effectively encrypt with theses systems.

2. Lattices

Even though a lattice is simple to describe, there are known problems which currently are NP-hard to solve. No known quantum algorithm does solve these problems with a relevant time advantage compared to normal, known and applicable algorithms.

2.1. Definitions

Definition 2.1.1. A lattice L is a n dimensional structure. It can be constructed from a set of linear independent vectors. If the set $b_1, ..., b_m$ of linear independent vectors in $\mathbb{R}^n$ is given. The lattice can be constructed as follows:

$$L(b_1, ..., b_m) = \{ \sum_{i=1}^{m} x_i b_i | \, x_i \in \mathbb{Z} \}$$

This means the lattice of $b_1, ..., b_m$ is constructed out of all possible integer combinations of the vectors. These vectors are a basis of L and are for simplicity often written as matrix B.

The vectors may be from any other set, but as the focus in this paper is on algorithms for classical computers, the sets $\mathbb{C}^n$ and $\mathbb{I}^n$ are ignored, as they will be represented using one of the other sets when being implemented.

Definition 2.1.2. The Euclidean length of a vector $v \in \mathbb{R}^n$, where $(v_1, ..., v_n)$ are the elements of the vector, will be written as $||v||$ and is defined as:

$$||v|| = \sqrt{\sum_{i=1}^{n} v_i^2}$$

Definition 2.1.3. The distance of a vector t to a lattice Λ is the Euclidean length of the difference between t and the vector $v \in \Lambda$ which is closest to t. This means the smallest distance that can be found between t and any vector in Λ.

$$dist(t, \Lambda) = inf\{||v - t|| : v \in \Lambda\} \text{ [Pöp+19]}$$

Definition 2.1.4. The minimal distance λ_1 of a lattice L is the Euclidean length of the shortest vector, excluding the zero vector [Pöp+19].

$$\lambda_1(L) = min_{v \in L \setminus \{0\}} ||v||$$

Definition 2.1.5. The n-th minimal distance λ_n of a lattice L is the Euclidean length of the n-th shortest vector, excluding the zero vector, which is linearly independent from all shorter vectors.

A good definition is given by Micciancio: "The *successive minima* $\lambda_i(L)$ [...] are defined as the smallest values $\lambda_i(L)$ such that the sphere of radius $\lambda_i(L)$ centered around the origin contains at least i linearly in-dependent lattice vectors."[Mic08, p. 84]

Definition 2.1.6. In consistency with [Pöp+19] $x \xleftarrow{\$} \chi$ is used to describe the sampling of $x \in R$ according to χ, if χ is a probability distribution over R.

It may also be written as $y \xleftarrow{\$} A$ in this case A is an algorithm, which is run with a random coin and whose output is assigned to y.

2.2. Problems on Lattices

Multiple interesting problems on and based on lattices exist. The most relevant problem for the following cryptographic system and lattice based cryptography in general is the *shortest vector problem* (SVP). It will be introduced in the following sections as well as the *approximate shortest vector problem* (SVP_γ).

2.2.1. SVP

Given a basis $B \in \mathbb{Q}^{n \times n}$ the *shortest vector problem* is the task to find a non-zero vector v in the lattice $L(B)$ such that $||v|| = \lambda_1(L(B))$. In other words, find one from all vectors whose Euclidean length is equal to the minimal distance of $L(B)$.

There often are multiple vectors per lattice who solve the SVP. Given the basis $B_e = \{\begin{pmatrix}0\\1\end{pmatrix}, \begin{pmatrix}1\\0\end{pmatrix}\}$ it becomes obvious that 4 possible solutions to the SVP in the lattice $L_e(B_e)$ exist. Namely $\begin{pmatrix}1\\0\end{pmatrix}, \begin{pmatrix}0\\1\end{pmatrix}, \begin{pmatrix}-1\\0\end{pmatrix}$ and $\begin{pmatrix}0\\-1\end{pmatrix}$ as they all have an Euclidean length of 1, which is the minimal distance of $L_e(B_e)$. This can also be seen in Figure 2.1.

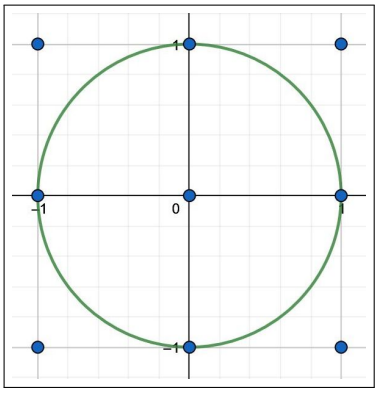

Figure 2.1.: Visual representation of the minimal distance λ_1 of the lattice L_e
Created with geogebra (21.10.2019)

2.2.2. Approximate SVP

The SVP can easily be expanded to the *approximate* SVP, also called SVP_γ. Where $\gamma \geq 1$ is added to λ_1 as a factor, to allow an approximate solution. Now a v with $||v|| \leq \gamma\lambda_1(L(B))$ is required to solve the problem. The precision of the approximation decreases with the growth of γ, which makes the problem easier to solve. $\gamma = 1$ gives the same result as the SVP.

Reusing the basis B_e and setting $\gamma = 2$, the Euclidean length of v now only has to be less than or equal to 2, as $\gamma * \lambda_1(L_e(B_e)) = 2 * 1 = 2$.
This means for example $\begin{pmatrix}1\\1\end{pmatrix}, \begin{pmatrix}2\\0\end{pmatrix} \in L_e(B_e)$ are now also a solution to the problem, as $|| \begin{pmatrix}1\\1\end{pmatrix} || = \sqrt{2} < 2$ and $|| \begin{pmatrix}2\\0\end{pmatrix} || = \sqrt{4} = 2$. This can also be seen in Figure 2.2, as well as some more vectors which now solve the problem.

2.2.3. Approximate SIVP

The *Approximate Shortest Independent Vectors Problem* ($SIVP_{n,\gamma}$) has the parameters n and γ and asks to find at least n linear independent vectors V in a lattice L where $||v|| \leq \gamma * \lambda_n$ applies to all $v \in V$.

This is the approximation of the n-th minimal distance and can be understood as multiplying the radius of the circle used in Micciancio's definition with the factor γ. While the amount of vectors to find stays the same, the area in which they may be becomes bigger.

2.2.4. Approximate GapSVP

$GapSVP_\gamma$ is a variant of SVP_γ which takes an additional parameter d. It decides if the Euclidean length of the shortest vector of $L(B)$ is smaller than or equal to

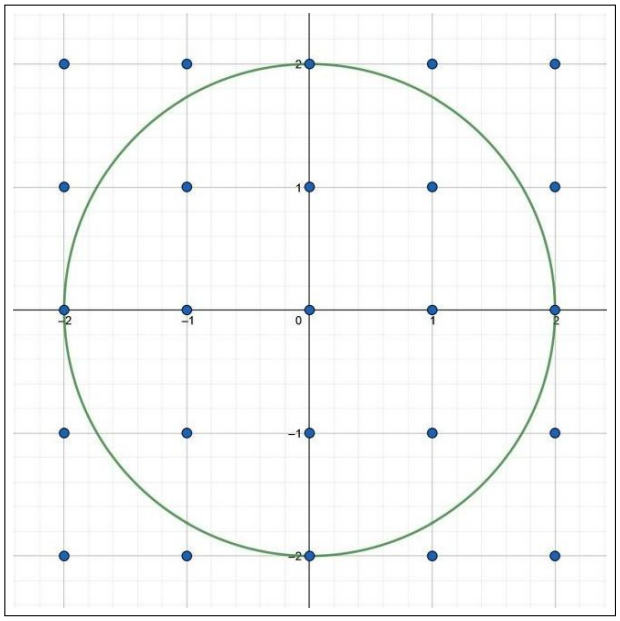

Figure 2.2.: Visual representation of the minimal distance λ_1 of the lattice L_e multiplied with the approximation factor $\gamma = 2$
Created with geogebra (21.10.2019)

d and in this case returns True. If this is not the case False is only returned when the length is greater than or equal to $\gamma * d$. If $d < \lambda_1(L(B)) \leq \gamma * d$ the output is undefined.

$$GapSVP_\gamma(B, d) = \begin{cases} \text{True}, & \text{if } \lambda_1(L(B)) \leq d \\ \text{False}, & \text{if } \lambda_1(L(B)) > \gamma * d \\ \text{Undefined}, & \text{otherwise} \end{cases}$$

While it is easy to see that $GapSVP_\gamma$ can be reduced to SVP_γ, a reduction from SVP_γ to $GapSVP_\gamma$ has not been found nor disproven for interesting γ with $2^n > \gamma \geq 1$ [Ebe18]. There is an upper bound for interesting γ as the complexity of the problem decreases with decreasing precision, as shown in the following section.

2.2.5. Complexity of SVP$_\gamma$

In the given examples the SVP_γ does not seem very hard to solve, but in reality it is. The simplicity of the given examples is based on two factors.

Firstly the little number of dimensions. The example only had 2 dimensions while in cryptography lattices have up to 500. It is obvious that adding 498 more

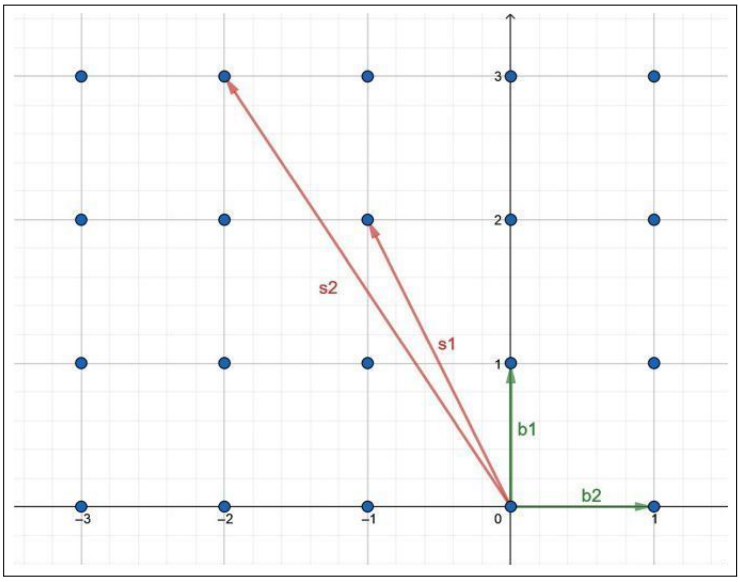

Figure 2.3.: Bases of different complexity S_1 and B_e of the same lattice.
Created with geogebra (21.10.2019)

vectors with 500 coordinates each to the base makes the problem more complicated.

The second simplification comes from choosing ideal short vectors. With the basis B_e one can easily imagine the lattice which is constructed out of them. It is also relatively easy to find an integer combination of these base vectors to get to any given point in the lattice L_e.

Given for example the basis $S_1 = \{\begin{pmatrix}-2\\3\end{pmatrix}, \begin{pmatrix}-1\\2\end{pmatrix}\}$ or $S_2 = \{\begin{pmatrix}17\\16\end{pmatrix}, \begin{pmatrix}16\\15\end{pmatrix}\}$ it does not become immediately clear that they indeed are bases of the already presented lattice L_e, as shown for S_1 in Figure 2.3. It is also not as easy as with B_e to figure out λ_1 without constructing the lattice or transforming the basis.

This is why basis reduction is one of the fundamental tools for solving lattice problems. Basis reduction can be considered to be the SIVP with $\gamma = 1$ and $n = \dim(L)$.

Hardness of SVP$_\gamma$

While the complexity in the cases above is the result of the dimension n or the given basis, the hardness of these problems is usually specified for a certain γ while the actual runtime depends on n. Multiple paper prove different hardnesses for various γ as shown by Eberhardt [Ebe18] on page 31. The following proves are sorted by increasing γ and are shown for the ℓ_2 norm.

Ajtai [Ajt98] showed that the SVP_γ with the smallest allowed γ, which is $\gamma = 1$ and therefore makes this problem equal to the standard SVP, is NP-hard using randomized polynomial time reduction.

The SVP_γ with the biggest γ which has been proven to be NP-hard is for $\gamma = 2^{(\log n)^{1-\epsilon}}$. This was proven by Haviv and Regev [HR07] under the assumption that NP is not a subset nor equal $RTIME(2^{poly(\log n)})$. A definition of $RTIME$ can be found in German in [Ebe18] and in English in [HR07].

The next relevant γ mentioned by Eberhardt is $\gamma = \mathcal{O}(\sqrt{n/\log n})$. It was shown in 1998 by Goldreich and Goldwasser [GG98] that this SVP_γ is in the intersection of NP and $coAM$. This means it is unlikely to be NP-hard because as Eberhardt mentions this would lead to a collapse of polynomial-time hierarchy, which is a widely used theorem in computational complexity theory. This also implies that any SVP_γ with $\gamma > \mathcal{O}(\sqrt{n/\log n})$ is most likely not NP-hard.
On the other hand the smallest γ which has been used in a cryptographic function is $\gamma = \tilde{\mathcal{O}}(n)$. According to Eberhardt it has been used in one-way functions. As it is unlikely to proof NP-hardness for $\gamma > \mathcal{O}(\sqrt{n/\log n})$ researchers try to find cryptographic systems which move the γ further down to hopefully one day drop below this border.

2.2.6. CVP

For completeness the *closes vector problem* (CVP), another problem on lattices, is also defined.
A basis $B \in \mathbb{Q}^{n \times n}$ and a vector $t \in \mathbb{Q}^n$ are given [Ebe18]. The solution to the CVP is the vector v with $||v - t|| = dist(t, L(B))$.

There also exists an approximate CVP, also called CVP_γ, which introduces γ for the same reason as the SVP_γ. It is multiplied with $dist(t, L(B))$ and we now search a vector v with $||v - t|| \leq \gamma * dist(t, L(B))$.

The CVP is closely related to the SVP, as it is the inhomogeneous version of the latter [Pöp+19]. It has also been proven to be NP hard up to $\gamma = 2^{(\log n)^{1-\epsilon}}$ with $\epsilon = (\log \log n)^{-c}$ fo any constant $c < \frac{1}{2}$ [DKS98].

3. Learning With Errors

Taking another step towards cryptography *Learning With Errors* (LWE) is introduced. This problem can be used as the foundation of various encryptions, namely *Chosen Plaintext Attack* (CPA) and *Chosen Cyphertext Attack* (CCA) secure public-key encryption, identity-based encryption and fully-homomorphic encryption schemes [Pöp+19].

LWE is based on *Learning Parity with Noise* (LPN) which in turn is based on *Learning Parity* (LP). This is why LP is introduced first and afterwards the problem will become gradually more complicate.

3.1. Learning Parity

Learning parity is a relatively simple problem. For an unknown secret $s \in \{0,1\}^n$ the solver receives $m \geq n$ samples. Each sample is a pair of (a_p, b_p) where a_p is uniformly distributed over $\{0,1\}^n$ and $b_p =< a_p, s > \mod 2$, with $< a_i, s > = \sum_{i=1}^{n} a_p[i] * s[i]$ [Raz18][Pöp+19]. LP can easily be solved by the Gaussian algorithm in $\mathcal{O}(n)$ [Ebe18].

Given the samples $\{(\{1,1,1\},0);(\{1,1,0\},1);(\{1,0,1\},0)\}$, the following matrix can be constructed:

$a[1]$	$a[2]$	$a[3]$	b
1	1	1	0
1	1	0	1
1	0	1	0

Subtracting the second from the first row, then the first from the last and finally the last from the second row, results in the following matrix, which solves the problem:

$a[1]$	$a[2]$	$a[3]$	b
0	0	1	1
0	1	0	0
1	0	0	1

So the secret was $\{1, 0, 1\}$.

The amount of samples required varies depending on the algorithm used to solve it. There are for example algorithms which require an exponential amount of samples because they focus on space efficiency.

3.2. Learning Parity with Noise

While LP is easy to solve, once we add noise to our sample set this becomes a hard problem [Pie12]. Learning Parity with Noise (LPN) does even become impossible to solve with a too small and too noisy set of samples.

The noise rate for LPN is $r \in \mathbb{R}$ with $0 < r < 0.5$. This rate is used for the Bernoulli distribution Ber_r, which returns 1 with a chance of r and 0 else.

The noise itself is called $e \in \mathbb{Z}_2$ and generated from Ber_r. Each sample b_i is modified by its own e, with $b_i =< a_i, s > +e \mod 2$ [Pie12].

Looking at the example above, a noise rate of $r = \frac{1}{3}$ would most likely introduce an error in one of the samples, which, no matter which sample is affected, falsifies the solution. The chance of this issue appearing will be reduced in implementations by introducing more constraints and using error correcting codes.

3.3. Learning With Errors

The step from LPN to LWE consist of expanding the underlying set. While LPN is based on the set $\mathbb{Z}_2$, samples and secrets in LWE can come from the set $\mathbb{Z}_q^n$, with $n, q \in \mathbb{N}$ and $q \geq 2$. The distribution of the errors is also changed to the Gaussian distribution $D_{\mathbb{Z},\sigma}$ over $\mathbb{Z}_q$, denoted the same way as in [Pöp+19]. The parameter σ is used in the weight assigned to all $x \in \mathbb{Z}$ in D, which is proportional to $\exp(\frac{-x^2}{2\sigma^2})$. The errors will be rounded to the closest integer.

To give a better understanding a sample LWE instance is constructed. In this case it is the *search* LWE ($sLWE$) problem as the secret s is searched. If the set $\mathbb{Z}_5^5$ is considered our base set a secret could be $s = \{0, 2, 4, 4, 1\}$.
The searching party will obviously not know the secret, but only the final output matrix, which is constructed in the following.

At least five samples are needed:
$$\left\{ \begin{array}{l} \{2, 3, 3, 4, 1\} \\ \{4, 1, 1, 3, 0\} \\ \{4, 1, 4, 4, 2\} \\ \{4, 4, 3, 4, 2\} \\ \{4, 0, 4, 4, 2\} \end{array} \right\}$$

The error can also be pre generated in this case with a standard deviation of 1.

5 consecutive numbers are randomly chosen from a set of 1000 numbers. The set is generated with random.org but as this is within $\mathbb{Z}$ the errors will be rounded to the next integer.
The errors are: $\{1, \quad -1, \quad 0, \quad 1, \quad 2\}$

Therefore our $sLWE$ instance is:

$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$	b
2	3	3	4	1	1
4	1	1	3	0	2
4	1	4	4	2	1
4	4	3	4	2	4
4	0	4	4	2	2

Solving this with the Gaussian algorithm and sorting the matrix results in:

$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$	b
1	0	0	0	0	2
0	1	0	0	0	4
0	0	1	0	0	4
0	0	0	1	0	2
0	0	0	0	1	0

This implies the secret has been $\{2, \quad 4, \quad 4, \quad 2, \quad 0\}$ which is incorrect. Therefore this instance has too many errors to be solved classically.

3.3.1. LWE and Lattices

The connection between Lattices and LWE is based on the hardness.

Regev [Reg09] showed a quantum worst-case to average-case reduction from GapSVP and SIVP to LWE. This means that if a solution to LWE in the average case is found, the same idea can be adapted and used to solve worst-case instances of GapSVP and SIVP with a quantum computer. These two problems are considered to be hard to solve as after years of research no efficient way to do so has been discovered. Therefore the reduction suggests that LWE is also hard to solve, because otherwise an efficient algorithm could be found.

Peikert [Pei09] added on to these results and showed that for large moduli $q \geq 2^{n/2}$ worst-case GapSVP can even be classically reduced to LWE with a probabilistic polynomial time reduction.

3.3.2. Decisional LWE

An equivalent problem to $sLWE$ is the decisional LWE ($dLWE$).

The $dLWE_{n,m,q,\chi}$ with $m \geq n$ is defined as follows:
Given m samples $\mod q$, with n unknowns, an error distribution χ and constructed with a random secret $s \in \mathbb{Z}_q^n$ decide if either *all* samples are actual samples of the form $(a, b =< s, a > +e \mod q)$ with $e \xleftarrow{\$} \chi$ or if *all* have been randomly chosen from the uniform distribution $(a, b) \xleftarrow{\$} U(\mathbb{Z}^n \times \mathbb{Z}_q)$ [cf. Pöp+19, p. 5].

An additional property of $sLWE$ and $dLWE$ is, as stated by Pöppelmann et al. (2019), that a solver for either of them can be modified to solve the other as well. Therefore their hardness is equivalent.

3.3.3. Decisional Ring LWE

Used in the following systems is a variant of the $dLWE$ over Rings, the *decisional Ring LWE* ($dRLWEm, q, \chi$).

As clarification $Z[X]$ is the set of all polynomials $a_0 + a_1 * X + a_2 * X^2 + ... + a_j * X^j$ where

$$a_i = \begin{cases} \in Z, & \text{if } i \leq j \\ 0, & \text{if } i > j \end{cases}$$

This means $Z[X]$ is the set of all polynomials with X and all exponents in $\mathbb{N}$ that can possibly constructed with factors in Z.

For this problem R (not to be confused with $\mathbb{R}$) will be the ring $Z[X]/(X^n + 1)$, which means all polynomials modulo $X^n + 1$. Therefore no polynomial will have an X^j with $j > n$.

Additionally R_q is defined as R/qR which means all factors in R (the a_i's) will be modulo q.

n will also be limited to powers of 2 for simplicity.

The $dRLWEm, q, \chi$ asks when given $m \geq 1$ samples if either all of them are proper samples or all are sampled from the uniform distribution $(a, b) \xleftarrow{\$} U(R_q \times R_q)$. A proper sample in this case is for a secret s, uniformly randomly chosen in $U(R_q)$ and shared by all samples, a sample of the form $(a, b = a * s + e \mod q)$ where e is sampled from the distribution χ for each sample [Pöp+19].

4. NewHope

The first cryptosystem that is analysed is NewHope as presented by Pöppelmann et al. (2019) in [Pöp+19]. It consists of two *Key Encapsulation Mechanisms* (KEM) namely NewHope-CPA-KEM and NewHope-CCA-KEM. The hardness is based on the assumed hardness of RLWE. Both KEMs are based on the NewHope-CPA Public Key Encryption (PKE) scheme which will be presented first, followed by the transformation to the KEMs.

4.1. CPA and CCA

To get an understanding of the security, CPA and CCA are briefly explained. In all following scenarios, even outside this section, up to three different imaginary persons are used to simulate communication. Alice, Bob and Eve. Alice and Bob try to communicate in a secure way while Eve in our case tries to read the messages send. For an explanation of CPA and CCA only Alice and Eve are needed.

A system is CPA secure if the following task can not be solved with a probability not-negligible higher than 50%.

Alice initiates her encryption protocol e.g. she initializes her keys. Eve who eventually wants to crack this system may now send any message to Alice and receives in return the encrypted message. Eve uses this information to try and learn about the encryption used. Once Eve wants to try the task she generates two messages which have not been used yet and sends them to Alice. Alice randomly decides which message to encrypt and returns the cypher to Eve. Eve now has to tell which of the two messages Alice encrypted, without asking Alice for further cyphertexts. As random guessing yields a chance of 50% Eve must answer not-negligible more than 50% of the instance correctly to break the system [ADR02]. Any public-key system must be CPA secure as in these cases Eve does not even have to ask Alice to encrypt a message but can do so herself with the public-key.

A CCA secure system is more secure than a CPA secure system, as it adds onto the CPA task. To prove CCA security a similar task as above must not be solved with a probability non-negligible higher than 50%. CCA security is split into two levels. CCA1, also called lunchtime-attack and CCA2.

For CCA1 Eve gains the ability during her learning phase to additionally send any Ciphertext to Alice, to which Alice replies with the deciphered message. There-

fore Eve can not only try and find a way to deduce which message causes which cyphertext, but also which cyphertext may be caused by which message. Eve eventually has to answer the same question as above.

For CCA2 Eve does gain the same ability as in CCA1 but additionally the ability to also ask Alice to decipher any cyphertext after Eve received the cyphertext to solve. There obviously has be the limitation that Eve may not ask Alice to decipher the cyphertext which she received as part of her task, but any variation of it may be deciphered by Alice [CS98][RS92]. Therefore CCA2 even implies that similar cyphertexts not originate from similar messages.

NewHope-CCA-KEM is only proven to be CCA1 secure, as shown later.

4.2. NewHope-CPA-PKE

4.2.1. Preliminaries

Before introducing the actual algorithms a few basics have to be specified and defined.

The security parameters n, q, γ are specified for different security levels and are globally readable by methods. Some methods require certain constraints for the parameters, as e.g. $q < 2^{15}$. The authors of [Pöp+19] set them to $q = 12289, k = 8$ and $n = 512$ or $n = 1024$. Other parameters fitting the constraints, may be choose, however the algorithms are optimized for these values.

The 32-byte seed used in key generation should be sampled from an unpredictable true random number generator. In the implementation used for testing a pseudo-random number generator, which is based on AES256, is used. It is initialized with the c function *rand()* which in turn is initialized with the timestamp. Further information about AES can be found in [Dwo+01].

Furthermore *SHAKE256*, *SHAKE128* and related functions as specified in [ST15] are used throughout the implementation as hard hash functions. *SHAKE256* takes two arguments, l and d. l is the amount of output bytes which should be generated and d is a byte array used as initial seed. *SHAKE128* works in a similar way. It can act as a pseudo random number generator.

$b2i(x)$ will be used to describe the transformation of the byte x into an integer.

The definitions are:

Definition 4.2.1. The Hamming Weight of a byte x is the amount of bits set to 1.

$$HW(x)$$

Definition 4.2.2. The bit-reversal of an unsigned integer v and a power of two n is defined as:

$$BitRev(v) = \sum_{i=0}^{\log_2(n)-1}(((v \gg i)\&1) \ll (\log_2(n) - 1 - i))$$

Definition 4.2.3. The bit-reversal of a whole polynomial s is defined as:

$$PolyBitRev(s) = \sum_{i=0}^{n-1} s_i X^{BitRev(i)}$$

Definition 4.2.4. The *Number Theoretic Transform* (NTT) is used to transform a polynomial to another domain. The variable ω is an n-th primitive root of unity and $\gamma = \sqrt{\omega} \mod q$.

$$NTT(g) = \hat{g} = \sum_{i=0}^{n-1} \hat{g}_i X^i$$

$$\hat{g}_i = \sum_{j=0}^{n-1} \gamma^j g_j \omega^{ij} \mod q$$

Definition 4.2.5. The inverse of the NTT is used to return a transformed polynomial to the original domain.

$$NTT^{-1}(\hat{g}) = g = \sum_{i=0}^{n-1} g_i X^i$$

$$g_i = (n^{-1}\gamma^{-1} \sum_{j=0}^{n-1} \hat{g}_j \omega^{-ij}) \mod q$$

The NTT is useful as it allows an easy multiplication of two transformed polynomials. In the standard domain a multiplication of polynomials requires multiplication of each element of each polynomial with each element of the other.
To multiply two transformed polynomials only coefficient-wise multiplication has to be performed. The resulting polynomial can be transformed to the original domain and will be the same as if the multiplication was performed without the transform.

Therefore the multiplication of two polynomials $a, b \in R_q = \mathbb{Z}_q[X]/(X^n + 1)$ with the result $c \in R_q$ can be written as $c = NTT^{-1}(NTT(a) \circ NTT(b))$ where $\circ$ is used as operator for coefficient wise multiplication [Pöp+19].

NTT usually requires bit-reversal operations which in this case are not part of the definition but rather executed separately as they may be omitted when transforming random noise.

Using this transformation reduces the amount of operations used significantly but as trade-off has limits on the parameter choices. There for example exists a relation between n and q [BCA04]. The parameters chosen in NewHope fit the requirements though.

Further reading on NTT can be done in "Some historical notes on number theoretic transform" [BCA04].

4.2.2. Algorithms

The most basic algorithm used is $SAMPLE$, which deterministically samples polynomials in R_q from ψ_8^n.

Algorithm 1 Deterministic sampling of polynomials in R_q from ψ_8^n [Pöp+19, p. 8]

1: **function** SAMPLE($seed \in \{0, ..., 255\}^{32}$, positive integer $nonce$)
2: r $\leftarrow R_q$
3: $extseed \leftarrow \{0, ..., 255\}^{34}$
4: $extseed[0:31] \leftarrow seed[0:31]$
5: $extseed[32] \leftarrow nonce$
6: **for** i **from** 0 **to** $n/64 - 1$ **do**
7: $extseed[33] \leftarrow i$
8: $buf \leftarrow$ SHAKE256$(128, extseed)$
9: **for** j **from** 0 **to** 63 **do**
10: $a \leftarrow buf[2 * j]$
11: $b \leftarrow buf[2 * j + 1]$
12: $r_{64*i+j} = \text{HW}(a) + q - \text{HW}(b) \mod q$
13: **return b** $\in R_q$

Sample is the centered binomial distribution ϕ_k with $k = 8$. k is not a parameter and is always 8. $SAMPLE$ has a mean of 0, variance $k/2 = 4$ and a standard deviation of $\zeta = \sqrt{8/2}$ [Pöp+19].

It works by initially taking a random 32-byte seed and a one-byte nonce. The nonce is added to the seed, so the same seed can be used multiple times with different nonces. First r is initialized with a n long 0-polynomial and $extseed$ (as in extended seed) gets assigned with 34 memory bytes whose value may be anything. Afterwards the seed and the nonce get put into $extseed$. The following for-Loop can be imagined as splitting n into intervals of the size 64. This is the first case which requires n to be a power of 2 and to be greater or equal to 64. Once in the loop the last byte of $extseed$ is always set to the current index. This gives a unique seed per nonce and iteration to hash with SHAKE256. It additionally takes the number 128 as argument, which tells it to output 128 bytes. Having received the hash the next loop runs exactly 64-times which is half the size of the hash. Per loop the next two bytes of the hash are being looked at. The Hamming Weight of each is calculated and used together with q to calculated the coefficient of r at this point.

Note: $-HW(b) \equiv q - HW(b) \mod q$, however the latter implementation prevents underflow.

$SAMPLE$ eventually returns a random r $\in R_q$.

GenA expands a seed to an NTT transformed polynomial. First $\hat{a}$ is initialized as already NTT transformed polynomial. Then *extseed* seed is initialized and set to the given seed, except the 33rd byte. The following for-Loop splits n into 64-byte intervals again. The index of the current iteration is used as 33rd seed byte. Using Shake128Absorb a 200 byte state is generated from the seed. Looping 64 times for the current interval SHAKE128Squeeze is called which takes a number j and a state and returns a $168 * j$ long byte-array buf and a new state. The following loop is used to iterate through buf. Again two bytes are being looked at at the same time. The first operation is to prepend the second byte to the first one and save the result as integer in val. If $val < 5 * q$ it can be used and assigned to the current coefficient of $\hat{a}$. Afterwards a value for the next coefficient in $\hat{a}$ can be searched. If val does not fullfill the condition it is simply discarded. In either case while there are unassigned coefficients in $\hat{a}$ left move to the next two bytes in buf or if none are left or the current interval in n is finished generate a new buf.

Eventually return a NTT transformed polynomial with all coefficients randomly set to values $< 5 * q$.

Algorithm 2 Deterministic generation of $\hat{a}$ by expansion of a seed [Pöp+19, p. 10]

function GenA($seed \in \{0, ..., 255\}^{32}$)
  $\hat{a} \leftarrow R_q$
  $extseed \leftarrow \{0, ..., 255\}^{33}$
  $extseed[0:31] \leftarrow seed[0:31]$
  for i **from** 0 **to** $n/64 - 1$ **do**
    $ctr \leftarrow 0$
    $extseed[32] \leftarrow i$
    $state \leftarrow$ SHAKE128Absorb($extseed$)
    while $ctr < 64$ **do**
      $buf, state \leftarrow$ SHAKE128Squeeze($1, state$)
      $j \leftarrow 0$
      for $j < 168$ **and** $ctr < 64$ **do**
        $val \leftarrow$ b2i($buf[j]$)|(b2i($buf[j + 1]$)$\ll 8$)
        if $val < 5 * q$ **then**
          $\hat{a}_{i*64+ctr} \leftarrow val$
          $ctr \leftarrow ctr + 1$
        $j \leftarrow j + 2$
  return $\hat{a} \in R_q$

Pöppelmann et al. (2019) specify a polynomial encoding and a decoding algorithm, which only works for $q < 2^{14}$.

EncodePolynomial in this case has an NTT transformed polynomial of the length n as input. It then creates a new byte-array where for every 4 integers in the polynomial, 7 bytes are added to the output array. Each integer is represented by 4 bytes. The total $4 * 4 = 16$ bytes of the integers can be compressed into 7, as every integer is mod q and therefore has a maximum of 14 bit.

$$7 \text{ byte} = 7 * 8 \text{ bit} = 56 \text{ bit} = 4 * 14 \text{ bit}.$$

This means it outputs a byte array of the length $7 * n/4$.

Algorithm 3 Encoding a polynomial in R_q to a byte array [Pöp+19, p. 11]

function EncodePolynomial($\hat{s}$)
  $r \leftarrow \{0, ..., 255\}^{7*n/4}$
  for i **from** 0 **to** $n/4 - 1$ **do**
    $t0 \leftarrow \hat{s}_{4*i+0} \mod q$
    $t1 \leftarrow \hat{s}_{4*i+1} \mod q$
    $t2 \leftarrow \hat{s}_{4*i+2} \mod q$
    $t3 \leftarrow \hat{s}_{4*i+3} \mod q$
    $r[7 * i + 0] \leftarrow t0\&0\text{xff}$
    $r[7 * i + 1] \leftarrow (t0 \gg 8)|(t1 \ll 6)\&0\text{xff}$
    $r[7 * i + 2] \leftarrow (t1 \gg 2)\&0\text{xff}$
    $r[7 * i + 3] \leftarrow (t1 \gg 10)|(t2 \ll 4)\&0\text{xff}$
    $r[7 * i + 4] \leftarrow (t2 \gg 4)\&0\text{xff}$
    $r[7 * i + 5] \leftarrow (t2 \gg 12)|(t3 \ll 2)\&0\text{xff}$
    $r[7 * i + 6] \leftarrow (t3 \gg 6)\&0\text{xff}$
  return $r \in \{0, ..., 255\}^{7*n/4}$

DecodePolynomial reverses the encoding and takes a byte array of length $7 * n/4$ as input and outputs a NTT transformed polynomial of length n.

Algorithm 4 Decoding of a polynomial represented as a byte array into an element in R_q [Pöp+19, p. 11]

function DecodePolynomial($v \in \{0, ..., 255\}^{7*n/4}$)
  for i **from** 0 **to** $n/4 - 1$ **do**
    $r \leftarrow R_q$
    $r_{4*i+0} \leftarrow \text{b2i}(v[7 * i + 0])||((\text{b2i}(v[7 * i + 1])\&0\text{x3f}) \ll 8)$
    $r_{4*i+1} \leftarrow (\text{b2i}(v[7 * i + 1]) \gg 6)||(\text{b2i}(v[7 * i + 2]) \ll 2)||$
      $(\text{b2i}(v[7 * i + 3]\&0\text{x0f}) \ll 10)$
    $r_{4*i+2} \leftarrow (\text{b2i}(v[7 * i + 3]) \gg 4)||(\text{b2i}(v[7 * i + 4]) \ll 4)||$
      $(\text{b2i}(v[7 * i + 3]\&0\text{x03}) \ll 12)$
    $r_{4*i+3} \leftarrow (\text{b2i}(v[7 * i + 5]) \gg 2)||(\text{b2i}(v[7 * i + 6]) \ll 6)$
  return $r \in R_q$

EncodePK takes the public key, so a polynomial and the public seed, as input. It transforms the polynomial into a byte array, appends the public seed and returns the resulting array, which has a length of $7 * n/4 + 32$.

Algorithm 5 Encoding of the public key [Pöp+19, p. 11]

function EncodePK($\hat{b} \in R_q, publicseed \in \{0, ..., 255\}^{32}$)
  $r \leftarrow \{0, ..., 255\}^{7*n/4+32}$
  $r[0:7 * n/4 - 1] \leftarrow$ EncodePolynomial($\hat{b}$)
  $r[7 * n/4:7 * n/4 + 31] \leftarrow publicseed[0:31]$
  return $r \in \{0, ..., 255\}^{7*n/4+32}$

DecodePK takes a byte array of the length $7 * n/4 + 32$ as input. This array consists of $7 * n/4$ bytes which represent an encoded polynomial, while the last 32 represent the public seed. This means the decode function splits off the first $7 * n/4$ bytes and runs them through DecodePolynomial. It eventually returns the combination of the decoded polynomial and the last 32 bytes of the entered array, namely the public seed.

Algorithm 6 Decoding of the public key [Pöp+19, p. 11]

function DecodePK($pk \in \{0, ..., 255\}^{7*n/4+32}$)
  $\hat{b} \leftarrow$DecodePolynomail($pk[0:7 * n/4 - 1]$)
  $seed \leftarrow pk[7 * n/4:7 * n/4 + 31]$
  return $(\hat{b} \in R_q, seed \in \{0, ..., 255\}^{32})$

The third pair of encoding and decoding functions is to encode a binary message as polynomial and decode a polynomial to a message.

Encode takes an array of 32 bytes (the message) as input. It iterates of every byte and for those over each bit. During this loop it assigns each coefficient in a polynomial v a value of 0 if the current bit is 0 or a value of $q/2$ if the bit is 1. It returns a polynomial of length n.

Algorithm 7 Message encoding [c. Pöp+19, p. 12]

function ENCODE($\mu \in 0, ..., 255^{32}$)
  $v \leftarrow R_q$
  for i **from** 0 **to** 31 **do**
    for j **from** 0 **to** 7 **do**
      $mask \leftarrow -((\mu[i] \gg j)\&1)$
      $v_{8*i+j+0} \leftarrow mask\&(q/2)$
      $v_{8*i+j+256} \leftarrow mask\&(q/2)$
      if n **equals** 1024 **then**
        $v_{8*i+j+512} \leftarrow mask\&(q/2)$
        $v_{8*i+j+768} \leftarrow mask\&(q/2)$
  return $v \in R_q$

Decode reverses the encoding and takes a polynomial of length n as input.

The function iterates through all 255 encoded bits and decodes each. If $n = 512$ and the sum of all coefficients for one bit is bigger than $\lfloor q/2 \rfloor$ the bit is 0, otherwise it is 1. The threshold for $n = 1024$ is q.

This functionality is realized by setting t to $(q-1)/2$, if the current encoded bit was 0, or to $\frac{1}{2}$ if it was a 1 before encoding.

The following operations can be summarized as $t = t*2$ for $n = 512$ and $t = t*4$ for $n = 1024$.

In the case that $n = 512$, $q/2$ will be subtracted from t. Therefore $t = q/2-1 > 0$ if the old bit was 0, in contrary if it was 1: $t = -q/2 + 1 < 0$.

If $n = 1024$, q will be subtracted from t, resulting in $q - 2 > 0$ if the old bit was 0 and $-q + 2 < 0$ if the old bit was a 1.

The function now exploits how c implements negative numbers and extracts the first bit of t indicating if the number is negative. Due to t being always negative when the old bit was 1, only the indicating bit has to be extracted from t and inserted at the appropriate place in the array.

Noteworthy is that t is a relatively larger number and therefore compensates error very well.

The insertion happens by shifting the index down by 3 to only switch the element of the array every 8th step, as 8 bits are inserted per byte. By running a bit-wise AND of the index and 7 only the last 3 bytes and therefore only the numbers 0 to 7 are extracted to shift t to the appropriate position in the current byte. The resulting t is added to the existing bit with a bit-wise OR as this allows a fast merging of the 1s of both bytes.

Algorithm 8 Message decoding [c. Pöp+19, p. 12]

function DECODE($v \in R_q$)
  $\mu \leftarrow \{0, ..., 255\}^{32}$
  for i **from** 0 **to** 255 **do**
    $t \leftarrow |(v_{i+0} \mod q) - (q-1)/2|$
    $t \leftarrow t + |(v_{i+256} \mod q) - (q-1)/2|$
    if n **equals** 1024 **then**
      $t \leftarrow t + |(v_{i+512} \mod q) - (q-1)/2|$
      $t \leftarrow t + |(v_{i+768} \mod q) - (q-1)/2|$
      $t \leftarrow (t - q)$
    else
      $t \leftarrow (t - q/2)$
    $t \leftarrow t \gg 15$
    $\mu[i \gg 3] \leftarrow \mu[i \gg 3] | (t \ll (i \& 7))$
  return $\mu \in \{0, ..., 255\}^{32}$

Another group of functions is cyphertext compression, decompression, encoding and decoding. These are used for reducing the size of the cyphertext and compressing a polynomial and the cyphertext together.

Compress takes a polynomial in R_q as input and compresses it to $\frac{3}{8}$ of his original byte size. The method iterates through groups of 8 bytes. These bytes are each compressed to 3 bits and all concatenated to each other. This results in 3 compressed bytes per 8 original bytes, as 8bytes$-> 8 * 3\text{bit} = 24\text{bit} = 3\text{byte}$. Overall this function returns a byte array of the length $3 * n/8 = \frac{3}{8}n$.

Algorithm 9 Ciphertext compression [Pöp+19, p. 13]

function COMPRESS($v' \in R_q$)
  $k \leftarrow 0$
  $t \leftarrow \{0, ..., 255\}^8$
  $h \leftarrow \{0, ..., 255\}^{3*n/8}$
  for l **from** **to** $n/8 - 1$ **do**
    $i \leftarrow 8 * l$
    for j **from** 0 **to** 7 **do**
      $t[j] \leftarrow v'_{i+j} \mod q$
      $t[j] \leftarrow ((\text{b2i}(t[j] \ll 3) + q/2)/q) \& 7$
    $h[k+0] \leftarrow t[0] | (t[1] \ll 3) | (t[2] \ll 6)$
    $h[k+1] \leftarrow (t[2] \gg 2) | (t[3] \ll 1) | (t[4] \ll 4) | (t[5] \ll 7)$
    $h[k+2] \leftarrow (t[5] \gg 1) | (t[6] \ll 2) | (t[7] \ll 5)$
    $k \leftarrow k + 3$
  return $h \in \{0, ..., 255\}^{3*n/8}$

Decompress reverses Compress. It takes a byte array of the length $3 * n/8$ as input and expands the now compressed bytes back to the originals. As 3 compressed byte represent 8 uncompressed bytes, the function iterates over the compressed bytes in groups of 3. The set of bytes which is currently selected is split into 8 parts of 3 bit each. These 3 bits are then each expanded to a full byte in a sub-loop. The function eventually returns a polynomial in R_q.

Algorithm 10 Ciphertext decompression [c. Pöp+19, p. 13]

function DECOMPRESS($h \in \{0, ..., 255\}^{3*n/8}$)
  $k \leftarrow 0$
  $r \leftarrow R_q$
  for l **from** **to** $n/8 - 1$ **do**
    $i \leftarrow 8 * l$
    $r_{i+0} \leftarrow h[k + 0]\&7$
    $r_{i+0} \leftarrow (h[k + 0] \gg 3)\&7$
    $r_{i+0} \leftarrow (h[k + 0] \gg 6)|((h[k + 1] \ll 2)\&4)$
    $r_{i+0} \leftarrow (h[k + 1] \gg 1)\&7$
    $r_{i+0} \leftarrow (h[k + 1] \gg 4)\&7$
    $r_{i+0} \leftarrow (h[k + 1] \gg 7)|((h[k + 2] \ll 1)\&6)$
    $r_{i+0} \leftarrow (h[k + 2] \gg 2)\&7$
    $r_{i+0} \leftarrow (h[k + 0] \gg 5)$
    $k \leftarrow k + 3$
    for j **from** 0 **to** 7 **do**
      $r_{i+j} \leftarrow (r_{i+j} * q + 4) \gg 3$
  return $r \in R_q$

EncodeC takes a polynomial and a compressed cyphertext as input. It encodes the polynomial and appends the encoded cyphertext to the byte array. The merged array is returned.

Algorithm 11 Ciphertext encoding [Pöp+19, p. 13]

function ENCODEC($\hat{u} \in R_q, h \in \{0, ..., 255\}^{3*n/8}$)
  $c[0 : 7 * n/4 - 1] = \text{EncodePolynomial}(\hat{u})$
  $c[7 * n/4 : 7 * n/4 + 3 * n/8 - 1] \leftarrow h$
  return $c \in \{0, ..., 255\}^{7*n/4+3*n/8}$

DecodeC takes a byte array of length $7 * n/4 + 3 * n/8$ as input. It then splits the array before $7 * n/4$ and decodes the polynomial in the first half of the array. It then returns the decoded polynomial and the compressed cyphertext which is the second part of the input byte array.

Algorithm 12 Ciphertext decoding [Pöp+19, p. 13]

function DecodeC($c \in \{0,...,255\}^{7*n/4+3*n/8}$)
  $\hat{u} \leftarrow$ DecodePolynomial($c[0:7*n/4-1]$)
  $h \leftarrow c[7*n/4:7*n/4+3*n/8-1]$
  return $(\hat{u} \in R_q, h \in \{0,...,255\}^{3*n/8})$

All tools needed for the PKE are defined, so the three algorithms for key generation, encryption and decryption are given.

PKE-Gen generates a secret and public key and requires no input variables. The first operation is saving the ideally truly randomly sampled 32 byte seed. Afterwards the seed gets extended to 64 bytes by SHAKE256. The first 32 bytes of the extended seed are the public seed and will be part of the public key, while the second 32 bytes are used as noise to sample the other factors and are required to be kept secret. Now a NTT transformed polynomial is generated from the public seed using the previously defined GenA. Furthermore our secret key s and its NTT transformation $\hat{s}$ are initialized from reversing the output *Sample* produces from the noise seed and 0. The error and its NTT transformation will be created the same way, but using the noise seed and 1 as parameters for *Sample*. Finally all parts come together by multiplying the polynomial generated by the public seed with the secret key (polynomial) and adding the error. The function returns the encoding of the noisy polynomial and the public seed as public key, which is an RLWE instance, as well as $\hat{s}$ as the secret key.

Algorithm 13 NewHope-CPA-PKE Key Generation [Pöp+19, p. 7]

1: **function** NewHope-CPA-PKE.Gen()
2: $seed \xleftarrow{\$} \{0,...,255\}^{32}$
3: $z \leftarrow$ SHAKE256$(64, seed)$
4: $publicseed \leftarrow z[0:31]$
5: $noiseseed \leftarrow z[32:63]$
6: $\hat{a} \leftarrow$ GenA$(publicseed)$
7: $s \leftarrow$ PolyBitRev(Sample$(noiseseed, 0)$)
8: $\hat{s} \leftarrow$ NTT(s)
9: $e \leftarrow$ PolyBitRev(Sample$(noiseseed, 1)$)
10: $\hat{e} \leftarrow$ NTT(e)
11: $\hat{b} \leftarrow \hat{a} \circ \hat{s} + \hat{e}$
12: **return** (pk=EncodePK$(\hat{b}, publicseed)$,sk=EncodePolynomial$(\hat{s})$)

PKE-Encrypt is used to encrypt the message μ. To do so the method requires the public key pk, the message μ and a randomly sampled coin as input. First the public key is decoded. Afterwards $\hat{a}$ is generated from the public key, which results in the same $\hat{a}$ as in the key generation. Now the secret of the sender and two errors are sampled from coin. The NTT transformation $\hat{t}$ of the secret is also ini-

tialized. Following this another RLWE instance is generated by multiplying $\hat{a}$ and $\hat{t}$ and adding the error. Afterwards the message μ is encoded into its polynomial representation. This representation then gets added to the reversed NTT of the product of the RLWE instance of the public key $\hat{b}$ and the new secret $\hat{t}$. Alongside the coded message the error e'' is added as well. The sum is then compressed and encoded alongside the RLWE instance $\hat{u}$ to the cipher c.

Algorithm 14 NewHope-CPA-PKE Encryption [Pöp+19, p. 7]

function NewHope-CPA-PKE.Encrypt($pk \in \{0,...,255\}^{7*n/4+32}, \mu \in \{0,...,255\}^{32}, coin \in \{0,...,255\}^{32}$)
- $(\hat{b}, publicseed) \leftarrow \text{DecodePk}(pk)$
- $\hat{a} \leftarrow \text{GenA}(publicseed)$
- $s' \leftarrow \text{PolyBitRevers}(\text{Sample}(coin, 0)$
- $e' \leftarrow \text{PolyBitRevers}(\text{Sample}(coin, 1)$
- $e'' \leftarrow \text{Sample}(coin.2)$
- $\hat{t} \leftarrow \text{NTT}(s')$
- $\hat{u} \leftarrow \hat{a} \circ \hat{t} + \text{NTT}(e')$
- $v \leftarrow \text{Encode}(\mu)$
- $v' \leftarrow \text{NTT}^{-1}(\hat{b} \circ \hat{t}) + e'' + v$
- $h \leftarrow \text{Compress}(v')$
- **return** $c = \text{EncodeC}(\hat{u}, h)$

PKE-Decrypt is the simples of the PKE functions. It takes the ciphertext c and the secret key sk, matching the public key used to encode c, as input. First the ciphertext is decoded to $\hat{u}$ and h. The secret key is decoded as well. Afterwards the polynomial v' is received by decompressing h. Now the message μ can be extracted from v' by subtracting the NTT inverse of $\hat{u} \circ \hat{s}$.

Algorithm 15 NewHope-CPA-PKE.Decrypt [Pöp+19, p. 7]

function NewHope-CPA-PKE.Decrypt($c \in \{0,...,255\}^{7*n/4+3*n/8}, sk \in 0,...,255^{7*n/4}$)
- $(\hat{u}, h) \leftarrow \text{DecodeC}(c)$
- $\hat{s} \leftarrow \text{DecodePolynomial}(sk)$
- $v' \leftarrow \text{Decompress}(h)$
- $\mu \leftarrow \text{Decode}(v' - NTT^{-1}(\hat{u} \circ \hat{s}))$
- **return** $\mu \in \{0,...,255\}^{32}$

The encrypting and decrypting works, as:

$$
\begin{aligned}
& \mu = \text{Decode}(v' - \text{NTT}^{-1}(\hat{u} \circ \hat{s})) && \\
\Leftrightarrow \quad & \mu = \text{Decode}((\text{NTT}^{-1}(\hat{b} \circ \hat{t}) + e'' + v) - \text{NTT}^{-1}(\hat{u} \circ \hat{s})) && \text{Replace } v' \\
\Leftrightarrow \quad & \mu = \text{Decode}((\text{NTT}^{-1}((\hat{a} \circ \hat{s} + \hat{e}) \circ \hat{t}) + e'' + v) - \text{NTT}^{-1}(\hat{u} \circ \hat{s})) && \text{Replace } \hat{b} \\
\Leftrightarrow \quad & \mu = \text{Decode}((\text{NTT}^{-1}((\hat{a} \circ \hat{s} + \hat{e}) \circ \hat{t}) + e'' + v) - \text{NTT}^{-1}((\hat{a} \circ \hat{t} + \text{NTT}(e')) \circ \hat{s})) && \text{Replace } \hat{u} \\
\Leftrightarrow \quad & \mu = \text{Decode}((a \circ s + e) \circ s' + e'' + v - (a \circ s' + e') \circ s)) && \text{Calculate NTT}^{-1} \\
\Leftrightarrow \quad & \mu = \text{Decode}(a \circ s \circ s' + e \circ s' + e'' + v - a \circ s' \circ s - e' \circ s)) && \text{Resolve the most inner brackets} \\
\Leftrightarrow \quad & \mu = \text{Decode}(e \circ s' + e'' + v - e' \circ s)) && \text{Remove } a \circ s \circ s' \\
\Leftrightarrow \quad & \mu = \text{Decode}(v + e'' + e \circ s' - e' \circ s)) && \text{Reorder} \\
\Leftrightarrow \quad & \mu = \text{Decode}(\text{Encode}(\mu) + e'' + e \circ s' - e' \circ s)) && \text{Replace } v \\
\Leftrightarrow \quad & \mu = \mu && \text{Decode reverts Encode, while being able to correct for errors.}
\end{aligned}
$$

4.3. NewHope-CPA-KEM

The PKE system can easily be expanded to a *Key Encapsulation Method* (KEM), which allows Alice and Bob to switch to a faster symmetric key encryption scheme after the key exchange.

CPA-KEM-Gen is not different from the PKE-Gen.

Algorithm 16 NewHope-CPA-KEM Key Generation

function NewHope-CPA-KEM.Gen()
  $(pk, sk) \xleftarrow{\$}$ NewHope-CPA-PKE.Gen()
  return (pk, sk)

CPA-KEM-Encaps is a small expansion of PKE-Encrypt, as it at first randomly samples a coin and then generates an new coin $coin'$ and the seed K for the shared secret ss out of it using SHAKE256. The seed is then encrypted with PKE-Encrypt which receives a public key,the seed K and $coin'$. KEM-Encrypt

eventually outputs the encrypted seed and the actual shared secret, which is the result of hashing the seed.

Algorithm 17 NewHope-CPA-KEM Encapsulation [Pöp+19, p. 14]

function NEWHOPE-CPA-KEM.ENCAPS(pk)
  $coin \xleftarrow{\$} \{0, ..., 255\}^{32}$
  $K||coin' \leftarrow \text{SHAKE256}(64, coin) \in \{0, ..., 255\}^{32+32}$
  $c \leftarrow \text{NewHope-CPA-PKE.Encrypt}(pk; K; coin')$
  $ss \leftarrow \text{SHAKE256}(32, K)$
  return (c, ss)

CPA-KEM-Decaps is also not much different from PKE-Decrypt. It receives the cipher and the private/secret key and delegates decryption to PKE-Decrypt. It then returns the hash of the secret seed the decryption returns.

Algorithm 18 NewHope-CPA-KEM Decapsulation [Pöp+19, p. 14]

function NEWHOPE-CPA-KEM.DECAPS(c, s)
  $K' \leftarrow \text{NewHope-CPA-PKE.Decrypt}(c, sk)$
  return $ss = \text{SHAKE256}(32, K')$

4.4. NewHope-CCA-KEM

Increasing the security level even further a CCA-KEM can be constructed from the CPA-PKE.

The security proof for the correct expansion of the model is much more complicated and many systems rely on the *Fujisaki-Okamoto* (FO) transform [FO99], which allows construction of a IND-CCA2 secure PKE system from a one-way secure PKE system, while keeping a tight security prove. This transform has been, in a slight variation, proven to be secure even in the quantum random oracle mode [TU16] and can therefore not be broken by quantum computers, if the underlying PKE is secure.

The Fujiskai Okamoto transform and the quantum proof are, however only applicable to error free systems, which RLWE obviously is not. Therefore the NewHope authors rely on Hofheinz et al. (2017) [HHK17], which constructed a modular transformation, to allow a construction of an IND-CCA secure KEM from an imperfect IND-CPA secure PKE scheme. They give a definition of QFO_m^y, which has been adapted by Pöppelmann et al. (2019).

QFO$_m^y$ requires a PKE with the methods KeyGen, Encrypt and Decrypt. Furthermore M is defined as the set of all possible messages, while len_s, len_K, len_d and len_{ss} are parameters. $G : \{0, ..., 255\}^* \rightarrow \{0, ..., 255\}^{len_K} \times R^E \times \{0, ..., 255\}^{len_d}$ and $F : \{0, ..., 255\}^* \rightarrow \{0, ..., 255\}^{len_{ss}}$ are hash functions.

Based on QFO_m^y, QKEM_m^y is defined depending on PKE, G and F, therefore QKEM_m^y=QFO_m^y[PKE,G,F] [Pöp+19] and with the following three functions.

QKEM_m^y.KeyGen replaces the secret key with a group of the secret key, the private key and a random byte array with the length len_s.

Algorithm 19 QKEM_m^y Key Generation [c. Pöp+19, p. 15]

function QKEM_m^y.KeyGen()
$(pk, sk) \xleftarrow{\$} \text{PKE.KeyGen}()$
$s \xleftarrow{\$} \{0, ..., 255\}^{len_s}$
$\overline{sk} \leftarrow (sk, pk, s)$
return $(pk, \overline{sk})$

QKEM_m^y.Encaps takes the public key as input to encrypt the shared secret. It first generates a random message and hashes it together with the public key. This generates a random $K, coin'$ and d.The cyphertext c consists of the random message which acted a seed. The shared secret is hashed from the K, the cyphertext and d. c will then be replaced by a group of the cyphertext and d. c and the shared secret are returned.

Algorithm 20 QKEM_m^y Encapsulation [c. Pöp+19, p. 15]

function QKEM_m^y.Encaps(pk)
$\mu \xleftarrow{\$} M$
$(K, coin', d) \leftarrow G(pk||\mu)$
$c \leftarrow \text{PKE.Encrypt}(pk, \mu; coin')$
$ss \leftarrow F(K||c||d)$
$\bar{c} \leftarrow (c, d)$
return $(\bar{c}, ss)$

QKEM_m^y.Decaps is able to decrypt μ from the ciphertext saved as part of c. It now generates the hash of the public key together with μ. This should result in the same $K, coin$ and d. It then compares the original ciphertext and d with a newly encrypted cyphertext and the newly generated d'. If the compare returns true, the message was valid and can be used as shared secret. If they do not match an error occurred or was forced and the shared secret will be generated with the usage of s which has not been used yet, but was sampled randomly. Therefore malformed cyphertext are recognized.

Algorithm 21 QKEM_m^y Decapsulation [c. Pöp+19, p. 15]

function $\text{QKEM}_m^y.\textsc{Decaps}((c, d), (sk, pk, s))$
  $\mu' \leftarrow \text{PKE.Decrypt}(c, sk)$
  $(K', coin'', d') \leftarrow G(pk||\mu')$
  if $c = \text{PKE.Encrypt}(pk, \mu'; coin'')$ **and** $d = d'$ **then**
    return $ss' \leftarrow F(K'||c||d)$
  else
    return $ss' \leftarrow F(s||c||d)$

The actual implementation of NewHope-CCA-KEM is similar to the theoretical algorithms, but the authors added the hash of the public key as 4th value to the secret key. They also added nested hashing, when hashing more than one value, by hashing the second value first before hashing the hash with the first value.

As Hash function SHAKE256 is used for G and F, while all *len* variables are set to 32.

Therefore the actual algorithms used in the implementation are:

Algorithm 22 NewHope-CCA-KEM Key Generation [Pöp+19, p. 16]

function NewHope-CCA-KEM.Gen()
  $(pk, sk) \xleftarrow{\$} \text{NewHope-CPA-PKE.Gen}()$
  $s \xleftarrow{\$} \{0, ..., 255\}^{32}$
  return $(pk, \overline{sk} = (sk||pk||\text{SHAKE256}(32, pk)||s))$

Algorithm 23 NewHope-CCA-KEM Encapsulation [Pöp+19, p. 16]

function NewHope-CCA-KEM.Encaps(pk)
  $coin \xleftarrow{\$} \{0, ..., 255\}^{32}$
  $\mu \xleftarrow{\$} \text{SHAKE256}(32, coin) \in \{0, ..., 255\}^{32}$
  $K||coin'||d \leftarrow \text{SHAKE256}(96, \mu||\text{SHAKE256}(32, pk)) \in \{0, ..., 255\}^{32+32+32}$
  $c \leftarrow \text{NewHope-CPA-PKE.Encrypt}(pk, \mu; coin')$
  $ss \leftarrow \text{SHAKE256}(32, K||\text{SHAKE256}(32, c||d))$
  return $(\bar{c} = c||d, ss)$

Algorithm 24 NewHope-CCA-KEM Decapsulation [Pöp+19, p. 16]

function NewHope-CCA-KEM.Decaps($\bar{c}, \overline{sk}$)
$c||d \leftarrow \bar{c} \in \{0,...,255\}^{3*n/8+7*n/4+32}$
$sk||pk||h||s \leftarrow \overline{sk} \in \{0,...,255\}^{7*n/4+7*n/4+32+32+32}$
$\mu' \leftarrow$ NewHope-CPA-PKE.Decrypt(c, sk)
$K'||coin''||d' \leftarrow$ SHAKE256($\mu'||h$) $\in \{0,...,255\}^{32+32+32}$
if $c =$ NewHope-CPA-PKE.Encrypt($pk, \mu'; coin''$) **and** $d = d'$ **then**
$fail \leftarrow 0$
else
$fail \leftarrow 1$
$K_0 \leftarrow K'$
$K_1 \leftarrow s$
return $ss =$ SHAKE256($32, K_{fail}||$SHAKE256$(32, c||d)$)

Minor implementation tweaks, like the Montgomery reduction will not be specified, but can be found at [Pöp+19, p. 22].

This concludes the algorithm specification as every required algorithm for a IND-CCA-KEM has been defined.

4.5. Runtime Analysis of New Hope

New Hope specifies CPU cycles used for all 4 algorithm implementations, however this analysis is looking at the actual instructions run during the execution. The strongest algorithm with the most operations NH-1024-CCA-KEM has been tested for its assembly operation usage to determine the most important instructions, hardware should be able to execute efficiently to work optimally with New Hope.

First a suitable test instance has to be created. Therefore testNewHope.c (see Appendix A.1) has been created.

In testNewHope.c the relevant steps from the provided PQCgenKAT_kem.c have been adapted. It first initializes the c randomiser with the time stamp and then initializes the internal randomiser with the c randomiser. It then generates a Key-Pair, encrypts the shared secret and decrypts the shared secret. All three steps are split with 'breakFunction()', which only serves the purpose to be recognized as break point when analysing the code.

To determine which assembly operations are used at runtime the GNU De-Bugger (GDB) can be used. With the GDB the program can be stopped at break-points at any point during the execution and examined. Therefore breakpoints are set to 'crypto_kem_keypair', 'crypto_kem_enc' and 'crypto_kem_dec'.

GDB also gives the ability to not continue the program as normal after a break point, but to only move exactly one operation ahead, which will be called moving a step. Steps can be split into two kinds, first where GDB steps ahead, but over

any sub functions that are called, these are then evaluated immediately as one step, this is the default *step* operation. The second kind is called 'step into' and the matching GDB command is *stepi*. In this case GDB does also step into each function that is called and will execute the called function step by step as well. In these tests only *stepi* is used.

After performing a step or when arriving at a breakpoint GDB prints the line number of the compiled program at which it currently is.

To retrieve the assembly operation, which was just executed the line number can be extracted and given to the GDB command *disas*. This command takes two line numbers separated by a comma as input and returns the disassembled code beginning with the first specified line up to one before the second line. So 'disas line1, line2' would disassemble the interval $[line1, line2)$.

To automate this process and allow easy multi-threading a simple Java program has been written which makes use of the easy cross thread accessibility and the Java ProcessBuilder. It can be found at appendix A.2.

This implementation has been run on an *AMD Opteron Processor 33280* at 2550 MHz with Ubuntu 18.04.3 LTS, gdb 8.1.0.20180409 and gcc 7.4.0.

It generated the output files found in the directory assemblyOps, summarized in Appendix A.3, A.4 and A.5

The Java implementation produced the mentioned files in 100 iterations for the optimized implementation.

As Pöppelmann et al. (2019) mentioned there is no difference between the optimized and the reference implementation and thus these results are represent both implementations.

4.5.1. Results

The tables for all three functions rank the most instructions in a similar order with just a close by instructions swapping places. The results mainly show the expected operations at the top.

According to Huang and Peng (2002) [HP02, 12f.] the most frequent operation is *mov* with a frequency of 12.5%. If all similar move operations are summarized they add up to a frequency of 25.7%. The move instruction is the most frequent operation in this analysis as well, however with a much higher frequency of nearly 40%. Not considering that the related instruction *movzbl* and *movzwl* are also in the top 5 most frequent instructions for all methods. This shows that the move instruction is heavily used. It therefore is the first and most obvious operation, hardware running this encryption should perform fast. Improving the execution time of move instructions can speed up the encryption significantly. Huang and

Peng (2002) showed that in 2002 the *mov* instruction could theoretically be improved by reducing the amount of micro operation cycles needed by 11% and the amount of integers loaded by 20%. This leaves room for speculation if another speed-up can be found or specifically designed for this KEM. Without speculation it can be said that choosing hardware with a fast implementation of move is useful for running New Hope.

The second most frequent instruction is *add* with a bit less than 10%. In Huang and Peng (2002) *add* only ranked fourth with 5.1%. New Hope may differ from the average case due to the frequent addition of error to the polynomials. This makes the add instruction another criteria to choose good hardware.

Another instruction that is used frequently, but not in the top 65 instructions in Huang and Peng (2002), is *lea*. However *les* a similar instruction is present. These two operations may have been swapped due to the usage of *LD_BIND_NOW=1* or newer implementations. Loading is always an important operation and therefore no surprise to be found in the top frequent functions in these test or [HP02], although in different variants. Nevertheless it is the third instruction or rather instruction group to consider when choosing hardware.

Two more important instructions are *xor* and *imul*. They match in frequency in the tests and in [HP02]. With a frequency of 0.4% and 0.3% they rae not nearly as frequent as *mov*, but they still are in the top ten most frequent function.

This can be compiled to a rough list of the most important operations for New Hope and can give an idea after which additional criteria hardware can be selected.

5. FrodoKEM

FrodoKEM is the second lattice based algorithm, which is analysed in this paper. This key encapsulation method has been submitted by Alkim et al. (2019) [Alk+19] to the NIST competition.

5.1. FrodoPKE

5.1.1. Preliminaries

Its is similar to New Hope as they have same underlying hardness assumptions based on LWE. The main difference is that FrodoKEM is based on LWE and not like New Hope based on a sub group, the RLWE problem. Thus the name 'Frodo' as it got rid of the ring.

The underlying scheme of FrodoKEM originates in Lindner and Peikert (2011) [LP11]. Furthermore FrodoKEM is an extension of FrodoPKE, which is extended in a similar way as New Hope with the Fujisaki-Okamoto transform (see section 4.4). Alkim et al. also considered another paper by Jiang et al. (2018) [Jia+18] which presented a variant of the Fujisaki-Okamoto transform in the quantum random oracle model, that requires one hash function less than [TU16], which was used to proof the quantum security of the FO transform.

The reasoning of Alkim et al. behind removing the ring from the LWE problem is, that a lattice used for a standard LWE instance is less "algebraically structured", while RLWE and the Module-LWE, which is a problem used in other NIST submissions [s. Ala+19, p. 10], introduce structure to their lattices, to speed up computation and efficiently run schemes based on them [c. Alk+19, p. 5]. However this structure introduces another vector for possible attacks. If algebraic structures can be abused to solve the underlying problems these schemes are broken, while FrodoKEM remains secure as it requires no structure within its lattice. This lack of structure also allows a simple implementation as no advanced mathematics have to be used to speed up computation, like the NTT (definition 4.2.4) of New Hope.

FrodoKEM uses a Gaussian distribution to sample its errors. The distribution has a standard deviation of $\sigma = 2.3$ for the security levels 1 and 3 and a deviation of $\sigma = 1.4$ for level 5 [c. Alk+19, p. 7]. As this parameter choice does not allow the classic full quantum reduction used for LWE problems, Alkim et al. show a reduction with these parameters [Alk+19, p. 38] and justify the worst case to average case reduction from hard lattice problems.

Definition 5.1.1. The function $i2b(x)$ is defined as converting an integer to its corresponding bit array.

The following constraints and definitions for parameters are give by Alkim et al. [c. Alk+19, p. 13]. These parameters are globally available:

- ξ is a probability distribution over $\mathbb{Z}$
- $q = 2^D$ with $D \leq 16$ is used as modulus
- $n, \overline{m}, \overline{n}$ with $n \equiv 0 \mod 8$, are used as matrix dimensions
- $B < D$ is the number of bits encoded in each matrix entry
- $l = B * \overline{m} * \overline{n}$ is the number of bits encoded per matrix
- len_{seed_A} is the length of the seed for pseudorandom matrix generation
- $len_{seed_{SE}}$ is the length of the seed for pseudorandom error sampling
- $x \xleftarrow{\$} U(y)$ says that x is randomly sampled from the uniform distribution over y
- T_χ is a distribution table and defined in more detail later

5.1.2. Algorithms

The first algorithms given in the specification of FrodoKEM are Encode and Decode.

Encode takes a bit string k of the length l as input and transforms it to a Matrix $K \in \mathbb{Z}_q^{\overline{m} \times \overline{n}}$. It iterates over every column and every row of the matrix and assigns it its designated value. When currently in the i-th column and j-th row calculate the unsigned integer represented by the next B bits beginning with $k_{(i*\overline{n}+j)*B}$. Then multiply the resulting integer with $q/2^B = 2^{D-B} \geq 1$ and save the result in the matrix at $K_{i,j}$. Once all elements of K have been filled return it.

Algorithm 25 Encode a bit string into a matrix [c. Alk+19, p. 14]

function FRODO.ENCODE($k \in \{0,1\}^l$)
  $K \leftarrow \mathbb{Z}_q^{\overline{m} \times \overline{n}}$
  for i **from** 0 **to** $\overline{m} - 1$ **do**
    for j **from** 0 **to** $\overline{n} - 1$ **do**
      $t \leftarrow \sum_{h=0}^{B-1} k_{(i*\overline{n}+j)*B+h} * 2^h$
      $K_{i,j} \leftarrow t * q/2^B$
  return $K = (K_{i,j})_{0 \leq i \leq \overline{m},\ 0 \leq j \leq \overline{n}} \in \mathbb{Z}_q^{\overline{m} \times \overline{n}}$

Decode reverses the encoding and creates a bit string from a given Matrix K. The function iterates over all columns and rows of the Matrix and decodes the value there. Decoding a value happens by multiplying it with $2^B/q < 1$, which is the inverse of value the number got expanded with while encoding. The result is rounded, reduced $\mod 2^B$ and converted into a bit array. This bit array then gets inserted at the right spot in the output bit string k. Once every value has been decoded k is returned.

Algorithm 26 Decode a matrix into a bit string [c. Alk+19, p. 14]

function FRODO.DECODE($K \in \mathbb{Z}_q^{\overline{m} \times \overline{n}}$)
  $k \leftarrow \{0,1\}^l$
  for i **from** 0 **to** $\overline{m} - 1$ **do**
    for j **from** 0 **to** $\overline{n} - 1$ **do**
      $t \leftarrow \text{i2b}(\lfloor K_{i,j} * 2^B/q \rceil \mod 2^B)$
      for h **from** 0 **to** $B - 1$ **do**
        $k_{(i*\overline{n}+j)*B+h} \leftarrow t_h$
  return $k \in \{0,1\}^l$

Decode is able to decode the correct k even if values in K have an error of up to $+\frac{2^{D-B}}{2} - 1 = 2^{D-B-1} - 1 = q/2^{B+1} - 1$ or $-\frac{2^{D-B}}{2} = -2^{D-B-1} = -q/2^{B+1}$ as this error is rounded away.

The next two function perform a similar task as Encode and Decode, however without error correction as D bits are packed into each entry in the matrix and for matrices of the dimensions $n_1 \times n_2$ rather than $\overline{m} \times \overline{n}$.

Pack takes a matrix $C \in \mathbb{Z}_q^{n_1 \times n_2}$ as input. It iterates over all columns and rows and extracts the D bits of each entry. The bits are then inserted into the bit string, which is eventually returned. In a real software implementation this string will be padded with 0's to make the length a multiple of 8 because it is saved as bytes.

Algorithm 27 Packing a matrix into a bit string [c. Alk+19, p. 14]

function FRODO.PACK($C \in \mathbb{Z}_q^{n_1 \times n_2}$)
  $b \leftarrow \{0,1\}^{D*n_1*n_2}$
  for i **from** 0 **to** $n_1 - 1$ **do**
    for j **from** 0 **to** $n_2 - 1$ **do**
      $t \leftarrow \text{i2b}(C_{i,j})$
      for h **from** 0 **to** $D - 1$ **do**
        $b_{(i*n_2+j)*D+h} \leftarrow t_{D-1-h}$
  return $b \in \{0,1\}^{D*n_1*n_2}$

Unpack reverses Pack and extracts an integer value from every D bits. To do so it iterates over all columns and rows as well, extracts the correct bit string, converts it to an integer and assigns it to the currently selected entry in the matrix C.

Algorithm 28 Extract a matrix from a bit string [c. Alk+19, p. 14]

function FRODO.UNPACK($b \in \{0,1\}^{D*n_1*n_2}$)
  $C \leftarrow \mathbb{Z}_q^{n_1 \times n_2}$
  for i **from** 0 **to** $n_1 - 1$ **do**
    for j **from** 0 **to** $n_2 - 1$ **do**
      $C_{i,j} \leftarrow \sum_{h=0}^{D-1} b_{(i*n_2+j)*D+h} * 2^{D-1-l}$
  return $C \in \mathbb{Z}_q^{n_1 \times n_2}$

A big difference between the New Hope and Frodo is the implementation of the Sample function. While New Hope (see algorithm 1) hashed a random seed with a nonce and then extracted integers by using the hamming weight of two bytes, Frodo does use a discrete probability density function described by a table. This means additionally to the random seed a tables of the form $T_\chi = (T_\chi(0), T_\chi(1), ..., T_\chi(s))$ is given as input to Sample, where s is a positive integer and is part of the supported Set S_χ of χ with $S_\chi = \{-s, -s+1, ..., -1, 0, 1, ..., s-1, s\}$ [c. Alk+19, p. 14].

The table has to satisfy multiple conditions.
Firstly $\chi(z) = \chi(-z)$ for $z \in S_\chi$, therefore only table entries for positive z have to be specified.
Furthermore $T_\chi(0) * 2^{-len_\chi} = \frac{1}{2}\chi(0) - 1$

and $T_\chi(z) * 2^{-len_\chi} = \frac{1}{2}\chi(0) - 1 + \sum_{i=1}^{z} \chi(i)$ for $1 \leq z \leq s$

have to be fulfilled [c. Alk+19, p. 14].

Sample therefore does take a seed r of the length len_χ and a table T_χ as input. It then calculates the integer value t of the seed without the first bit. Afterwards it iterates over the whole Table T_χ and increases a counting variable e by 1 for every $T_\chi(z) < t$. Eventually use the first bit of the seed, which was not used yet, to decide if e should be multiplied with -1 and return e afterwards.

Algorithm 29 Sampling an integer from $\mathbb{Z}$ [c. Alk+19, p. 15]

function FRODO.SAMPLE($r \in \{0,1\}^{len_\chi}, T_\chi$)
  $t \leftarrow \sum_{i=1}^{len_\chi - 1} r_i * 2^{i-1}$
  $e \leftarrow 0$
  for z **from** 0 **to** $s-1$ **do**
    if $t > T_\chi(z)$ **then**
      $e \leftarrow e + 1$
  $e \leftarrow (-1)^{r_0} * e$
  return $e \in \mathbb{Z}$

It is stressed, that the whole table has to be iterated and the comparison between T_χ and t has to execute in constant time. Only with this condition timed-side channel attacks can be avoided [c. Alk+19, p. 15].

SampleMatrix is sampling a whole $n_1 * n_2$ matrix using Sample. This method has a bit string r of $n_1 * n_2$ bit strings as input, of which each has a length of len_χ, as well as a Table T_χ. It then iterates through each column and each row of the matrix, which is being generated and generates a sample, with a new r_i each time, for all entries. The randomly sampled matrix is eventually returned.

Algorithm 30 Sampling a matrix from $\mathbb{Z}^{n_1 \times n_2}$ [c. Alk+19, p. 15]

function FRODO.SAMPLEMATRIX($r = (r^{(0)}, r^{(1)}, ..., r^{(n_1 * n_2 - 1)}) \in \{0,1\}^{n_1 * n_2 * len_\chi}, n_1, n_2, T_\chi$)
  $E \leftarrow \mathbb{Z}_q^{n_1 \times n_2}$
  for i **from** 0 **to** $n_1 - 1$ **do**
    for j **from** 0 **to** $n_2 - 1$ **do**
      $E_{i,j} \leftarrow$ Frodo.Sample($r^{i * n_2 + j}, T_\chi$)
  return $E \in \mathbb{Z}^{n_1 \times n_2}$

Gen is used to generate pseudo random matrices. As input a seed with a length of len_{seed_A} is given with which AES [Dwo+01] is initialized. It then iterates through each column of the matrix and 8 rows at a time. The column count and row count are then encoded as 16 bit long bit strings and extended with 96 0's. This 128 bit long string is then fed into AES. The 128 bit long string which is returned is then split into 8 groups of 16 bits, which are transformed into integers, reduced by mod q and set as value for the next 8 rows of the matrix. Once all entries have been set the matrix is returned.

Algorithm 31 Generating a random matrix using AES128

function FRODO.GEN($seed_A \in \{0,1\}^{len_{seed_A}}$)
  $A \leftarrow \mathbb{Z}^{n \times n}$
  for i **from** 0 **to** $n-1$ **do**
    $j \leftarrow 0$
    for $j < n$ **do**
      $b \leftarrow \text{i2b}(i)||\text{i2b}(j)||\{0\}^{96}$
      $c_{i,j}||c_{i,j+1}||...||c_{i,j+7} \leftarrow \text{AES128}_{seed_A}(b)$
      for k **from** 0 **to** 7 **do**
        $A_{i,j+k} \leftarrow \text{b2i}(c_{i,j+k}) \mod q$
      $j \leftarrow j + 8$
  return $A \in \mathbb{Z}_q^{n \times n}$

There also exists an alternative implementation which uses SHAKE128 and its modularity. For SHAKE128 a new seed for each column is generated by prepending the 16 bit value of the current column count to the seed. Now SHAKE128 can be run with the column unique seed and the parameter specifying how long the output bit string should be, which is set to $16n$. This results in enough bits to fill the whole column.

Algorithm 32 Generating a random matrix using SHAKE128

function FRODO.GEN($seed_A \in \{0,1\}^{len_{seed_A}}$)
  $A \leftarrow \mathbb{Z}^{n \times n}$
  for i **from** 0 **to** $n-1$ **do**
    $b \leftarrow \text{i2b}(i)||seed_A$
    $c_{i,0}||c_{i,1}||...||c_{i,n-1} \leftarrow \text{SHAKE128}(b, 16n)$
    for j **from** 0 **to** $n-1$ **do**
      $A_{i,j} \leftarrow \text{b2i}(c_{i,j}) \mod q$
  return $A \in \mathbb{Z}_q^{n \times n}$

These are all basic algorithms needed for FrodoPKE, therefore we define the key generation, encryption and decryption in the following.

KeyGen does not require any input. It at first randomly chooses a $seed_A$ from the uniform distribution over $\{0,1\}^{len_{seed_A}}$ and generates a pseudo random matrix from it. Then $seed_{SE}$ is randomly chosen from the uniform distribution over $\{0,1\}^{len_{seed_{SE}}}$. This seed is prepended with $0x5F$ and then used as seed for SHAKE, which is also given the parameter $2 * n * \overline{n} * len_\chi$ to output as many bits. These bits are stored in $2 * n * \overline{n}$ bit strings, each of the length len_χ. The first $n * \overline{n}$ bit strings are used together with $n, \overline{n}$ and T_χ to sample the secret matrix, while the second $n * \overline{n}$ bit strings are used with the same other parameters to sample the

error matrix. The $B = AS + E$ is computed and the public key $pk = (seed_A, B)$ and the private key S are returned.

Algorithm 33 FrodoPKE Key generation [c. Alk+19, p. 17]

function FrodoPKE.KeyGen()
$seed_A \xleftarrow{\$} U(\{0,1\}^{len_{seed_A}})$
$A \leftarrow \text{Frodo.Gen}(seed_A)$
$seed_{SE} \xleftarrow{\$} U(\{0,1\}^{len_{seed_{SE}}})$
$r = (r^{(0)}, r^{(1)}, ..., r^{(2*n*\overline{n}-1)}) \leftarrow \text{SHAKE}(0x5F||seed_{SE},\ 2*n*\overline{n}*len_\chi)$
$S \leftarrow \text{Frodo.SampleMatrix}(r^{(0)}, r^{(1)}, ..., r^{(n*\overline{n}-1)}, n, \overline{n}, T_\chi)$
$E \leftarrow \text{Frodo.SampleMatrix}(r^{(n*\overline{n})}, r^{(n*\overline{n}+1)}, ..., r^{(2*n*\overline{n}-1)}, n, \overline{n}, T_\chi)$
$B \leftarrow AS + E$
return $(pk \leftarrow (seed_A, B), sk \leftarrow S)$

Enc does take a message μ and a public key pk as input. First it regenerates the matrix A from $seed_A$ which is included in the public key. Then $seed_{SE}$ is, as in KeyGen, randomly chosen from the uniform distribution over $\{0,1\}^{len_{seed_{SE}}}$. The seed gets prepended with $0x96$ this time and put into SHAKE, which is instructed to output $(2*\overline{m}*n + \overline{m}*\overline{n})*len_\chi$ bits. From the first $\overline{m}*n*len_\chi$ bits a pseudo random secret S' is sampled, from the second an error E' and from the remaining $\overline{m}*\overline{n}*len_\chi$ a second error E'', which has different dimensions than the first error. Now $B' = S'A + E'$ and $V = S'B + E''$ are computed. The message is encoded with Frodo.Encode and added to V. Then the function returns the ciphertext c which consist of B' and the sum of V and the encoded message.

Algorithm 34 FrodoPKE Encryption [c. Alk+19, p. 17]

function FrodoPKE.Enc($\mu \in M, pk = (seed_A, B) \in \{0,1\}^{len_{seed_A}} \times \mathbb{Z}_q^{n \times \overline{n}}$)
$A \leftarrow \text{Frodo.Gen}(seed_A)$
$seed_{SE} \xleftarrow{\$} U(\{0,1\}^{len_{seed_{SE}}})$
$r = (r^{(0)}, r^{(1)}, ..., r^{(2*n*\overline{m}+\overline{m}*\overline{n}-1)}) \leftarrow \text{SHAKE}(0x96||seed_{SE},\ (2*n*\overline{m}+\overline{m}*\overline{n})*len_\chi)$
$S' \leftarrow \text{Frodo.SampleMatrix}(r^{(0)}, r^{(1)}, ..., r^{(n*\overline{m}-1)}, \overline{m}, n, T_\chi)$
$E' \leftarrow \text{Frodo.SampleMatrix}(r^{(n*\overline{m})}, r^{(n*\overline{m}+1)}, ..., r^{(2*n*\overline{m}-1)}, \overline{m}, n, T_\chi)$
$E'' \leftarrow \text{Frodo.SampleMatrix}(r^{(2*n*\overline{m})}, r^{(n*\overline{m}+1)}, ..., r^{(2*n*\overline{m}+\overline{m}*\overline{n}-1)}, \overline{m}, \overline{n}, T_\chi)$
$B' \leftarrow S'A + E'$
$V \leftarrow S'B + E''$
$C_1 \leftarrow B'$
$C_2 \leftarrow V + \text{Frodo.Encode}(\mu)$
return $c \leftarrow (C_1, C_2)$

Dec is very simple. It takes a ciphertext $c = (C_1, C_2) \in \mathbb{Z}_q^{\overline{m} \times n} \times \mathbb{Z}_q^{\overline{m} \times \overline{n}}$ and a secret key $sk = S \in \mathbb{Z}_q^{n \times \overline{n}}$ as input. It then computes $M = C_2 - C_1 S$, which can be decoded to the message μ' with Frodo.Decode. μ' is then returned.

$\mu' = \mu$ if the errors e which are left, are for each entry $-q/2^{B+1} \leq e < q/2^{B+1}$ as explained after algorithm 26 as well as in [c. Alk+19, p. 17].

Algorithm 35 FrodoPKE Decryption [c. Alk+19, p. 17]

function FrodoPKE.Dec($c = (C_1, C_2) \in \mathbb{Z}_q^{\overline{m} \times n} \times \mathbb{Z}_q^{\overline{m} \times \overline{n}}, sk = S \in \mathbb{Z}_q^{n \times \overline{n}}$)
  $M = C_2 - C_1 S$
  $\mu' \leftarrow$ Frodo.Decode(M)
  return $\mu' \in M$

5.2. Frodo-CCA-KEM

Based on the the Fujisaki-Okamoto transform (see section 4.4), the work by Targhi and Unruh (2016) [TU16] and the work by Hofheinz et al. (2017) [HHK17] the Frodo-PKE can be expanded to a CCA secure KEM in a similar way as New Hope was.

Alkim et al. define a KEM$^y = FO^y[PKE, G_1, G_2, F]$ and the additional parameters $len_S, len_k, len_{pkh}, len_{ss}$. $G_1 : \{0,1\}* \rightarrow \{0,1\}^{len_{pkh}}, G_2 : \{0,1\}* \rightarrow R \times \{0,1\}^{len_k}$ and $F : \{0,1\}* \rightarrow \{0,1\}^{len_{ss}}$ are hash functions. M is defined as message space.

This results in the following KEM algorithms for key generation, encapsulation and decapsulation:

KEM^y.KeyGen generates the key pair with PKE.KeyGen and samples a random secret s. It then hashes the public key pkh and combines the secret key, s, the public key and pkh and sets them as new secret key. The new public/secret key pair is returned.

Algorithm 36 KEM^y Key Generation [c. Alk+19, p. 19]

function KEM^y.KeyGen()
  $(pk, sk) \xleftarrow{\$}$ PKE.KeyGen()
  $s \xleftarrow{\$} \{0,1\}^{len_s}$
  $pkh \leftarrow G_1(pk)$
  $sk' \leftarrow (sk, s, pk, pkh)$
  return (pk, sk')

KEM^y.Encaps takes a public key as input. It then samples a random message from the set of all possible messages. It then hashes the hash of the public key together with the message and splits the resulting hash into r and k. While r is used as seed for the encryption k is part of the seed, which the shared secret gets hashed from. The random message is then encrypted with PKE.Enc, which

receives the message and the public key as parameters and k as well, which should be used as seed for SHAKE. The hash of the cyphertext and k is then saved as shared secret. Eventually the function returns the cyphertext and the shared secret.

Algorithm 37 KEM^y Encapsulation [c. Alk+19, p. 19]

function KEM^y.ENCAPS(pk)
  $\mu \xleftarrow{\$} M$
  $(r, k) \leftarrow G_2(G_1(pk)||\mu)$
  $c \leftarrow \text{PKE.Encrypt}(\mu, pk; r)$
  $ss \leftarrow F(c||k)$
  return (c, ss)

KEM^y.Decaps receives the cyphertext and the secret key, consisting of the original secret key, s, the public key and pkh. It decodes the message μ' with PKE.Dec the cyphertext and the secret. It then hashes pkh with the message and receives r' and k'. It now re-encrypts the message with r' as seed to compare it to the cyphertext and validate it. If the cyphertexts match generate the shared secret by hashing c and k'. If they do not match generate a fake shared secret by hashing c and s. In either way return the shared secret.

Algorithm 38 KEM^y Decapsulation [c. Alk+19, p. 19]

function KEM^y.DECAPS($c, (sk, s, pk, pkh)$)
  $\mu' \leftarrow \text{PKE.Decrypt}(c, sk)$
  $(r', k') \leftarrow G_2(pkh||\mu')$
  if $c = \text{PKE.Enc}(\mu', pk; r')$ **then**
    return $ss' \leftarrow F(c||k)$
  else
    return $ss' \leftarrow F(c||s)$

5.3. FrodoKEM

From these functions and Frodo-PKE an IND-CCA secure KEM can be constructed.

FrodoKEM.KeyGen merges the KEM.KeyGen and the Frodo-PKE.KeyGen. First $s, seed_{SE}, z$ are sampled together from a uniform distribution. $seed_A$ is not sampled but gained by expanding z with SHAKE. After that already described PKE.KeyGen takes place until right before the end, where B is packed to b. It then is hashed together with $seed_A$ as required by KEM.KeyGen. The returned public key is no longer a tuple but rather the concatenation of $seed_A$ and b. The secret key also has a concatenation of $s, seed_A$ and b as its first value, followed by S and pkh as separate values, which makes the secret key a 3-tuple.

Algorithm 39 FrodoKem.KeyGen [c. Alk+19, p. 20]

function FRODOKEM.KEYGEN()

$s||seed_{SE}||z \xleftarrow{\$} U(\{0,1\}^{len_s+len_{seed_{SE}}+len_z})$

$seed_A \leftarrow \text{SHAKE}(z, len_{seed_A})$

$A \leftarrow \text{Frodo.Gen}(seed_A)$

$r = (r^{(0)}, r^{(1)}, ..., r^{(2*n*\overline{n}-1)}) \leftarrow \text{SHAKE}(0x5F||seed_{SE},\ 2*n*\overline{n}*len_\chi)$

$S \leftarrow \text{Frodo.SampleMatrix}(r^{(0)}, r^{(1)}, ..., r^{(n*\overline{n}-1)}, n, \overline{n}, T_\chi)$

$E \leftarrow \text{Frodo.SampleMatrix}(r^{(n*\overline{n})}, r^{(n*\overline{n}+1)}, ..., r^{(2*n*\overline{n}-1)}, n, \overline{n}, T_\chi)$

$B \leftarrow AS + E$

$b \leftarrow \text{Frodo.Pack}(B)$

$pkh \leftarrow \text{SHAKE}(seed_A||b, len_{pkh})$

return $(pk \leftarrow seed_A||b; sk' \leftarrow (s||seed_A||b, S, pkh))$

FrodoKEM.Encaps merges the KEM and the PKE algorithms as well. Given a public key, it first chooses the random message and then hashes it, but r is replaced with $seed_{SE}$. It then generates the pseudo random bit string from $seed_{SE}$. It then proceeds with the normal PKE specification until except it uses Frodo.pack to pack B' and save it as c_1 and uses Frodo.unpack to unpack B from b. It eventually packs C into c_2. It then uses c_1 and c_2 together with k to generate the shared secret and returns it. It returns the cyphertext c as well, which is a c_2 appended to c_1.

Algorithm 40 FrodoKEM.Encaps [c. Alk+19, p. 20]

function FRODOKEM.ENCAPS($pk = seed_A||b \in \{0,1\}^{len_{seed_A}+D*n*\overline{n}}$)

$\mu \xleftarrow{\$} U(\{0,1\}^{len_\mu})$

$pkh \leftarrow \text{SHAKE}(pk, len_{pkh})$

$seed_{SE}||k \leftarrow \text{SHAKE}(pkh||\mu, len_{seed_{SE}} + len_k)$

$A \leftarrow \text{Frodo.Gen}(seed_A)$

$r = (r^{(0)}, r^{(1)}, ..., r^{(2*n*\overline{m}+\overline{m}*\overline{n}-1)}) \leftarrow \text{SHAKE}(0x96||seed_{SE},\ (2*n*\overline{m}+\overline{m}*\overline{n})*len_\chi)$

$S' \leftarrow \text{Frodo.SampleMatrix}(r^{(0)}, r^{(1)}, ..., r^{(n*\overline{m}-1)}, \overline{m}, n, T_\chi)$

$E' \leftarrow \text{Frodo.SampleMatrix}(r^{(n*\overline{m})}, r^{(n*\overline{m}+1)}, ..., r^{(2*n*\overline{m}-1)}, \overline{m}, n, T_\chi)$

$E'' \leftarrow \text{Frodo.SampleMatrix}(r^{(2*n*\overline{m})}, r^{(n*\overline{m}+1)}, ..., r^{(2*n*\overline{m}+\overline{m}*\overline{n}-1)}, \overline{m}, \overline{n}, T_\chi)$

$B' \leftarrow S'A + E'$

$B \leftarrow \text{Frodo.Unpack}(b, n\overline{n})$

$V \leftarrow S'B + E''$

$c_1 \leftarrow \text{Frodo.Pack}(B')$

$C \leftarrow V + \text{Frodo.Encode}(\mu)$

$c_2 \leftarrow \text{Frodo.Pack}(C)$

$ss \leftarrow \text{SHAKE}(c_1||c_2||k, len_{ss})$

return $(c \leftarrow c_1||c_2; ss)$

FrodoKEM.Decaps is much more complexe than PKE.Decaps, as it merges it with KEM.DEcaps which requires a ciphertext comparison, therefore a lot of steps need for encryption have to be repeated. Decaps receives the cipertext and the secret key as input. Right in the beginning c_1 and c_2 are unpacked to B' and C. With these matrices the message μ' can be decoded. Afterwards the public key is parsed and a lot of steps from the encryption are repeated. $seed'_{SE}$ and k' are initialized as well as the pseudo random bit string. Now S', E' and E'' are sampled again. A is generated and B is unpacked. Now B'' and V can be calculated. The encoded μ' is added to V to calculate the second ciphertext C'. Now the received unpacked ciphertext and the newly calculated one are compared to check the validity of the received ciphertext. If they match, the received one has been genuinely calculated by someone else to send a shared secret, which then is returned. If they do not match either something went wrong or a chosen ciphertext was injected by an attacker. In this case the shared secret will be generated with s included, which has randomly sampled and therefore produces an unrelated random shared secret.

Algorithm 41 FrodoKEM.Decaps [c. Alk+19, p. 20]

function FrodoKEM.Decaps($c = c_1||c_2 \in \{0,1\}^{(\overline{m}*n+\overline{m}*\overline{n})*D}, sk' = (s||seed_A||b, S, pkh) \in \{0,1\}^{len_s+len_{seed_A}+D*n*\overline{n}} \times \mathbb{Z}_q^{n\times\overline{n}} \times \{0,1\}^{len_{ss}}$)
  $B' \leftarrow$ Frodo.Unpack(c_1)
  $C \leftarrow$ Frodo.Unpack(c_2)
  $M \leftarrow C - B'S$
  $\mu' \leftarrow$ Frodo.Decode(M)
  $pk \leftarrow seed_A||b$
  $seed'_{SE}||k' \leftarrow$ SHAKE($pkh||\mu, len_{seed_{SE}} + len_k$)
  $r = (r^{(0)}, r^{(1)}, ..., r^{(2*n*\overline{m}+\overline{m}*\overline{n}-1)}) \leftarrow$ SHAKE($0x96||seed_{SE}$, $(2*n*\overline{m}+\overline{m}*\overline{n})*len_\chi$)
  $S' \leftarrow$ Frodo.SampleMatrix($r^{(0)}, r^{(1)}, ..., r^{(n*\overline{m}-1)}, \overline{m}, n, T_\chi$)
  $E' \leftarrow$ Frodo.SampleMatrix($r^{(n*\overline{m})}, r^{(n*\overline{m}+1)}, ..., r^{(2*n*\overline{m}-1)}, \overline{m}, n, T_\chi$)
  $E'' \leftarrow$ Frodo.SampleMatrix($r^{(2*n*\overline{m})}, r^{(n*\overline{m}+1)}, ..., r^{(2*n*\overline{m}+\overline{m}*\overline{n}-1)}, \overline{m}, \overline{n}, T_\chi$)
  $A \leftarrow$ Frodo.Gen($seed_A$)
  $B \leftarrow$ Frodo.Unpack($b, n\overline{n}$)
  $B'' \leftarrow S'A + E'$
  $V \leftarrow S'B + E''$
  $C' \leftarrow V +$ Frodo.Encode(μ')
  if $B'||C = B''||C'$ **then**
    return $ss \leftarrow$ SHAKE($c_1||c_2||k', len_{ss}$)
  else
    return $ss \leftarrow$ SHAKE($c_1||c_2||s, len_{ss}$)

This concludes the algorithm definitions, for the IND-CCA secure KEM Frodo.

5.4. Runtime Analysis of FrodoKEM

As the authors of New Hope, the authors of FrodoKEM have analysed CPU cycles and memory usage, however the focus of this paper still lies on assembly instructions used. FrodoKEM is specified with three different parameter sets for different security levels. They also offer choices while compiling. Their flexible make script allows the user to choose between gcc and clang as compilers. They also offer a choice between three different architectures to compile for, namely x64, x86 and ARM. Furthermore they added 3 options to choose from how optimized the output program should be. Firstly the reference compilation which is just a straightforward compilation, secondly the 'fast generic' option which optimizes matrix generation, multiplication and addition. The third option 'fast' instructs the compiler to use AVX2 instructions, which are specifically designed for vector operations and give a significant speed up. It is optional though as not every system supports AVX2 yet. The last two options are the choice between AES and SHAKE for matrix generation and if the standard openssl library should be used or if a custom implementation is preferred. The choice between AES and SHAKE matters because if a system offers the AES-NI instructions AES is significantly faster than SHAKE, as these instructions are made for AES. If those are not present SHAKE is faster. This is also relevant for future usage, as SHAKE will, as a new standard, most likely receive its own instruction set and therefore gain another huge speed boost [c. Alk+19, 26f.].

All these options only highlight the fact, that FrodoKEM relies on various instruction sets to be efficient. These sets however limit the hardware FrodoKEM can efficiently run on. These sets are standard in newer processors, however when running on slightly older hardware or in a virtualized OS they may not be present.

FrodoKEM was tested with a slight variation of the programs used for New Hope. However not a lot of changes had to be made because the functions were mainly standardised by the NIST. The new code can be found in appendix B.1 and appendix B.2. The main changes consists of changing names and paths, however the biggest change is that EvaluateGDB now breaks upon entering a function called *main*, which even is double checked. This was restructured as the optimized compilation of Frodo changes the breakFunction to an in-place operation, thus the breakFunction will never be called.

5.4.1. Results

$FrodoKEM1344$ was tested with 'fast-generic' instructions and SHAKE, as none of the available test environments offered the AES-NI instruction set or AVX2. These tests however yielded no usable results in a reasonable time. 20 Threads individually logged over 25.000.000 instructions without finishing the key generation. In comparison New Hope only required around 11.830.587 to complete all

three algorithms and only about 2.766.105 to finish key generation.

Running $FrodoKEM640$, after modifying the test code accordingly, also exceeded 7.000.000 instructions before finishing the key generation and did not produce any usable data within the test time.

This leads to the conclusion, that testing the whole system in one bruteforce attempt does not work. FrodoKEM should be revisited with a more selective approach that for example gathers information about the instruction composition of functions on a small scale and then upscales them considering the input variables.

6. Conclusion

While the outcome of the analysis of each of the two different cryptographic systems was very different they have a major similarity, lattices.

6.1. Lattices in cryptography

When introducing a lattice it seems simple, because its just all integer combinations of a set of linear independent vectors. But while looking further into the topic one can easily stumble across many interesting and surprisingly hard to solve problems. At first it does not seem hard to find the shortest vector or to find the closest one. That exactly makes lattices so interesting. When an easily describable problem over an easily describable body is not easily solvable and has not been solved for years it does suggest that this problem may not have a trivial solution. Many researchers have tried to prove the hardness or the weakness of lattice problems and slowly narrow down how hard these problems actually are or at which level of precision.

The simple problems and yet hard solutions are perfect for cryptography and have formed multiple groups of cryptographic systems. There are LWE based systems like FrodoKEM which value their generality over the advantages which come with structure in the lattices. These structures are for example exploited by RLWE to which New Hope belongs. And even more systems like Module-LWE exist, which however have not been discussed in this work.

6.2. New Hope and FrodoKEM

Both discussed systems showed pros and cons, however New Hope performed way better in the runtime analysis. This is why New Hope seems like a promising candidate for the NIST competition as it offers a decent security proof, while using small key sizes and not too many operations.

As the analysis has shown it can also greatly profit by optimizing just a few instructions. In contrary to FrodoKEM it does not rely on certain instruction sets to be fast, but can profit from them as well as it uses for example SHAKE too.

FrodoKEM requires without optimized instruction sets so many more instructions that it could not be properly evaluated. While the lack of these instructions was no problem when running the PQCrypto-KAT test case to en- and decrypt based on a deterministic seed, they still cause a bit discrepancy. In some cases

every operation counts, especially on smaller devices and that is where FrodoKEM fails. As emphasised in [RI16], Smart Devices arrive in every little niche of the day to day life, but they yet should be secure. While some micro controller already offer specialized instructions many do not and the smaller the devices become the more effort it is to included these sets. Therefore FrodoKEM seems not ideal to be chosen as new encryption standard. This however may change if micro controller producers focus on implementing the required instruction sets.

FrodoKEM can convince in the security aspect though, as they do not have possibly exploitable structure in their lattices. If the assumption underlying FrodoKEM breaks RLWE also breaks. If however RLWE breaks the LWE problem does not necessarily break. This is why it may be worth, even considering the bad performance, to pick a standard with less risk.

Out of these two systems New Hope will probably be faster on micro controllers and may lead to an immediate jump in security in these areas. FrodoKEM on the other hand offers long term stability and is simple to implement on any platform. The requirement of instruction-sets may initially slow the spreading though until the market catches up and offers these instructions as standard.

Bibliography

[ADR02] Jee Hea An, Yevgeniy Dodis, and Tal Rabin. "On the Security of Joint Signature and Encryption". In: *Advances in Cryptology — EUROCRYPT 2002*. Ed. by Lars R. Knudsen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 83–107. ISBN: 978-3-540-46035-0.

[Ajt98] Miklós Ajtai. "The Shortest Vector Problem in L2 is NP-hard for Randomized Reductions (Extended Abstract)". In: *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*. STOC '98. Dallas, Texas, USA: ACM, 1998, pp. 10–19. ISBN: 0-89791-962-9. DOI: 10.1145/276698.276705. URL: http://doi.acm.org/10.1145/276698.276705.

[Ala+19] Gorjan Alagic, Jacob Alperin-Sheriff, Daniel Apon, David Cooper, Quynh Dang, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, et al. *Status Report on the First Round of the NIST Post-Quantum Cryptography Standardization Process*. US Department of Commerce, National Institute of Standards and Technology, 2019-01. Jan. 2019. URL: https://doi.org/10.6028/NIST.IR.8240.

[Alk+19] Erdem Alkim, Joppe W. Bos, Léo Ducas, Patrick Longa, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Chris Peikert, Ananth Raghunathan, and Douglas Stebila. *FrodoKEM - Learning With Errors Key Encapsulation - Algorithm Specifications And Supporting Documentation*. July 2019. URL: https://frodokem.org/files/FrodoKEM-specification-20190702.pdf.

[BBD09] Daniel J. Bernstein, Johannes Buchmann, and Erik Dahmen. *Post-Quantum Cryptography* -. 2009. Aufl. Berlin Heidelberg: Springer Science & Business Media, 2009. ISBN: 978-3-540-88702-7.

[BCA04] M Bhattacharya, R Creutzburg, and J Astola. "Some historical notes on number theoretic transform". In: *Proc. 2004 Int. TICS Workshop on Spectral Methods and Multirate Signal Processing*. Vol. 2004. 2004.

[Che+16] Lidong Chen, Stephen P. Jordan, Yi-Kai Liu, Dustin Moody, Rene C. Peralta, Ray A. Perlner, and Daniel C. Smith-Tone. *Report on Post-Quantum Cryptography*. US Department of Commerce, National Institute of Standards and Technology, 2016-04-28. Apr. 2016. URL: http://dx.doi.org/10.6028/NIST.IR.8105.

[CS98] Ronald Cramer and Victor Shoup. "A practical public key cryptosystem provably secure against adaptive chosen ciphertext attack". In: *Advances in Cryptology — CRYPTO '98*. Ed. by Hugo Krawczyk. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 13–25. ISBN: 978-3-540-68462-6.

[DKS98] I. Dinur, G. Kindler, and S. Safra. "Approximating-CVP to within almost-polynomial factors is NP-hard". In: *Proceedings 39th Annual Symposium on Foundations of Computer Science (Cat. No.98CB36280)*. Nov. 1998, pp. 99–109. DOI: 10.1109/SFCS.1998.743433.

[Dwo+01] Morris J. Dworkin, Elaine B. Barker, James R. Nechvatal, James Foti, Lawrence E. Bassham, E. Roback, and James F. Dray Jr. *ADVANCED ENCRYPTION STANDARD (AES)*. Federal Inf. Process. Stds. (NIST FIPS) - 197. Nov. 2001. URL: https://doi.org/10.6028/NIST.FIPS.197.

[Ebe18] Jan Eberhardt. *Gitterbasierte Post-Quantum-Kryptographie*. Nov. 2018.

[FO99] Eiichiro Fujisaki and Tatsuaki Okamoto. "Secure Integration of Asymmetric and Symmetric Encryption Schemes". In: *Advances in Cryptology — CRYPTO' 99*. Ed. by Michael Wiener. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 537–554. ISBN: 978-3-540-48405-9.

[GG98] Oded Goldreich and Shafi Goldwasser. "On the Limits of Non-approximability of Lattice Problems". In: *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*. STOC '98. Dallas, Texas, USA: ACM, 1998, pp. 1–9. ISBN: 0-89791-962-9. URL: https://groups.csail.mit.edu/cis/pubs/shafi/1998-stoc.pdf.

[HHK17] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. *A Modular Analysis of the Fujisaki-Okamoto Transformation*. Cryptology ePrint Archive, Report 2017/604. https://eprint.iacr.org/2017/604. 2017.

[HP02] Jer Huang and Tzu-Chin Peng. "Analysis of x86 instruction set usage for DOS/Windows applications and its implication on superscalar design". In: *IEICE Transactions on Information and Systems* 85.6 (2002), pp. 929–939.

[HR07] Ishay Haviv and Oded Regev. "Tensor-based Hardness of the Shortest Vector Problem to Within Almost Polynomial Factors". In: *Proceedings of the Thirty-ninth Annual ACM Symposium on Theory of Computing*. STOC '07. San Diego, California, USA: ACM, 2007, pp. 469–477. ISBN: 978-1-59593-631-8. DOI: 10.1145/1250790.1250859. URL: http://doi.acm.org/10.1145/1250790.1250859.

[Jia+18] Haodong Jiang, Zhenfeng Zhang, Long Chen, Hong Wang, and Zhi Ma. "IND-CCA-Secure Key Encapsulation Mechanism in the Quantum Random Oracle Model, Revisited". In: *Advances in Cryptology – CRYPTO 2018*. Ed. by Hovav Shacham and Alexandra Boldyreva. Cham: Springer International Publishing, 2018, pp. 96–125. ISBN: 978-3-319-96878-0.

[LP11] Richard Lindner and Chris Peikert. "Better Key Sizes (and Attacks) for LWE-Based Encryption". In: *Topics in Cryptology – CT-RSA 2011*. Ed. by Aggelos Kiayias. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 319–339. ISBN: 978-3-642-19074-2.

[Mic08] Daniele Micciancio. "Efficient Reductions Among Lattice Problems". In: *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '08. San Francisco, California: Society for Industrial and Applied Mathematics, 2008, pp. 84–93. URL: http://dl.acm.org/citation.cfm?id=1347082.1347092.

[Pei09] Chris Peikert. "Public-key Cryptosystems from the Worst-case Shortest Vector Problem: Extended Abstract". In: *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing*. STOC '09. Bethesda, MD, USA: ACM, 2009, pp. 333–342. ISBN: 978-1-60558-506-2. DOI: 10.1145/1536414.1536461. URL: http://doi.acm.org/10.1145/1536414.1536461.

[Pie12] Krzysztof Pietrzak. "Cryptography from Learning Parity with Noise". In: *SOFSEM 2012: Theory and Practice of Computer Science*. Ed. by Mária Bieliková, Gerhard Friedrich, Georg Gottlob, Stefan Katzenbeisser, and György Turán. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 99–114. ISBN: 978-3-642-27660-6.

[Pöp+19] Thomas Pöppelmann, Erdem Alkim, Roberto Avanzi, Joppe Bos, Léo Ducas, Antonio de la Piedra, Peter Schwabe, and Douglas Stebila. *NewHope - Algorithm Specifications and Supporting Documentation*. July 2019. URL: https://newhopecrypto.org/data/NewHope_2019_07_10.pdf.

[Raz18] Ran Raz. "Fast Learning Requires Good Memory: A Time-Space Lower Bound for Parity Learning". In: *J. ACM* 66.1 (Dec. 2018), 3:1–3:18. ISSN: 0004-5411. DOI: 10.1145/3186563. URL: http://doi.acm.org/10.1145/3186563.

[Reg09] Oded Regev. "On Lattices, Learning with Errors, Random Linear Codes, and Cryptography". In: *J. ACM* 56.6 (Sept. 2009), 34:1–34:40. ISSN: 0004-5411. DOI: 10.1145/1568318.1568324. URL: http://doi.acm.org/10.1145/1568318.1568324.

[RI16] Francesco Regazzoni and Paolo Ienne. "Instruction Set Extensions for Secure Applications". In: *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*. DATE '16. Dresden, Germany: EDA Consortium, 2016, pp. 1529–1534. ISBN: 978-3-9815370-6-2. URL: http://dl.acm.org/citation.cfm?id=2971808.2972165.

[RS92] Charles Rackoff and Daniel R. Simon. "Non-Interactive Zero-Knowledge Proof of Knowledge and Chosen Ciphertext Attack". In: *Advances in Cryptology — CRYPTO '91*. Ed. by Joan Feigenbaum. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 433–444. ISBN: 978-3-540-46766-3.

[ST15] National Institute of Standards and Technology. *SHA-3 STANDARD: PERMUTATION-BASED HASH AND EXTENDABLE OUTPUT FUNCTIONS*. Federal Inf. Process. Stds. (NIST FIPS) - 202. Aug. 2015. URL: https://doi.org/10.6028/NIST.FIPS.202.

[TU16] Ehsan Ebrahimi Targhi and Dominique Unruh. "Post-Quantum Security of the Fujisaki-Okamoto and OAEP Transforms". In: *Theory of Cryptography*. Ed. by Martin Hirt and Adam Smith. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 192–216. ISBN: 978-3-662-53644-5.

Appendices

A. NewHope-KEM Assembly Operations

A.1. C-Code to execute all functions

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "rng.h"
#include "api.h"

#define MAX_MARKER_LEN    50
#define KAT_SUCCESS          0
#define KAT_FILE_OPEN_ERROR -1
#define KAT_DATA_ERROR      -3
#define KAT_CRYPTO_FAILURE  -4

void breakFunction();

int main(void){

  //init srand
  srand ((unsigned int) time (NULL));

  unsigned char entropy_input[48];
  unsigned char pk[CRYPTO_PUBLICKEYBYTES], sk[CRYPTO_SECRETKEYBYTES];
  unsigned char ct[CRYPTO_CIPHERTEXTBYTES], ss[CRYPTO_BYTES], ss1[
    CRYPTO_BYTES];
  int ret_val;

  for (int i=0; i<48; i++)
          entropy_input[i] = rand();

  randombytes_init(entropy_input, NULL, 256);

  // Generate the public/private keypair
    if ( (ret_val = crypto_kem_keypair(pk, sk)) != 0) {
        printf("crypto_kem_keypair returned <%d>\n", ret_val);
        return KAT_CRYPTO_FAILURE;
    }

  breakFunction();


  if ( (ret_val = crypto_kem_enc(ct, ss, pk)) != 0) {
            printf("crypto_kem_enc returned <%d>\n", ret_val);
            return KAT_CRYPTO_FAILURE;
```

```
42          }
43
44    breakFunction();
45
46      if ( (ret_val = crypto_kem_dec(ss1, ct, sk)) != 0) {
47          printf("crypto_kem_dec returned <%d>\n", ret_val);
48          return KAT_CRYPTO_FAILURE;
49      }
50
51    breakFunction();
52
53
54
55    return 0;
56  }
57
58  void breakFunction(){
59    printf("In the break\n");
60    return;
61  }
```

A.2. Java-Code to automate GDB

```
1  import java.io.File;
2  import java.io.FileWriter;
3  import java.io.IOException;
4  import java.io.OutputStream;
5  import java.util.*;
6
7  public class Main {
8
9      /**
10      * A Map of the operations to the lists of how often they have been
       counted during each cycle
11      */
12     private static final HashMap<String, ArrayList<Long>> KeyPairOpCount
       = new HashMap<>();
13     /**
14      * The amount of completed KeyPair-Generations so far
15      */
16     private static int KeyPairIterations = 0;
17
18     private static final HashMap<String, ArrayList<Long>> EncOpCount =
       new HashMap<>();
19     private static int EncIterations = 0;
20
21     private static final HashMap<String, ArrayList<Long>> DecOpCount =
       new HashMap<>();
22     private static int DecIterations = 0;
23
24     //Used to customize files
25     private static final long timestamp = System.currentTimeMillis();
26
27     //Used to validate if parent Thread is still alive.
28     private static boolean running = true;
29
30
31     public static void main(String[] args) throws IOException {
32
33         int totalThreadAmount;
34         int maxParallelThreads;
35
36
37         try {
38             totalThreadAmount = Integer.parseInt(args[0]);
39             maxParallelThreads = Integer.parseInt(args[1]);
40         } catch (Exception e){
41             System.out.println("Error parsing arguments. Specify the
       amount of full iterations as the first and the amount of parallel
       Threads as the second argument.");
42             return;
43         }
44
45         final ProcessBuilder pb = new ProcessBuilder( "/bin/bash","-c","
```

```
    printenv LD_BIND_NOW" );
46      Map<String, String> env = pb.environment();
47
48      //Set LD_BIND_NOW to 1 to load all dynamic libraries at the start
     of the program.
49      //This is needed to prevent errors during debugging.
50      env.put( "LD_BIND_NOW", "1" );
51      Process p = pb.start();
52      System.out.println( "LD_BIND_NOW="+new Scanner(p.getInputStream()
    ).nextLine() );
53
54
55
56
57
58      for(int i = 0; i < totalThreadAmount; i++) {
59
60          //Check how many iterations have been started minus the ones
    that finished to get the amount of currently running Threads
61          while(i - DecIterations >= maxParallelThreads){
62              try {
63                  Thread.sleep(30000);
64                  System.out.println(DecIterations+"/"+
    totalThreadAmount);
65              } catch (InterruptedException e) {
66                  e.printStackTrace();
67              }
68          }
69
70
71          //Start gdb with the test-file and the & and disown operation
     to prevent the process from dying a short while after the starting
    user logs out.
72          pb.command("gdb", "testNewHope", "&", "disown");
73
74
75          new Thread(() -> {
76              try {
77                  subroutine(pb.start());
78              } catch (IOException e) {
79                  e.printStackTrace();
80              }
81          }).start();
82
83          //Sleep to prevent using the same initial sample for the
    random number generator.
84          try {
85              Thread.sleep(1001);
86          } catch (InterruptedException e) {
87              e.printStackTrace();
88          }
89
90      }
91
```

```
92          while (DecIterations < totalThreadAmount){
93              try {
94                  Thread.sleep(30000);
95                  System.out.println(DecIterations+"/"+totalThreadAmount);
96              } catch (InterruptedException e) {
97                  e.printStackTrace();
98              }
99          }
100
101         writeList("KeyGeneration");
102         writeList("KeyEncryption");
103         writeList("KeyDecryption");
104     }
105
106     private static void subroutine(Process p) throws IOException {
107         OutputStream out = p.getOutputStream();
108
109         //We want to skip all c-code before and between this to only
    examine the relevant functions.
110         // So we can just start the program now and it will automatically
     stop there.
111         out.write("break crypto_kem_keypair\n".getBytes());
112         out.write("break crypto_kem_enc\n".getBytes());
113         out.write("break crypto_kem_dec\n".getBytes());
114
115         out.write("run\n".getBytes());
116         out.flush();
117
118
119         Scanner s = new Scanner(p.getInputStream());
120
121
122         //The amount each assembler operation has been counted
123         HashMap<String, Long> opCount = new HashMap<>();
124
125         //Ignored all lines produced by startup
126         for(int i = 0; i < 15; i++) s.nextLine();
127
128         //Ignore the line produced by the 'break crypto_kem_keypair'
    command.
129         s.nextLine();
130         s.nextLine();
131         s.nextLine();
132
133         //Ignore more lines produced by 'run'
134         s.nextLine();
135         s.nextLine();
136
137         initiateNextStep(out, s, opCount, "KeyGeneration");
138
139         //Skip to next break point
140         out.write("continue\n".getBytes());
141         out.flush();
142
```

```
143
144         //Reset the amount each assembler operation has been counted
145         opCount = new HashMap<>();
146
147
148         initiateNextStep(out, s, opCount, "KeyEncryption");
149
150         //Skip to next break point
151         out.write("continue\n".getBytes());
152         out.flush();
153
154
155         //Reset the amount each assembler operation has been counted
156         opCount = new HashMap<>();
157
158         //Ignore the lines first generated by executing 'run'.
159         initiateNextStep(out, s, opCount, "KeyDecryption");
160
161
162         System.out.println(Thread.currentThread().getName()+" End");
163     }
164
165     private static void initiateNextStep(OutputStream out, Scanner s,
    HashMap<String, Long> opCount, String s2) throws IOException {
166         //Ignore the lines first generated by executing 'run' or '
    continue'.
167         for (int i = 0; i < 2; i++) System.out.println(Thread.
    currentThread().getName()+" Skipped after continue: "+s.nextLine());
168
169         //Print which function the current Thread is now analyzing
170         System.out.println(Thread.currentThread().getName()+" Started: "+
    s2);
171
172         //Adding the two operations used to initiate the function which
    have been skipped.
173         opCount.put("push", 1L);
174         opCount.put("mov", 1L);
175
176         //Analyze the current function
177         opCount = countOps(s, out, opCount);
178
179         addToList(opCount,s2);
180     }
181
182
183     private static HashMap<String, Long> countOps(Scanner s, OutputStream
     out,  HashMap<String, Long> opCount) throws IOException {
184
185         Long currentLine = Long.decode(s.nextLine().split("\\s+")[2]);
186
187         //The amount of lines skipped.
188         int skipCount = 0;
189
190         //To keep track of the iterations
```

```
191         long i = 0L;
192 
193         //A helper to merge multiline output
194         StringBuilder construct;
195 
196         //Stores current relevant line
197         String line;
198         String[] splitLine;
199 
200         //A helper to keep track of the method we are in
201         String method = "";
202 
203         //Store the current operation
204         String op;
205 
206         //Temporarly store the count of the current operation
207         Long count;
208 
209         //To skip the first part if currentLine could not be updated
210         boolean skipDisas = false;
211 
212         //While we are not in the designated break function we still
    execute operations relevant to the function to analyze.
213         while(!method.equalsIgnoreCase("breakFunction")){
214 
215             //Check if parent is still alive
216             if(!running) return null;
217 
218             //Give some debug info every 100000 iterations to see the
    process is still alive
219             if(i % 100000 == 0) {
220                 System.out.println(Thread.currentThread().getName()+"
    Iteration: "+i);
221                 System.out.println(Thread.currentThread().getName()+"
    Currently in method: "+method);
222             }
223 
224             i++;
225 
226             //If no line number could be extracted from the last line
    skip disassembly.
227             if(!skipDisas) {
228 
229                 //Disassemble the current line, as LIne+1 is the excluded
     line to disassemble up to.
230                 out.write(("disas 0x" + Long.toHexString(currentLine) + "
    , 0x" + Long.toHexString(currentLine + 1) + "\n").getBytes());
231                 out.flush();
232 
233                 //Ignore next line
234                 s.nextLine();
235 
236                 construct = new StringBuilder(s.nextLine());
237                 line = s.nextLine();
```

```
238
239             //If the line contains assembler as third word ignore it
    as it is additional info by gdb
240             if( construct.toString().split("\\s+").length > 3 &&
    construct.toString().split("\\s+")[3].equalsIgnoreCase("assembler"))
    construct = new StringBuilder();
241
242             //If the last read line does not end the assembler dump
    concat it to the previous line and keep reading
243             while (!line.contains("End of assembler dump.")) {
244                 construct.append(line);
245                 line = s.nextLine();
246             }
247
248             String[] constParts = construct.toString().split("\\s+");
249
250             //Counter inconsistency in the gdb output. It outputs two
     different line styles based on information it get generate. These can
     be distinguished by the starting character of the second element.
251             if(constParts[2].startsWith("<")){
252                 op = constParts[3];
253             } else{
254                 op = constParts[2];
255             }
256
257             count = opCount.get(op);
258
259
260             if (count == null) {
261                 opCount.put(op, 1L);
262             } else {
263                 opCount.put(op, count + 1);
264             }
265
266         } else{
267             skipDisas = false;
268         }
269
270         //Step into the next operation
271         out.write("stepi\n".getBytes());
272         out.flush();
273
274
275         line = s.nextLine();
276
277         try{
278             splitLine = line.split("\\s+");
279
280             //If the line has less than 2 elements it is malformed.
    Go to the catch clause
281             if(splitLine.length < 2) throw new NumberFormatException
    ();
282
283
```

```
284                 construct = new StringBuilder(splitLine[1].trim());
285
286                 //If the extracted lineNumber does not start with 0x
    indicating it is a hex number it is not a usable line number.
287                 if(!construct.toString().startsWith("0x")) throw new
    NumberFormatException();
288
289                 currentLine = Long.decode(construct.toString());
290
291                 //If a method name is specified extract it
292                 if(splitLine.length >= 4) {
293                     if(!method.equals(splitLine[3])) {
294                         method = splitLine[3];
295                     }
296                 }
297
298             } catch (NumberFormatException e){
299
300                 //Count all lines that have been skipped for the
    statistics
301                 skipCount++;
302
303                 //Skip next disassemble as we have no line-index to
    disassemble at
304                 skipDisas = true;
305             }
306
307         }
308
309         //Add the skipped lines to the operations
310         opCount.put("LinesSkipped", (long) skipCount);
311
312         return opCount;
313
314     }
315
316
317     /**
318      * Add the operations a thread counted to the total count list.
319      * Synchronized to prevent ConcurrentModificationExceptions
320      * @param count
321      * @param type
322      */
323     private static synchronized void addToList(HashMap<String, Long>
    count, String type){
324
325         //Distinguish the 3 cases.
326         if(type.equalsIgnoreCase("KeyGeneration")){
327
328             //For all operations present in the OperationCount Map add
    the counted number.
329             // If the operation has not been counted add 0 to keep the
    length of all lists consistent
330             for(String key : KeyPairOpCount.keySet()){
```

```
331
332                 Long value = count.get(key);
333                 if(value == null) value = 0L;
334                 KeyPairOpCount.get(key).add(value);
335
336                 count.remove(key);
337
338             }
339
340             //If any new keys are present they are not part of the
    OperationCount Map yet.
341             for(String key : count.keySet()){
342                 ArrayList<Long> values = new ArrayList<>();
343
344                 //Fill with 0 for all previous iterations to keep list
    length consistent
345                 for(int i = 0; i < KeyPairIterations; i++) values.add(0L)
    ;
346
347                 values.add(count.get(key));
348                 KeyPairOpCount.put(key, values);
349             }
350
351             KeyPairIterations++;
352             //Write the results to the file every 5 iterations beginning
    with the 1st one.
353             if(KeyPairIterations % 5 == 1) writeList(type);
354
355
356         } else if(type.equalsIgnoreCase("KeyEncryption")){
357
358             //For all operations present in the OperationCount Map add
    the counted number.
359             // If the operation has not been counted add 0 to keep the
    length of all lists consistent
360             for(String key : EncOpCount.keySet()){
361
362                 Long value = count.get(key);
363                 if(value == 0) value = 0L;
364                 EncOpCount.get(key).add(value);
365
366                 count.remove(key);
367
368             }
369
370             //If any new keys are present they are not part of the
    OperationCount Map yet.
371             for(String key : count.keySet()){
372                 ArrayList<Long> values = new ArrayList<>();
373                 //Fill with 0 for all previous iterations
374                 for(int i = 0; i < EncIterations; i++) values.add(0L);
375
376                 values.add(count.get(key));
377                 EncOpCount.put(key, values);
```

```
378             }
379
380             EncIterations++;
381             //Write the results to the file every 5 iterations beginning
    with the 1st one.
382             if(EncIterations % 5 == 1) writeList(type);
383
384
385         } else if(type.equalsIgnoreCase("KeyDecryption")){
386
387             //For all operations present in the OperationCount Map add
    the counted number.
388             // If the operation has not been counted add 0 to keep the
    length of all lists consistent
389             for(String key : DecOpCount.keySet()){
390
391                 Long value = count.get(key);
392                 if(value == 0) value = 0L;
393                 DecOpCount.get(key).add(value);
394                 //DecOpCount.put(key, since);
395
396                 count.remove(key);
397
398             }
399
400             //If any new keys are present they are not part of the
    OperationCount Map yet.
401             for(String key : count.keySet()){
402                 ArrayList<Long> values = new ArrayList<>();
403                 //Fill with 0 for all previous iterations
404                 for(int i = 0; i < DecIterations; i++) values.add(0L);
405
406                 values.add(count.get(key));
407                 DecOpCount.put(key, values);
408             }
409
410             DecIterations++;
411             //Write the results to the file every 5 iterations beginning
    with the 1st one.
412             if(DecIterations % 5 == 1) writeList(type);
413         } else{
414             throw new RuntimeException("Invalid type");
415         }
416     }
417
418     private static synchronized void writeList(String type){
419         File dir = new File("assemblyOps");
420         File file = new File("assemblyOps/"+type+"_"+timestamp+".txt");
421
422         if(!dir.exists()) dir.mkdir();
423
424         if(!file.exists()) {
425             try {
426                 file.createNewFile();
```

```
427             } catch (IOException e) {
428                 e.printStackTrace();
429             }
430         }
431 
432         try {
433             FileWriter fw = new FileWriter(file,false);
434             StringBuilder line0 = new StringBuilder("N=");
435             StringBuilder line1 = new StringBuilder();
436             ArrayList<StringBuilder> lines = new ArrayList<>();
437 
438             if(type.equalsIgnoreCase("KeyGeneration")){
439 
440                 stringifyCount(line0, line1, lines, KeyPairOpCount,
    KeyPairIterations);
441 
442             } else if(type.equalsIgnoreCase("KeyEncryption")){
443 
444                 stringifyCount(line0, line1, lines, EncOpCount,
    EncIterations);
445 
446             } else if(type.equalsIgnoreCase("KeyDecryption")){
447 
448                 stringifyCount(line0, line1, lines, DecOpCount,
    DecIterations);
449 
450             } else{
451                 throw new RuntimeException("Invalid type");
452             }
453 
454             line1.append("\n");
455 
456             fw.write(line0.toString());
457             fw.write(line1.toString());
458 
459             for(StringBuilder sb : lines){
460                 fw.write(sb.append("\n").toString());
461             }
462 
463             fw.close();
464 
465 
466         } catch (IOException e) {
467             e.printStackTrace();
468         }
469 
470     }
471 
472     private static void stringifyCount(StringBuilder line0, StringBuilder
     line1, ArrayList<StringBuilder> lines, HashMap<String, ArrayList<Long
    >> OpCount, int Iterations) {
473         //As all Lists are equally long the amount of lines can be
    calculated from any value
474         int linesSize = OpCount.get(OpCount.keySet().iterator().next()).
```

```
    size();
475
476         //Create a StringBuilder for each line
477         while (lines.size() < linesSize) {
478             lines.add(new StringBuilder());
479         }
480
481
482         for (String key : OpCount.keySet()) {
483             //Append the operation name to the first line
484             line1.append(key).append(",");
485             ArrayList values = OpCount.get(key);
486
487             //Append the operation count to the string builder of each
    line.
488             //Overall one line represents one iteration.
489             for (int i = 0; i < values.size(); i++) {
490
491                 lines.get(i).append(values.get(i)).append(",");
492
493             }
494         }
495
496         line0.append(Iterations).append("\n");
497     }
498
499 }
```

A.3. KeyGeneration Instruction Table

The malformed operation '4155' will be interpreted as malformed line and thus added to LinesSkipped.

Table A.1.: Instructions counted for NH-CCA-KEM.KeyGen() in 100 runs, with n = 1024.

Assmbly Instruction	Total	Average	Variance	StdDeviation
$\sum$	276610484	2766104,84		
LinesSkipped	176600	1766	0	0
testb	4000	40	0	0
dec	13000	130	0	0
shufps	39000	390	0	0
aeskeygenassist	13000	130	0	0
subq	5400	54	0	0
cltq	4681720	46817,2	208,727272727272	14,4473967456865
lea	12610304	126103,04	0,079191919191919	0,281410588272579
seta	3000	30	0	0
imul	5734400	57344	0	0
jmp	664103	6641,03	0,029393939393939	0,171446607997765
movabs	3400	34	0	0
jae	9300	93	0	0
neg	1000	10	0	0
cmove	1000	10	0	0
leaveq	28600	286	0	0
mov	109248988	1092489,88	474,914747474748	21,7925388028735
subl	3	0,03	0,02939393939394	0,171446607997765
rep	20000	200	0	0
setne	2000	20	0	0
add	27379862	273798,62	52,2379797979798	7,22758464481599
jne	86003	860,03	0,029393939393939	0,171446607997765
shll	1000	10	0	0
punpcklwd	2000	20	0	0
test	82000	820	0	0
in	2000	20	0	0
sarw	204800	2048	0	0
pshufd	2000	20	0	0
jbe	6252860	62528,6	52,1818181818181	7,22369837284324
push	2326000	23260	0	0
jns	1003	10,03	0,02939393939394	0,171446607997766
jmpq	250800	2508	0	0
shl	4643060	46430,6	52,1818181818181	7,22369837284324
ja	118260	1182,6	52,181818181818	7,22369837284324
jb	1560400	15604	0	0
je	69900	699	0	0
shr	2579800	25798	0	0
jg	110860	1108,6	52,181818181818	7,22369837284324
sub	4919200	49192	0	0
jl	206600	2066	0	0
cmpq	1445400	14454	0	0
retq	2311300	23113	0	0
mul	3400	34	0	0

addl	1999660	19996,6	52,1818181818181	7,22369837284324
cmp	1796303	17963,03	0,029393939393939	0,171446607997765
cmpl	3048823	30488,23	210,421313131313	14,5059061465085
repz	16000	160	0	0
js	1000	10	0	0
nop	120500	1205	0	0
rol	2872800	28728	0	0
pop	2298400	22984	0	0
xorps	42000	420	0	0
not	3780000	37800	0	0
ror	1512000	15120	0	0
and	8705300	87053	0	0
andq	1000	10	0	0
xor	12047700	120477	0	0
addq	2735000	27350	0	0
pxor	7000	70	0	0
cmpw	109260	1092,6	52,181818181818	7,22369837284324
aesenc	13000	130	0	0
shrq	731200	7312	0	0
jle	1796200	17962	0	0
movzbl	14519843	145198,43	838,267777777777	28,952854397758
or	1176860	11768,6	52,1818181818183	7,22369837284325
sar	4505600	45056	0	0
movzwl	12902400	129024	0	0
movl	320700	3207	0	0
aesenclast	1000	10	0	0
movq	469200	4692	0	0
setg	1000	10	0	0
cmpb	3686400	36864	0	0
callq	2329500	23295	0	0
setb	1000	10	0	0
pushq	100	1	0	0
movslq	3661006	36610,06	0,117575757575757	0,34289321599553
movups	35000	350	0	0
movb	1550403	15504,03	0,029393939393939	0,171446607997765

A.4. Encryption Instruction Table

The malformed operation '4155' will be interpreted as malformed line and thus added to LinesSkipped.

Table A.2.: Instructions counted for NH-CCA-KEM.Enc() in 100 runs, with n = 1024.

Assmbly Instruction	Total	Average	Variance	StdDeviation
$\sum$	417959340	4179593,4		
LinesSkipped	88300	883	0	0
testb	2000	20	0	0
dec	6500	65	0	0
shufps	19500	195	0	0
aeskeygenassist	6500	65	0	0
subq	7100	71	0	0
cltq	6844920	68449,2	208,727272727272	14,4473967456865
lea	19107800	191078	0	0
seta	1500	15	0	0
imul	9113600	91136	0	0
jmp	990000	9900	0	0
movabs	5300	53	0	0
jae	113700	1137	0	0
neg	26100	261	0	0
cmove	500	5	0	0
leaveq	39300	393	0	0
mov	166744880	1667448,8	469,636363636363	21,6710951185297
rep	10000	100	0	0
setne	1000	10	0	0
jne	55400	554	0	0
add	40493460	404934,6	52,1818181818183	7,22369837284325
shll	1500	15	0	0
punpcklwd	1000	10	0	0
test	41000	410	0	0
in	1000	10	0	0
sarw	204800	2048	0	0
pshufd	1000	10	0	0
jbe	9202560	92025,6	52,1818181818182	7,22369837284325
push	3432100	34321	0	0
jns	500	5	0	0
jmpq	351000	3510	0	0
shl	6889560	68895,6	52,1818181818182	7,22369837284325
ja	113760	1137,6	52,181818181818	7,22369837284324
jb	2204700	22047	0	0
je	47100	471	0	0
shr	3546800	35468	0	0
jg	110860	1108,6	52,181818181818	7,22369837284324
sub	7363900	73639	0	0
jl	309900	3099	0	0
cmpq	1901700	19017	0	0
retq	3424100	34241	0	0
mul	107700	1077	0	0
addl	3213860	32138,6	52,1818181818181	7,22369837284324

cmp	2638300	26383	0	0
cmpl	4784020	47840,2	208,727272727272	14,4473967456865
repz	8000	80	0	0
js	500	5	0	0
nop	164400	1644	0	0
rol	4468800	44688	0	0
pop	3393300	33933	0	0
xorps	21000	210	0	0
not	5880000	58800	0	0
ror	2352000	23520	0	0
and	13565100	135651	0	0
andq	500	5	0	0
xor	18491500	184915	0	0
addq	3729200	37292	0	0
pxor	3500	35	0	0
cmpw	109260	1092,6	52,181818181818	7,22369837284324
aesenc	6500	65	0	0
shrq	989600	9896	0	0
jle	2875500	28755	0	0
movzbl	21883840	218838,4	834,90909090909	28,894793491373
or	1820660	18206,6	52,1818181818181	7,22369837284324
sar	7091200	70912	0	0
movzwl	19195200	191952	0	0
movl	478800	4788	0	0
aesenclast	500	5	0	0
movq	694100	6941	0	0
setg	500	5	0	0
cmpb	5529600	55296	0	0
callq	3433200	34332	0	0
setb	500	5	0	0
pushq	100	1	0	0
movslq	5966500	59665	0	0
movups	17500	175	0	0
movb	2218400	22184.0.0	0	0

A.5. Decryption Instruction Table

Table A.3.: Instructions counted for NH-CCA-KEM.Dec() in 100 runs, with n = 1024.

Assmbly Instruction	Total	Average	Variance	StdDeviation
$\sum$	488488840	4884888,4		
LinesSkipped	0	0	0	0
subw	25600	256	0	0
subq	6700	67	0	0
cltq	9057720	90577,2	208,727272727273	14,4473967456865
lea	24912900	249129	0	0
imul	12492800	124928	0	0
jmp	968800	9688	0	0
movabs	5100	51	0	0
jae	213100	2131	0	0
neg	25700	257	0	0
leaveq	139100	1391	0	0
mov	195275480	1952754,8	469,636363636364	21,6710951185297
add	49223360	492233,6	52,1818181818181	7,22369837284325
shll	2000	20	0	0
jne	18500	185	0	0
sarw	307200	3072	0	0
jbe	9292960	92929,6	52,1818181818182	7,22369837284325
push	4404400	44044	0	0
jmpq	439400	4394	0	0
shl	8410360	84103,6	52,1818181818182	7,22369837284325
ja	109260	1092,6	52,1818181818181	7,22369837284324
jb	2350300	23503	0	0
je	17800	178	0	0
shr	5036800	50368	0	0
jg	110860	1108,6	52,1818181818181	7,22369837284324
sub	10528500	105285	0	0
jl	413200	4132	0	0
retq	4403100	44031	0	0
mul	107500	1075	0	0
cmpq	1657900	16579	0	0
addl	3875560	38755,6	52,1818181818182	7,22369837284325
cmp	2976600	29766	0	0
cmpl	5969720	59697,2	208,727272727273	14,4473967456865
nop	159400	1594	0	0
rol	3784800	37848	0	0
pop	4265300	42653	0	0
not	4980000	49800	0	0
ror	1992000	19920	0	0
and	13853800	138538	0	0
xor	16255900	162559	0	0
addq	3684100	36841	0	0
cmpw	109260	1092,6	52,1818181818181	7,22369837284324
shrq	962400	9624	0	0
negb	100	1	0	0
jle	3729700	37297	0	0

movzbl	23439640	234396,4	834,909090909091	28,894793491373
or	2195160	21951,6	52,1818181818182	7,22369837284325
sar	7910400	79104	0	0
movzwl	26896000	268960	0	0
movl	589200	5892	0	0
movq	628900	6289	0	0
shrw	25600	256	0	0
cmpb	5529600	55296	0	0
callq	4403100	44031	0	0
pushq	200	2	0	0
movslq	8121600	81216	0	0
movb	2194400	21944	0	0

B. FrodoKEM

B.1. C-Code to execute all functions

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "../tests/rng.h"
#include "../src/api_frodo1344.h"

#define MAX_MARKER_LEN    50
#define KAT_SUCCESS          0
#define KAT_FILE_OPEN_ERROR -1
#define KAT_DATA_ERROR      -3
#define KAT_CRYPTO_FAILURE  -4

void breakFunction();

int main(void){

  //init srand
  srand ((unsigned int) time (NULL));

  unsigned char entropy_input[48];
  unsigned char pk[CRYPTO_PUBLICKEYBYTES], sk[CRYPTO_SECRETKEYBYTES];
  unsigned char ct[CRYPTO_CIPHERTEXTBYTES], ss[CRYPTO_BYTES], ss1[
    CRYPTO_BYTES];
  int ret_val;

  for (int i=0; i<48; i++)
          entropy_input[i] = rand();

  randombytes_init(entropy_input, NULL, 256);

  // Generate the public/private keypair
    if ( (ret_val = crypto_kem_keypair_Frodo1344(pk, sk)) != 0) {
        printf("crypto_kem_keypair returned <%d>\n", ret_val);
        return KAT_CRYPTO_FAILURE;
    }

  breakFunction();


  if ( (ret_val = crypto_kem_enc_Frodo1344(ct, ss, pk)) != 0) {
            printf("crypto_kem_enc returned <%d>\n", ret_val);
            return KAT_CRYPTO_FAILURE;
        }
```

```
43
44   breakFunction();
45
46     if ( (ret_val = crypto_kem_dec_Frodo1344(ss1, ct, sk)) != 0) {
47         printf("crypto_kem_dec returned <%d>\n", ret_val);
48         return KAT_CRYPTO_FAILURE;
49     }
50
51   breakFunction();
52
53
54
55   return 0;
56 }
57
58 void breakFunction(){
59   printf("In the break\n");
60   return;
61 }
```

B.2. Java-Code to automate GDB

```
1  import java.io.File;
2  import java.io.FileWriter;
3  import java.io.IOException;
4  import java.io.OutputStream;
5  import java.util.*;
6
7  public class Main {
8
9      /**
10      * A Map of the operations to the lists of how often they have been
       counted during each cycle
11      */
12     private static final HashMap<String, ArrayList<Long>> KeyPairOpCount
       = new HashMap<>();
13     /**
14      * The amount of completed KeyPair-Generations so far
15      */
16     private static int KeyPairIterations = 0;
17
18     private static final HashMap<String, ArrayList<Long>> EncOpCount =
       new HashMap<>();
19     private static int EncIterations = 0;
20
21     private static final HashMap<String, ArrayList<Long>> DecOpCount =
       new HashMap<>();
22     private static int DecIterations = 0;
23
24     //Used to customize files
25     private static final long timestamp = System.currentTimeMillis();
26
27     //Used to validate if parent Thread is still alive.
28     private static boolean running = true;
29
30
31     public static void main(String[] args) throws IOException {
32
33         int totalThreadAmount;
34         int maxParallelThreads;
35
36
37         try {
38             totalThreadAmount = Integer.parseInt(args[0]);
39             maxParallelThreads = Integer.parseInt(args[1]);
40         } catch (Exception e){
41             System.out.println("Error parsing arguments. Specify the
       amount of full iterations as the first and the amount of parallel
       Threads as the second argument.");
42             return;
43         }
44
45         final ProcessBuilder pb = new ProcessBuilder( "/bin/bash","-c","
```

```
    printenv LD_BIND_NOW" );
46      Map<String, String> env = pb.environment();
47
48      //Set LD_BIND_NOW to 1 to load all dynamic libraries at the start
     of the program.
49      //This is needed to prevent errors during debugging.
50      env.put( "LD_BIND_NOW", "1" );
51      Process p = pb.start();
52      System.out.println( "LD_BIND_NOW="+new Scanner(p.getInputStream()
    ).nextLine() );
53
54
55
56
57
58      for(int i = 0; i < totalThreadAmount; i++) {
59
60          //Check how many iterations have been started minus the ones
    that finished to get the amount of currently running Threads
61          while(i - DecIterations >= maxParallelThreads){
62              try {
63                  Thread.sleep(30000);
64                  System.out.println(DecIterations+"/"+
    totalThreadAmount);
65              } catch (InterruptedException e) {
66                  e.printStackTrace();
67              }
68          }
69
70
71          //Start gdb with the test-file and the & and disown operation
     to prevent the process from dying a short while after the starting
    user logs out.
72          pb.command("gdb", "testFrodoAssembly", "&", "disown");
73
74
75          new Thread(() -> {
76              try {
77                  subroutine(pb.start());
78              } catch (IOException e) {
79                  e.printStackTrace();
80              }
81          }).start();
82
83          //Sleep to prevent using the same initial sample for the
    random number generator.
84          try {
85              Thread.sleep(1001);
86          } catch (InterruptedException e) {
87              e.printStackTrace();
88          }
89
90      }
91
```

```
92          while (DecIterations < totalThreadAmount){
93              try {
94                  Thread.sleep(30000);
95                  System.out.println(DecIterations+"/"+totalThreadAmount);
96              } catch (InterruptedException e) {
97                  e.printStackTrace();
98              }
99          }
100
101         writeList("KeyGeneration");
102         writeList("KeyEncryption");
103         writeList("KeyDecryption");
104     }
105
106     private static void subroutine(Process p) throws IOException {
107         OutputStream out = p.getOutputStream();
108
109         //We want to skip all c-code before and between this to only
    examine the relevant functions.
110         // So we can just start the program now and it will automatically
     stop there.
111         out.write("break crypto_kem_keypair_Frodo1344\n".getBytes());
112         out.write("break crypto_kem_enc_Frodo1344\n".getBytes());
113         out.write("break crypto_kem_dec_Frodo1344\n".getBytes());
114
115         out.write("run\n".getBytes());
116         out.flush();
117
118
119         Scanner s = new Scanner(p.getInputStream());
120
121
122         //The amount each assembler operation has been counted
123         HashMap<String, Long> opCount = new HashMap<>();
124
125         //Ignored all lines produced by startup
126         for(int i = 0; i < 15; i++) s.nextLine();
127
128         //Ignore the line produced by the 'break crypto_kem_keypair'
    command.
129         s.nextLine();
130         s.nextLine();
131         s.nextLine();
132
133         //Ignore more lines produced by 'run'
134         s.nextLine();
135         s.nextLine();
136
137         initiateNextStep(out, s, opCount, "KeyGeneration");
138
139         //Skip to next break point
140         out.write("continue\n".getBytes());
141         out.flush();
142
```

```
143
144         //Reset the amount each assembler operation has been counted
145         opCount = new HashMap<>();
146
147
148         initiateNextStep(out, s, opCount, "KeyEncryption");
149
150         //Skip to next break point
151         out.write("continue\n".getBytes());
152         out.flush();
153
154
155         //Reset the amount each assembler operation has been counted
156         opCount = new HashMap<>();
157
158         //Ignore the lines first generated by executing 'run'.
159         initiateNextStep(out, s, opCount, "KeyDecryption");
160
161
162         System.out.println(Thread.currentThread().getName()+" End");
163     }
164
165     private static void initiateNextStep(OutputStream out, Scanner s,
    HashMap<String, Long> opCount, String s2) throws IOException {
166         //Ignore the lines first generated by executing 'run' or '
    continue'.
167         for (int i = 0; i < 2; i++) System.out.println(Thread.
    currentThread().getName()+" Skipped after continue: "+s.nextLine());
168
169         //Print which function the current Thread is now analyzing
170         System.out.println(Thread.currentThread().getName()+" Started: "+
    s2);
171
172         //Adding the two operations used to initiate the function which
    have been skipped.
173         opCount.put("push", 1L);
174         opCount.put("mov", 1L);
175
176         //Analyze the current function
177         opCount = countOps(s, out, opCount);
178
179         addToList(opCount,s2);
180     }
181
182
183     private static HashMap<String, Long> countOps(Scanner s, OutputStream
     out,  HashMap<String, Long> opCount) throws IOException {
184
185         Long currentLine = Long.decode(s.nextLine().split("\\s+")[2]);
186
187         //The amount of lines skipped.
188         int skipCount = 0;
189
190         //To keep track of the iterations
```

```
191         long i = 0L;
192 
193         //A helper to merge multiline output
194         StringBuilder construct;
195 
196         //Stores current relevant line
197         String line;
198         String[] splitLine;
199 
200         //A helper to keep track of the method we are in
201         String method = "";
202 
203         //Store the current operation
204         String op;
205 
206         //Temporarly store the count of the current operation
207         Long count;
208 
209         //To skip the first part if currentLine could not be updated
210         boolean skipDisas = false;
211 
212         //While we are not in the designated break function we still
    execute operations relevant to the function to analyze.
213         while(!method.equalsIgnoreCase("main")){
214 
215             //Check if parent is still alive
216             if(!running) return null;
217 
218             //Give some debug info every 100000 iterations to see the
    process is still alive
219             if(i % 100000 == 0) {
220                 System.out.println(Thread.currentThread().getName()+"
    Iteration: "+i);
221                 System.out.println(Thread.currentThread().getName()+"
    Currently in method: "+method);
222             }
223 
224             i++;
225 
226             //If no line number could be extracted from the last line
    skip disassembly.
227             if(!skipDisas) {
228 
229                 //Disassemble the current line, as LIne+1 is the excluded
     line to disassemble up to.
230                 out.write(("disas 0x" + Long.toHexString(currentLine) + "
    , 0x" + Long.toHexString(currentLine + 1) + "\n").getBytes());
231                 out.flush();
232 
233                 //Ignore next line
234                 s.nextLine();
235 
236                 construct = new StringBuilder(s.nextLine());
237                 line = s.nextLine();
```

```
238
239             //If the line contains assembler as third word ignore it
    as it is additional info by gdb
240             if( construct.toString().split("\\s+").length > 3 &&
    construct.toString().split("\\s+")[3].equalsIgnoreCase("assembler"))
    construct = new StringBuilder();
241
242             //If the last read line does not end the assembler dump
    concat it to the previous line and keep reading
243             while (!line.contains("End of assembler dump.")) {
244                 construct.append(line);
245                 line = s.nextLine();
246             }
247
248             String[] constParts = construct.toString().split("\\s+");
249
250             //Counter inconsistency in the gdb output. It outputs two
     different line styles based on information it get generate. These can
     be distinguished by the starting character of the second element.
251             if(constParts[2].startsWith("<")){
252                 if(constParts[2].contains("main")) break;
253                 op = constParts[3];
254             } else{
255                 op = constParts[2];
256             }
257
258             count = opCount.get(op);
259
260
261             if (count == null) {
262                 opCount.put(op, 1L);
263             } else {
264                 opCount.put(op, count + 1);
265             }
266
267         } else{
268             skipDisas = false;
269         }
270
271         //Step into the next operation
272         out.write("stepi\n".getBytes());
273         out.flush();
274
275
276         line = s.nextLine();
277
278         try{
279             splitLine = line.split("\\s+");
280
281             //If the line has less than 2 elements it is malformed.
    Go to the catch clause
282             if(splitLine.length < 2) throw new NumberFormatException
    ();
283
```

```
284
285                 construct = new StringBuilder(splitLine[1].trim());
286
287                 //If the extracted lineNumber does not start with 0x
    indicating it is a hex number it is not a usable line number.
288                 if(!construct.toString().startsWith("0x")) throw new
    NumberFormatException();
289
290                 currentLine = Long.decode(construct.toString());
291
292                 //If a method name is specified extract it
293                 if(splitLine.length >= 4) {
294                     if(!method.equals(splitLine[3])) {
295                         method = splitLine[3];
296                     }
297                 }
298
299             } catch (NumberFormatException e){
300
301                 //Count all lines that have been skipped for the
    statistics
302                 skipCount++;
303
304                 //Skip next disassemble as we have no line-index to
    disassemble at
305                 skipDisas = true;
306             }
307
308         }
309
310         //Add the skipped lines to the operations
311         opCount.put("LinesSkipped", (long) skipCount);
312
313         return opCount;
314
315     }
316
317
318     /**
319      * Add the operations a thread counted to the total count list.
320      * Synchronized to prevent ConcurrentModificationExceptions
321      * @param count
322      * @param type
323      */
324     private static synchronized void addToList(HashMap<String, Long>
    count, String type){
325
326         //Distinguish the 3 cases.
327         if(type.equalsIgnoreCase("KeyGeneration")){
328
329             //For all operations present in the OperationCount Map add
    the counted number.
330             // If the operation has not been counted add 0 to keep the
    length of all lists consistent
```

```
331             for(String key : KeyPairOpCount.keySet()){
332 
333                 Long value = count.get(key);
334                 if(value == null) value = 0L;
335                 KeyPairOpCount.get(key).add(value);
336 
337                 count.remove(key);
338 
339             }
340 
341             //If any new keys are present they are not part of the
    OperationCount Map yet.
342             for(String key : count.keySet()){
343                 ArrayList<Long> values = new ArrayList<>();
344 
345                 //Fill with 0 for all previous iterations to keep list
    length consistent
346                 for(int i = 0; i < KeyPairIterations; i++) values.add(0L)
    ;
347 
348                 values.add(count.get(key));
349                 KeyPairOpCount.put(key, values);
350             }
351 
352             KeyPairIterations++;
353             //Write the results to the file every 5 iterations beginning
    with the 1st one.
354             if(KeyPairIterations % 5 == 1) writeList(type);
355 
356 
357         } else if(type.equalsIgnoreCase("KeyEncryption")){
358 
359             //For all operations present in the OperationCount Map add
    the counted number.
360             // If the operation has not been counted add 0 to keep the
    length of all lists consistent
361             for(String key : EncOpCount.keySet()){
362 
363                 Long value = count.get(key);
364                 if(value == 0) value = 0L;
365                 EncOpCount.get(key).add(value);
366 
367                 count.remove(key);
368 
369             }
370 
371             //If any new keys are present they are not part of the
    OperationCount Map yet.
372             for(String key : count.keySet()){
373                 ArrayList<Long> values = new ArrayList<>();
374                 //Fill with 0 for all previous iterations
375                 for(int i = 0; i < EncIterations; i++) values.add(0L);
376 
377                 values.add(count.get(key));
```

```
                EncOpCount.put(key, values);
            }

            EncIterations++;
            //Write the results to the file every 5 iterations beginning with the 1st one.
            if(EncIterations % 5 == 1) writeList(type);


        } else if(type.equalsIgnoreCase("KeyDecryption")){

            //For all operations present in the OperationCount Map add the counted number.
            // If the operation has not been counted add 0 to keep the length of all lists consistent
            for(String key : DecOpCount.keySet()){

                Long value = count.get(key);
                if(value == 0) value = 0L;
                DecOpCount.get(key).add(value);
                //DecOpCount.put(key, since);

                count.remove(key);

            }

            //If any new keys are present they are not part of the OperationCount Map yet.
            for(String key : count.keySet()){
                ArrayList<Long> values = new ArrayList<>();
                //Fill with 0 for all previous iterations
                for(int i = 0; i < DecIterations; i++) values.add(0L);

                values.add(count.get(key));
                DecOpCount.put(key, values);
            }

            DecIterations++;
            //Write the results to the file every 5 iterations beginning with the 1st one.
            if(DecIterations % 5 == 1) writeList(type);
        } else{
            throw new RuntimeException("Invalid type");
        }
    }

    private static synchronized void writeList(String type){
        File dir = new File("assemblyOps");
        File file = new File("assemblyOps/"+type+"_"+timestamp+".txt");

        if(!dir.exists()) dir.mkdir();

        if(!file.exists()) {
            try {
```

```
427                 file.createNewFile();
428             } catch (IOException e) {
429                 e.printStackTrace();
430             }
431         }
432 
433         try {
434             FileWriter fw = new FileWriter(file,false);
435             StringBuilder line0 = new StringBuilder("N=");
436             StringBuilder line1 = new StringBuilder();
437             ArrayList<StringBuilder> lines = new ArrayList<>();
438 
439             if(type.equalsIgnoreCase("KeyGeneration")){
440 
441                 stringifyCount(line0, line1, lines, KeyPairOpCount,
    KeyPairIterations);
442 
443             } else if(type.equalsIgnoreCase("KeyEncryption")){
444 
445                 stringifyCount(line0, line1, lines, EncOpCount,
    EncIterations);
446 
447             } else if(type.equalsIgnoreCase("KeyDecryption")){
448 
449                 stringifyCount(line0, line1, lines, DecOpCount,
    DecIterations);
450 
451             } else{
452                 throw new RuntimeException("Invalid type");
453             }
454 
455             line1.append("\n");
456 
457             fw.write(line0.toString());
458             fw.write(line1.toString());
459 
460             for(StringBuilder sb : lines){
461                 fw.write(sb.append("\n").toString());
462             }
463 
464             fw.close();
465 
466 
467         } catch (IOException e) {
468             e.printStackTrace();
469         }
470 
471     }
472 
473     private static void stringifyCount(StringBuilder line0, StringBuilder
     line1, ArrayList<StringBuilder> lines, HashMap<String, ArrayList<Long
    >> OpCount, int Iterations) {
474         //As all Lists are equally long the amount of lines can be
    calculated from any value
```

```
475         int linesSize = OpCount.get(OpCount.keySet().iterator().next()).
    size();
476 
477         //Create a StringBuilder for each line
478         while (lines.size() < linesSize) {
479             lines.add(new StringBuilder());
480         }
481 
482 
483         for (String key : OpCount.keySet()) {
484             //Append the operation name to the first line
485             line1.append(key).append(",");
486             ArrayList values = OpCount.get(key);
487 
488             //Append the operation count to the string builder of each
    line.
489             //Overall one line represents one iteration.
490             for (int i = 0; i < values.size(); i++) {
491 
492                 lines.get(i).append(values.get(i)).append(",");
493 
494             }
495         }
496 
497         line0.append(Iterations).append("\n");
498     }
499 
500 }
```