Department of Informatics

Master Thesis

Performance and Security of Modern and Post-Quantum Cryptographic Algorithms

Markus Johan Ragnhildstveit
Supervisor: Øyvind Ytrehus

June 1, 2020

Contents

Chapter 1

Introduction

The threat of a quantum computer having the capabilities to break modern Public Key Infrastructure (PKI) seems to be stepping closer and closer to be reality by companies such as Google, IBM, Intel and Microsoft. To meet this threat, a project was created by NIST called NIST Post Quantum Cryptography (NIST PQC). The goal of the project is to find a replacement for all cryptosystems that are broken by a large enough quantum computer. This means they intend to find both classical- and quantum-secure Public Key Encryption (PKE) schemes, Key Encapsulating Mechanisms (KEMs) and Digital Signatures.

In this thesis I will look closer at some of these PKE schemes. What I hope to achieve is to educate myself and hopefully others on Post-Quantum Cryptography. As you find many of the Post-Quantum Cryptographic algorithms already implemented in the NIST PQC project, I wanted to look at them and find out how they performed and how well optimized the implementations are.

1.1 Research Questions

The research questions I ended up with and try to answer are:

- What is considered bottlenecks in a computer when looking at performance measurements of post-quantum cryptographic algorithms?
- What is the performance of candidates in the NIST Post-Quantum Cryptography project and how do they compare to each other when resources are constrained?
- What kind of performance increase or decrease does updated versions of the cryptographic algorithm candidates have compared to the original versions in round 2 of the NIST Post-Quantum Cryptography project?
- What does the resources and components in a computer that affect the performance of post-quantum cryptographic algorithms say about the optimization of these algorithms?

While writing the thesis I will create a project for combining the cryptosystems and measure the performance of each system. I will also research to find areas that can affect the performance of these systems and what can be done for optimizing them. In this thesis I will start by looking at background information that can be important to know and understand before we look at the project itself and the results.

1.2 Thesis Outline

In the background chapter we will first look at the different types of mathematical principals the cryptographic algorithms in the NIST PQC project consists of, which are lattice based cryptography, code based cryptography and multivariate polynomial cryptography. When this is better understood we will look at the NIST PQC project itself. The candidates I am focusing on will be explained in chapter 3 for themselves. In the background chapter I will also go through how a quantum computer works and why it breaks modern PKI. The rest of the chapter will consist of different ways optimizations are possible, ways to measure the performance of the implementations and topics that are interesting to look at when it comes to post-quantum cryptography.

In the chapter about the candidates to the NIST PQC project, I will go through the six candidates that I will focus on. For each of them I will look at their features, how they work, a small discussion about their security, the parameters of the schemes and any Intellectual Property (IP) that they might have with possible patents.

The methods chapter I will look closer at what my methodology was through this thesis and how I worked with the project. This contains an explanation of how I researched areas and how I implemented the candidates and other features into the program I created.

In the results and discussion chapter I will present and visualize the results I got from the performance measurements. I will also try to find reasons behind the results and if we can connect them to some kind of optimization of the candidates for answering the research questions. The results I show and visualize are the results I believe gives me the best ability to answer the research questions.

In the conclusion and afterthoughts chapter I will try to answer the research questions from the results I found and give a few afterthoughts and future works of this topic.

Chapter 2

Background

2.1 Lattice Based Cryptography

Some of the Post-Quantum algorithms are implemented as lattice based cryptosystems. Since many of the lattice based cryptosystems are similar, I will describe them as a group. The difference for each algorithm will be described for itself. I will start with the description of a lattice based cryptosystem with learning with error (LWE) problem.

For convenience, I will make an example for Alice and Bob communicating. Alice generates a random matrix $A^{m \times n}$ where $a_{ij} < q$, $a_{ij} \in \mathbb{Z}$ for $i \in \{1, ..., n\}$ and $j \in \{1, ..., m\}$. This Matrix is public, and Bob has access to it. Alice then choose a random vector $\vec{x}$ as her private key, and calculates $\vec{u} = A\vec{x}$ as her public key.

Bob chooses a random and secret vector $\vec{s}$. He computes:

$$\vec{b_1} = A\vec{s} + \vec{e_1}$$

The vector $\vec{e_1}$ is an error vector, with small values so it will later be possible to decrypt the message. He then computes:

$$b_2 = \vec{s} \times \vec{u} + e_2 + bit \times q/2$$

The *bit* is the message Bob is sending to Alice, either zero or one. e_2 is a random error value where $e_2 << q/4$, which means e_2 is much smaller than $q/4$. The q-value is the same value as was used when A was generated. Bob sends $\vec{b}_1$ and b_2 to Alice. For Alice to find out what Bob sent, she use the equation:

$$\begin{aligned} b_2 - \vec{b}_1\vec{x} &= \vec{s}\vec{u} + e_2 + bit \times q/2 - (A\vec{s} + \vec{e}_1)\vec{x} \\ &= \vec{s}\vec{u} + e_2 + bit \times q/2 - (\vec{s}\vec{u} + \vec{e}_1\vec{x}) \\ &= (e_2 - \vec{e}_1\vec{x}) + bit \times q/2 \end{aligned}$$

If *bit* is zero in the last equation, Then the whole equation will be close to zero, otherwise the equation will be far from zero. This is because $(e_2 - \vec{e}_1\vec{x})$ is very small, and $bit \times q/2$ is only small if $bit = 0$. Therefore:

$$(e_2 - \vec{e}_1\vec{x}) + bit \times q/2 \approx bit * q/2$$

If you don't know the values of $\vec{x}$ beforehand, then the hardness of this cryptosystems is as hard as the learning with error (LWE) problem [1]. Some of the hard problems used in lattice based cryptosystems are:

- Learning With Error
- Shortest Vector Problem
- Closest Vector Problem
- Covering Vector Problem

2.2 Code Based Cryptography

Binary entropy function	$h(\cdot)$

Another type of cryptosystem that can be quantum-secure are code based cryptosystems. The basics used in the code based cryptosystems dates back to Claude Shannon in 1948. At this time, the goal of the technique was to remove errors that occured when data was sent through cables.

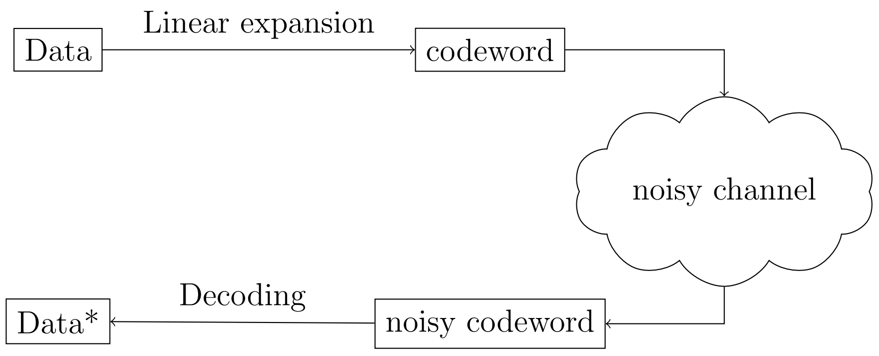

Figure 2.1: Linear codes for communication.

The linear expansion expands the data of length k to a codeword of length $n > k$. This means the codewords $\mathcal{C}$ is a subspace of a field with n dimensions. These codewords can be represented as linearly independent vectors in a matrix called *generator matrix*. Through the noisy channels bits may be flipped so that the message sent is changed. This means to successfully decode to the correct data it will depend on what linear expansion was used and how large the expansion was. The probability of decoding for a binary symmetric channel of error rate p is 1 if $\frac{k}{n} < (1 - h(p))$ [2]. This theorem is non constructive, which means there is no generic solution for polynomial time decoding. When the linear expansion is random, we know that the decoding is NP-complete [3]. Even a small amount of error is assumed to be difficult to remove without knowing the linear expansion. There does exists codes such as alternate codes that have a polynomial time decoder for $O(\frac{n}{log(n)})$ errors.

For this to be used in a cryptosystem, you could use the same scheme as in figure 2.1, but with some changes. Data is made into a codeword with linear expansion, and

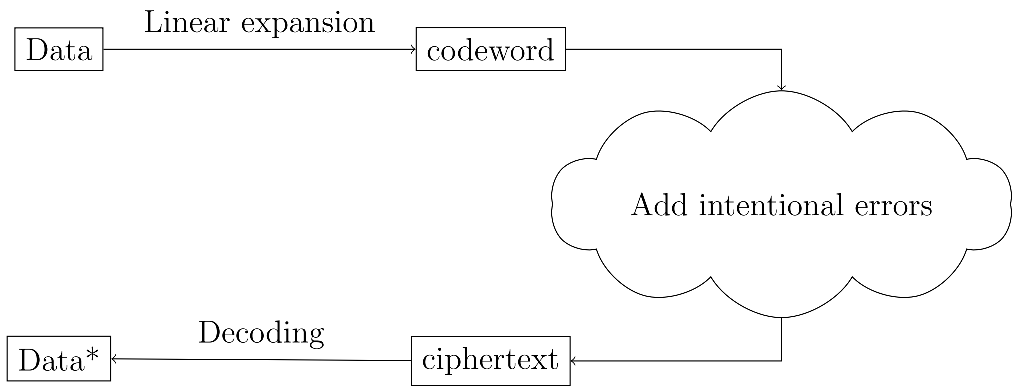

Figure 2.2: Linear codes for cryptosystem.

intentional errors are added. This means if a random linear code is used, no one will be able to decode the data efficiently as this was a NP-complete problem described earlier. If the code that was used was an alternate code, then it is known to have a fast decoder for the ones who know the code structure. If we assume the knowledge of the linear expansion does not reveal the code structure, we see that this can be used for public key encryption. The linear expansion is used to encrypt the data and some errors has to be added intentionally. The decoding is the decryption with the knowledge of the code structure as the private key. The probability of decoding successfully can be 1 as the amount of error added is intentionally and will therefore not need to extend above the threshold of $\frac{k}{n} < (1-h(p))$.This is of course important to find a balance between security, performance and probability of decoding correctly in a scheme like this.

Linear Error Correcting Codes

One way of using linear code is to use a generator matrix. A q-ary linear code $[n,k]$ is a k-dimensional subspace of the n-ary finite field $\mathbb{F}_q^n$. q will most often be 2 as most finite fields will be binary. A generator matrix can be described as $G \in \mathbb{F}_q^{n\times m}$ of the codewords $\mathcal{C}$ such that:

$$\mathcal{C} = \left\{\vec{x}G | \vec{x} \in \mathbb{F}_q^k\right\}$$

This can be used as an encoder defined as f_G:

$$\begin{aligned} f_G \colon \mathbb{F}_q^k &\to \mathcal{C} \in \mathbb{F}_q^n \\ \vec{x} &\mapsto \vec{x}G. \end{aligned}$$

To get the message back you can use the right inverse of G. So $f_G(\vec{x})G' = \vec{x}GG' = \vec{x}$. If G is in systematic/standard form, which means it is in the form:

$$G = \left[I_k | P\right]$$

Then the inverse of G is:

$$G' = \left[I_k | 0\right]^T$$

Parity check matrix and Syndrome

Parity check matrix	H
t-bounded decoder	$\Phi(\cdot)$
t-bounded syndrome decoder	$\Psi(\cdot)$
Hamming weight	$wt(\cdot)$

Using a parity check matrix and syndrome is a more relevant way of working with linear codes when creating a cryptosystem. The parity check matrix is a matrix to check the relation of a vector to the codewords in the linear code. The parity check matrix is a generator of the dual code of $\mathcal{C}$. It is therefore quite useful when it comes to decoding and correcting errors. A q-ary linear code $[n, k]$ is a k-dimensional subspace of $\mathbb{F}_q^n$. Let $r = n - k$, the parity check matrix $H \in \mathbb{F}_q^{r\times n}$ of the codewords $\mathcal{C}$ such that:

$$\mathcal{C} = \left\{\vec{x} \in \mathbb{F}_q^n \,\middle|\, \vec{x}H^T = 0\right\}$$

The syndrome of a vector $\vec{y} \in \mathbb{F}_q^n$ is

$$S_H(\vec{y}) = \vec{y}H^T$$

Definitions: If the syndrome of $\vec{y}$ is zero, it means it is a codeword. The linear code has several nice properties. For all $\vec{z} \in \mathbb{F}_q^n$ and all $\vec{c} \in \mathcal{C}$, the coset of $\vec{z}$ can be defined as an additive group like this:

$$Coset(\vec{z}) = \{\vec{z} + \vec{c} \,|\, \forall \vec{c} \in \mathcal{C}\} = S_H^{-1}(\vec{z})$$

Definition: $\Phi_{\mathcal{C}}(\cdot)$ can be considered a t-bounded decoder if for all $\vec{x} \in \mathcal{C}$, all $\vec{e} \in \mathbb{F}_q^n$ where $wt(e) \leq t$:

$$\Phi_{\mathcal{C}}(\vec{x} + \vec{e}) = \vec{x}$$

Definition: $\Psi_H(\cdot)$ can be considered a t-bounded H-syndrome decoder if for all $\vec{e} \in \mathbb{F}_q^n$ where $wt(\vec{e}) \leq t$:

$$\Psi_H(\vec{e}H^T) = \vec{e}$$

It is proven that $\exists\, \Phi_C(\cdot) \leftrightarrow \exists\, \Psi_H(\cdot)$. With this information, it is possible to create a public key encryption scheme. In 1978 Robert McEliece created a public key encryption scheme based on these principles.

Quasi-Cyclic Codes

Quasi-Cyclic Codes are something that will be mentioned a lot, therefore I will try to explain it. For a code to be cyclic it means that if we have $c = \{c_0, c_1, ..., c_n\}$ in the code, then $c' = \{c_n, c_0, ..., c_{n-1}\}$ will also be in the code. Turning c into c' is called a cyclic right shift. For quasi-cyclic code, a set amount of cyclic right shift needs to be applied before another codeword is discovered. Since we are working in finite fields, if we look at the polynomial $c(x)$ as a codeword and multiplying it with x is considered a cyclic right shift, then we find $x^b c(x)$ to be a codeword. The irreducible polynomial in Quasi-Cyclic codes has to be on the form $x^n - 1$, where b and n are coprimes.

2.3 Multivariate Polynomial Based Cryptography

Multivariate polynomial cryptography are a group of public key cryptosystems based on usually multivariate quadratic polynomials in a map over finite fields, often represented in an array. Many trapdoor functions were introduced before RSA, but were forgotten when RSA became popular. The first publications of MPKC (Multivariate Public Key Cryptosystem) were published in the early 1980s in Japanese, but not before 1988 the first modern versions of a MPKC were published.

The variables are often represented as an array. The public key P is the set of these polynomials:

$$P = (p_1(\vec{x}), ..., p_n(\vec{x}))$$

Where $\vec{x}$ is the vector of variables $\vec{x} = (x_1, ..., x_m)$. Each $p_i(\cdot)$ can be represented as:

$$p_i(\vec{x}) = \sum_j Q_{ji}x_j + \sum_j R_{ji}x_j^2 + \sum_{j>k} S_{jki}x_jx_k$$

This means the public key contains Q, R and S. All variables and coefficients are in the finite field $\mathbb{F}_q$.

There are a lot of different ways to hide the secret part of the system depending on the algorithm, but there are some similarities. Most of the MPKC hides a private transformation R by two affine and full-rank transformations. This can be written as:

$$P(\vec{x}) = Q(R(S(\vec{x}))) = \vec{z} = (z_1, ..., z_n)$$

The transformation (or map) R is a quadratic transformation where the inverse is easy to compute. It is the design of this transformation that decides the key of the cryptosystem. The disadvantage with MPKC is that they have relatively large key sizes compared of RSA and ECC. This can become a problem on smaller devices with limited memory. However they often are quite efficient to run.

2.4 Deterministic Encryption and Decryption

Deterministic encryption and decryption algorithms are algorithms who does not have any random element in the encryption and decryption. This means that for a given plaintext and key, the encryption will always return the same ciphertext. This is not a desired attribute, as it is possible to do a chosen ciphertext attack on the system because it is possible to recognize if the same plaintext being sent multiple times. A solution for this is to use padding with random elements.

2.5 Probabilistic Encryption and Decryption

A probabilistic encryption algorithm is the opposite of a deterministic encryption algorithm. It has some random part of the encryption which causes the ciphertext given a plaintext and key to change between encryptions. Some of the probabilistic encryption algorithms have a probabilistic decryption algorithm. A problem that can arise with a probabilistic decryption algorithm is that there can occur failures when decrypting the ciphertext. This causes the need for the sender to encrypt the data another time. If an algorithm has too large failure rate, it can not be considered an IND-CCA2 secure algorithm. The decryption failure rate (DFR) refers to the likelihood of decryption failing to decrypt the correct plaintext. There are however probabilistic encryption algorithms with a deterministic decryption. This means that they do not have failures in the decryption.

2.6 Security Notions and Attack Model

Some security notions will be used through the thesis. These are created so it is possible to prove the security of a cryptosystem. Most of the cryptosystems I will go through have a proven or claimed security notion of IND-CPA, IND-CCA or IND-CCA2. These security notions possesses the property of indistinguishability under different type of attacks. In the NIST PQC project, NIST intends to standardize one or more ”semantically secure” public key encryption or key encapsulating mechanism with regard to adaptive chosen ciphertext attack [4]. This is denoted as IND-CCA2.

COA

COA stands for *Ciphertext-Only Attack*. The most common COA is brute-force attack on the private key. The only information available for the adversary is the ciphertext. It is considered the weakest attack since the lack of information.

KPA

KPA stands for *Known Plaintext Attack*. In KPA the information available are pairs of plaintext and ciphertext corresponding together. The most famous example of this is under World War 2 with the successfull cryptanalysis of the enigma machine.

CPA

CPA stands for *Chosen Plaintext Attack*. CPA is based on the adversary being able to choose plaintext to be encrypted and have access to the result of the encryption. This attack is realistic with public key encryption since the public key is available, and therefore any adversary could use it to encrypt any plaintext. The IND-CPA notation which will be used for claiming security for schemes stands for *Indistinguishability under Chosen Plaintext Attack*. IND-CPA can be explained with the IND-CPA Game between a challenger and an adversary:

1. The challenger generates a private and public key keypair with a security parameter. The challenger shares the public key with the adversary.

2. The adversary can do polynomial time calculations with the public key and different plaintexts.
3. Eventually the adversary submits two plaintext without encrypting them.
4. The challenger chooses randomly between the two plaintext and return the encryption of the chosen one.
5. The adversary has to guess which plaintext of the encryption and is free to perform any computations before the guess. Finally, publishes the guess to the challenger.

If a public key encryption scheme is IND-CPA then the probability of the adversary guessing correct is $\frac{1}{2} + \epsilon(k)$. The $\epsilon(k)$ is the negligible function for the security parameter k.

CCA

CCA stands for *Chosen Ciphertext Attack*. CCA is based on the adversary being able to choose the ciphertext and see the plaintext corresponding to it. In reality this would require the adversary to be the recipient and have access to the communication channel. The IND-CCA notation will also be used through the thesis for claiming security for schemes. It stands for *Indistinguishability under Chosen Ciphertext Attack*. IND-CCA can be explained in a similar way as IND-CPA:

1. The challenger generates a private and public key keypair with a security parameter. The challenger shares the public key with the adversary.
2. The adversary may perform polynomial time calculations with the public key, encryptions and decryptions.
3. Eventually the adversary submits two plaintext without encrypting them.
4. The challenger chooses randomly between the two plaintext and return the encryption of the chosen one.
5. The adversary has to guess which plaintext of the encryption and is free to perform any computations and encryptions before the guess. However no decryptions. Finally, publishes the guess to the challenger.

CCA2

CCA2 stands for *Adaptive Chosen Ciphertext Attack*. CCA2 is similar to CCA, in additon to everything the adversary is given in CCA, the adversary in CCA2 also have access to the decryption oracle in the 5th step. IND-CCA2 is used as being ”semantically secure” in the NIST PQC project.

Open Key Model Attacks

The open key model attacks are attacks where the adversary have knowledge about the key used for scheme [5]. A type of attack is the Related-key attack. In this attack the aversary can encrypt plaintext with keys that have some mathematical relation with the key to the challenger.

Side-Channel Attacks

Side-Channel attacks are attacks that depend on gathering information through other means than a cryptanalysis. This could be a timing attack to measure the time it takes to perform the algorithm if it's not implemented in constant time.

2.7 Difficult Problems for Cryptography

The method cryptologist review if a cryptosystem is secure, is to reduce the difficulty of the system to a hard problem. These hard problems are either assumed or proven to either only have an exponential solution or non-deterministic solution. These problems are categorized as either NP-hard, NP-complete or at least no known efficient method for solving. Each candidate has to prove that their system either reduce to a known difficult problem, or give well enough proof of a new difficult problem for the cryptologists in NIST to review.

2.8 NIST Post-Quantum Cryptography Project

The NIST Post-Quantum Cryptography project is a competition between cryptographic algorithms for becoming a new standard for public key encryption, key encapsulation mechanism and digital signature algorithm. The reason this project started is to avoid the exploitation of the quantum mechanical phenomena that will affect many of public key cryptosystems in use today.

Algorithm	Type	Purpose	Impact of quantum computer
RSA	Asymmetric key	Signature & KEM	Not secure
ECC	Asymmetric key	Signature & KEM	Not secure
DSA	Asymmetric key	Signature & KEM	Not secure
AES	Symmetric key	Encryption	Double key size to be secure

December 20th, 2016 NIST sent a request for nominations for "Public-Key Post-Quantum Cryptographic Algorithms"[6]. The final deadline was set to November 30th, 2017 for candidate algorithms to be submitted. There were a total of 82 sumbmission for the first round, and 69 of these were accepted. The only criteria of round 1 submission acceptance were to be "complete and proper" as it is defined in FRN-Dec16 [7]. This required the submissions to have:

- Reference and optimized implementation of the submission in C
- Known-answer tests (KAT)
- Written specification
- Intellectual property (IP) statement

In addition algorithms were required in FRN-Dec16 [7] to be implemented in a wide range of hardware and software platforms.

The project identified three main areas for evaluating the candidate algorithms through the standardization process. These are:

1. Security
2. Cost and performance
3. Algorithm and implementation characteristics

According to NIST it seems unlikely that any of the known implementations of the candidates can be a full replacement for the current public key schemes used today. One of the reasons is that most of the candidates have a much larger key size. Including the fact that the algorithms for signature and key encapsulation may not work the same as RSA because they are different schemes.

When the second round started, 26 out of the 69 first-round candidates were selected. The candidates were evaluated on the three areas as described in order of importance. The security evaluation of the algorithms were not only based on the internal cryptanalysis of NIST, but also external cryptanalysis. Performance was also evaluated, but did not have a major role as it is still in a early portion of the evaluation process [8]. A few of the candidates were selected based on their unique and elegant design. This was done for diversification so it is not as easy to find a single type of attack that will eliminate all of the candidates. Out of the selected algorithms for round 2, 17 of them are public-key encryption and key-establishment schemes.

2.8.1 Security

The standardization of new public-key algorithms means the algorithms are going to be used in different applications, such as Transport Layer Security (TLS), Secure Shell (SSH), Internet Key Exchange (IKE), Internet Protocol Security (IPsec), and Domain Name System Security Extensions (DNSSEC) [8]. The algorithms will therefore be evaluated by their security in these and other applications. Security is the most important factor of the standardization, just as it was in AES and SHA-3 competitions.

For the ability to compare security between candidates easier, five security categories were defined. While there is significant uncertainties when estimating the security, the candidate were asked to provide a classification for their algorithms. The focus was to meet the requirements to 1, 3 and 5. The categories are:

1. Any attacks require at least the same processing power as key search on a block cipher with 128-bit key
2. Any attacks require at least the same processing power as collision search on a 256-bit hash function
3. Any attacks require at least the same processing power as key search on a block cipher with 192-bit key
4. Any attacks require at least the same processing power as collision search on a 384-bit hash function
5. Any attacks require at least the same processing power as key search on a block cipher with 256-bit key

These security categories makes it easier to compare performance trade-off with security.

2.8.2 Performance and Cost

The performance and cost of an algorithm is the second most important area when comparing the candidates. This inlcudes but not limited to:

- The size of public and private keys
- Performance of encryption, decryption and keygeneration
- Probability of decryption failure
- Memory usage

The memory usage is a reference to both the size of the algorithm and Random Access Memory (RAM) usage. The hope for the computational efficiency of the algorithms is that it will improve over the currently standardized public key algorithms today. NIST will do preliminary efficiency analysis on a reference platform, but invites the public to do similar performance analysis on different platforms.

2.8.3 Algorithm & Implementation Characteristics

NIST will prefer algorithms with features not present in the current standards in public key encryption. This can be different things like running efficiently on many different platforms, taking advantage of parallelism or implemented with instruction set extensions. They will also look at factors that can hinder the usage of candidate algorithms, like intellectual property and terms of license of the algorithms and implementations.

2.8.4 Candidates

The candidates of round 2 in the NIST PQC project consists of 17 key encapsulation mechanism and/or public key encryption schemes. I will look closer at six of these in chapter 3. These six have in common that they all got a PKE specification. The round 2 project also consists of 9 digital signature algorithms.

2.9 PKE and KEM

Public Key Encryption (PKE) and Key Encapsulating Mechanism (KEM) are terms that will be used a lot in this thesis. A PKE scheme is a scheme that can encrypt and decrypt an arbitrary message. A system like this can either be a hybrid solution, where a KEM and Data Encapsulating Mechanism (DEM) is used, or a PKE primitive. A PKE primitive are algorithms that directly encrypt and decrypt data without using a DEM. PKE primitives are often not IND-CCA2 secure, and are therefore converted to an IND-CCA2 secure KEM with the conversion specified in [9].

2.9.1 PKE Primitives Advantages

PKE primitives are often more efficient compared to the KEM version when transferring small amount of data to a recipient. This advantage disappears when the amount of data increases, as it will be more efficient to encrypt with a symmetric key encryption scheme. A PKE primitive can also achieve IND-CPA security, which makes it more flexible by being able to be converted to a IND-CCA2 secure KEM.

2.9.2 KEM Advantages

A KEM can achieve IND-CCA2 security, which is considered the security standard by NIST in the PQC project. It can also simply be converted to a hybrid scheme with a symmetric key encryption scheme. Public key cryptography is mostly used for key distribution, which is exactly what a KEM does.

2.10 Quantum Computers

Quantum computers have been a hot topic in the recent years. Google claim they have reached quantum supremacy with their programmable superconducting processor [10]. A quantum computer works by taking advantage of the quantum mechanical phenomena that happens at the microscopic scale. For quantum mechanics to take affect you often have to work with atoms, electrons and photons.

A classical computer stores information in bits, which can be represented by 0 or 1. A quantum computer stores information in qubits. Qubits can be represented as 0 or 1, but can also be in a superposition. When a qubit is in a superposition, it is both 0 and

1 simultaneously. This can be represented like this:

$$|\Psi\rangle = \alpha * |0\rangle + \beta * |1\rangle$$

When we observe the qubit, the system collapses to either $|0\rangle$ or $|1\rangle$. The probability of collapsing to each depend on α and β. This gives the property $\alpha^2 + \beta^2 = 1$. With multiple qubits, it is possible to entangle the qubits. This means the qubits entangled can not be described independently. Let's take Bell State as an example:

$$\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$$

If one of the qubits is measured to 0 or 1, then we know the result of the other qubit to be the same. A controlled NOT gate (or CNOT) is often used to maximize the entanglement between qubits. The hadamard gate is often used to get a qubit in superposition. These gates and many other can be used to create an algorithm, where the goal is to maximize the probability for the qubits to collapse to the correct result. A quantum computer often have to run an algorithm multiple times to make sure the qubits do not collapse to the wrong result.

2.11 Shor's Algorithm

Shor's algorithm is the reason NIST created the post-quantum cryptography project and why modern public key cryptography is not considered secure against reasonable sized quantum computers. The algorithm is design for doing integer-factorization efficiently (in $O(log\ N)$ time). Shor's algorithm works by finding the period for a function:

$$f(x) = a^x \mod N$$

Where a is a random number and N is the number to be factored. When the order r is found where $f(x + r) = f(x)$ and is the smallest positive even integer, then we can find a factor of N by $gcd(a^{r/2} + 1, N)$ or $gcd(a^{r/2} - 1, N)$ as long as $a^{r/2} \not\equiv -1 \pmod{N}$. If some of the condition does not meet, it is just to start the algorithm again with another random number.

2.12 Grover's Algorithm

Grover's algorithm is the reason it is recommended for most cryptographic algorithms to increase key sizes. The algorithm is often considered a searching algorithm where it is possible to search through N unsorted items in $O(\sqrt{N})$ time. This means it is possible to solve a brute-force attack with a quadratic speedup compared to the classical counterpart, as a brute-force attack is just a search for the correct key/information. In theory this reduces the security of a cryptographic algorithm with 2^{128} steps for brute-fore to 2^{64} steps, and 2^{256} steps to 2^{128} steps.

2.13 FPGA

FPGA stands for Field-Programmable Gate Array, which is programmable logic arrays. An FPGA can be used for improving the performance of an algorithm through a logic circuit. This opens up for possibility for public key encryption for being implemented in a parallel fashion. Some public key encryption schemes are already implemented for FPGA, like McEliece [11] and RSA [12]. The difference of FPGA from other gate arrays is the possibility to program the gates and ease of use. A program can be written in hardware description language, loaded onto the FPGA-hardware and work. The major difference of programming of an FPGA and software is that FPGA programming is about designing logic circuits, while software programming is about executing code sequentially. An algorithm or parts of an algorithm rewritten to an FPGA usually have a substantial boost to performance.

An FPGA is interesting to look at because an implementation of a cryptosystem can be optimized by using a software/hardware solution between a computer and an FPGA. The idea is that the most time consuming functions and operations can mostly be implemented in the FPGA, while the rest of the program runs on the CPU. An example done on LightSaber (which is a candidate in NIST PQC project) found 9x speedup [13].

2.14 Performance Analysis of Data Encryption Algorithms

A possibility for analysing the performance of public key encryption scheme is to run the algorithm while measuring the time it takes. This will give an indication of how the PKE scheme performs on that specific system. As there are a lot of factors that affect the performance of a PKE scheme, it is not possible to make a performance analysis of one system and say it will perform just the same on other systems with other specifications. This is why it is important to do performance analysis on multiple systems with different specifications.

Even in a single computer there are a lot of factors that affect the performance of the algorithms. Things like clock-speed of the CPU and RAM, available cache, different optimizations, other processes running and many other factors will affect the performance to varying degrees. Some of these factors will be interesting to make an analysis of their impact on performance of the PKE algorithm. Other factors might not be as interesting to look at, such as the impact other processes. A possibility to lower this factor is to time measure the system multiple time, and take the median of these results.

2.14.1 Finding the Biggest Time Consumers in the Implementation

Finding the biggest time consumers in an algorithm is meant by measuring the time each method and function uses of the total. This can be done by proprietary programs. The goal of this is to find the bottleneck of the algorithm, which can then be optimized. A way to do this is to use software like *Gprof* for Linux.

2.15 SUPERCOP

SUPERCOP stands for System for Unified Performance Evaluation Related to Cryptographic Operations and Primitives. The goal of SUPERCOP is to measure the performance of different cryptographic algorithms on different type of systems to give developers of cryptosystem an idea of the real performance of their implementations. These cryptosystems vary from hash functions, secret-key stream ciphers, public-key encryption systems, public-key signature systems, and public-key secret-sharing systems. SUPERCOP is divided so that eBATS (ECRYPT Benchmarking of Asymmetric Systems) handles the benchmarking of PKE schemes, KEM and signatures. Some of the candidates are implemented in the SUPERCOP system already, and would therefore give a good representation of the performance for these cryptosystems compared to my own computer.

2.16 Hybrid Public Key Encryption Scheme/Composite Mode

A hybrid public key encryption scheme is a scheme combining a post-quantum PKE or KEM scheme with a modern PKE or KEM scheme. This type of scheme is a temporary solution because of the lack of standardized post-quantum PKE and KEM schemes. It is not to be confused with hybrid cryptosystem, which is a combination of a KEM and symmetric key encryption to create a public key encryption. The idea of hybrid PKE scheme is to combine the schemes so potential flaws in either can be backed up by the other scheme. In a case where the security of the post-quantum PKE scheme were not good enough after some cryptanalysis or side channel attack, then you still have a fallback solution like RSA- or ECC-encryption. This solution will clearly be slower than each solution alone, but for critical information this solution is the most secure. Some of the candidates will be better suited as part of a hybrid public key encryption. Size of ciphertext and maximum size of plaintext will be a large factor for how well each candidates is suited.

A potential solution will be described below. This is just a possible solution of how to combine two PKE schemes without the use of a symmetric encryption scheme. The idea is to use one of the PKE schemes to encrypt the ciphertext of the other PKE scheme. The interesting part when designing a solution like this is which algorithm is going to be the Outer-Level Encryption (OLE) and Inner-Level Encryption (ILE). The outer-level encryption is encrypting the inner-level ciphertext. There might be both security concerns and performance differences by doing it either way. With regard to performance it would make most sense to use the algorithm with the least expansion from the data to the ciphertext as the inner-level encryption. This means there will not be as much to encrypt for the outer-level encryption algorithm to encrypt. Compared to the other way around where the OLE algorithm have to encrypt more data because the larger expansion from the ILE algorithm. Of course, this assumes that the encryption speed of each PKE scheme are somewhat similar, which is not necessarily the case. If the size of ciphertext is not equal to multiple of the plaintext size of the OLE encryption scheme, then padding might be needed. A wrong type of padding might cause security concerns. This type of implementation is just an example of only using the PKE schemes. A maybe better solution is to use two KEM and XOR the shared secret from each KEM together. Many of the candidates for the NIST PQC expand the ciphertext noticeable more than RSA or ECDH does, and would therefore most likely be a better OLE scheme.

2.17 GNU Compiler Collection

GNU Compiler Collection, or GCC for short is a compiler for multiple programming languages, where C and C++ are the most notable. Created as the official compiler for GNU operative system, it has been adopted by other Unix-like operating systems, like Linux. There is also a version of GCC in Windows. The portability is the reason why I will use GCC as the compiler for the project. GCC is often considered faster in many cases than a similar C compiler in many operative systems [14]. This is not always the case and if someone want to get the best performance in their operative system, they should test each compiler.

2.17.1 Optimizations with GCC

There are different types of optimization that can be done when compiling code with GCC. The goal can be to optimize speed, or to reduce the size of the compiled code. These are seperated in levels of optimizations with the option **-O** when compiling.

O1 - The O1 option or level 1 optimization has the goal of optimizing the code in a short amount of time. The optimization that are done might increase memory usage and compilation time, but in return the speed is increased and the compiled code size is decreased.

O2 - The O2 option or level 2 optimization enables all speed optimization that does not increase the compiled code size. This means there is more focus on optimizing the speed at the cost of extra compile time compared to level 1. All optimizations done in level 1 is also done for level 2.

Os - The Os option or often called size optimization. As the name says it is focused on optimizing the size of the compiled code. This option does many similar optimizations as level 2 optimization but focuses on decreasing the size as number one priority.

O3 - O3 or level 3 optimization is focused exclusively on optimizing the speed. The trade-off with level 3 optimization is that the compiled code size will increase. The optimization of inline function is usually a large contributor to the compiled code size increase.

The usual flag in use in production of software is the O2 flag. The reason for this is that application usually get quite large so that size becomes a limitation because of bandwidth or other reason, including to the limitation of the CPU cache size. It is however possible to compile parts of the program with the O3 flag for better optimization.

2.17.2 Static Library and Linking

Static libraries will be used for linking the implementations of the public key cryptosystems. This is done to avoid the small overhead of shared library that can appear when calling the dynamically linked functions. However this will result in a larger compiled code size.

When linking functions in the static libraries there may arise some problems because of the similarly named functions. This can be avoided by prefixing names of functions in each static library to something that identifies that specific library. It is also possible to use namespaces in C or C++. This especially a problem with candidates of the NIST PQC project as these have many similar but often small differences in functions.

2.17.3 Assembly Sections

In C and C++ it is possible to create section in the code written directly in assembly. If a competent developer writes a function in assembly there is a possibility to increase the performance over fully optimized C or C++ code. This is however not a portable solution as ANSI C code. For a developer with little experience in assembly it is not recommended as the compiler will usually do a better job at optimizing.

2.18 Optimization of C/C++

Optimization can be divided in three parts, compute-bound, memory-bound and IO-bound. Optimizations past compiler optimization are usually not worth the time for a software development team unless it is a critical part of the software or related to UI-performance. It is usually more worth to upgrade the hardware with better specifications. In the case of public key encryption there is an argument for making performance optimization since it is the backbone to all encrypted communication between computers.

2.18.1 Compute-Bound Optimization

Compute-bound optimization is optimization of the function that uses the most CPU-time. This can be done by running your program on real world data, and create a report with information of the CPU runtime for each function in you program. This can be created by tools like Gprof which creates a report if the correct flags are used in compilation. The reason this is done is to find software-bottlenecks of the program.

Choose Efficient Algorithms

Choosing algorithms that are the most efficient can give a big performance boost to the program. E.g. If you have a large sorted dataset, then it would be wastefull of resources to use sequential search instead of a binary search. This is because sequantial search have a runtime of $O(n)$ while binary search has a runtime of $O(log(n))$. Although the different Big-O notations are not directly possible to convert to performance increase or decrease, it is a good measurement of how your program will run if the input size increases.

Simple Code

Writing simple code may come as a surprise to some, but it is important to make the code readable for the compiler. This is because most of the optimization is done by the computer. If there are complicated expressions, the compiler might have trouble to understand the code and turn off some of the optimizations.

Efficient Code

When writing a program, make sure operations are not unnecessarily computed multiple times and that the operations are used correctly. An example can be to reduce the number of multiplication in a loop:

```
void slowexample() {
    int a = 42, b = 233, c = 281;
    int arr[100000];

    for (int i = 0; i < 100000; ++i) {
        arr[i] = a*b*c*i;
    }
}

void fastexample() {
    int a = 42, b = 233, c = 281;
    int arr[100000];
    int temp = a*b*c;

    for (int i = 0; i < 100000; ++i) {
        arr[i] = temp*i;
    }
}
```

It is important to be aware of what operations that take time. Things like reading and writing to a file and operations on entire arrays. This includes operations that are fast but are used in loops, especially if the loop has many iterations, like in the example.

Compiler Options

Compiler options like the O flags in the earlier section is also an important part of optimizing. Be sure that the correct optimization flags are used for the program. GCC have the possibility to insert `#pragma` section like `inline` for critical parts of the code to increase the performance.

Other Compute-Bound Optimizations

There are a lot of other kinds of compute-bound optimizations that can be done, but the most important ones are mentioned above.

2.18.2 Memory-Bound

Memory-bound optimization is about optimizing the features and operations related to the memory and cache. We have already discussed the fact that too many inline function might fill up the instruction cache and lead to performance drop as a result of the need to read to slower memory. Some other examples of memory-bound optimizations are:

- Iterate through columns before rows in a 2d array.
- Locality in memory of often used variables.
- Use reference instead of copying large structs and arrays.
- Iterate forward instead of backwards through an array to make sure the cache is able to ”predict” the correct addresses that will be accessed next.

2.18.3 IO-Bound

IO-bound optimizations are optimization for input and output to files, network and console. I will not focus on these optimization as they can be tricky to do and not much sense to go through in this thesis.

2.19 ANSI C Code

ANSI C is a standard of the C language published by the American National Standards Institute (ANSI). The standard is the specified standard NIST wants submitters in the post-quantum cryptography project to submit their reference code in. The reason the standard is recommended for developers writing in C is to help with portability between compilers.

2.20 Instruction Sets

The instruction sets available to a computer will affect the performance of algorithms running on the computer. Most of the candidate algorithms have implementations using binary operations. For the optimized solution of these operations, many have decided to use specialized instruction sets such as AVX. The most common instruction set standard used for computers today is the x86 with 64 bits. The x86 instruction set can come with extensions for inter alia SSE, AES-NI, AVX and AVX2 instructions.

2.20.1 AVX Instruction Set

AVX stands for Advanced Vector Extension and is an extension of the x86 instruction set. AVX is used for simultaneous operations on several floating point numbers. It performs Single Instruction on Multiple pieces of Data (SIMD) [15]. The AVX instruction set has the ability to do operations on eight 32-bit or four 64-bit floating point numbers. The AVX2 instruction set expands available operations to work on 256-bit data types. The 256-bit data type can be considered a single long integer. Therefore AVX2 can be used to be an optimized solution for operations on larger numbers. The AVX512 extends this to 512-bit integers, or 16x 32-bit floating point numbers. Working with the floating point numbers, it could be considered a way to parallelize a program on a single core, as operations on all floating point number will be done in a single instruction on the CPU. The data-types used for AVX and AVX2 in C and C++ are `__m256`, `__m256d` and `__m256i` which is the datatypes corresponding to 8x 32-bit floating point numbers, 4x 64-bit floating point numbers and 256-bit integer respectively. SSE is the same but for 128-bit data, which means operations on 4x 32-bit floating point numbers or other types of data.

2.20.2 AES-NI

AES-NI stands for Advanced Encryption Standard New Instructions and are instructions for doing optimized operation of the standardized symmetric key encryption scheme AES. These instructions are used by some of the candidates implementations through OpenSSL. A possibility when the NIST PQC project is finished and candidates have been chosen, is to implement instruction set to optimize the performance of these algorithms.

2.21 Finite Fields

Since finite fields is an important part of most of the candidate cryptosystems, I will explain it and possible optimization of it. Finite fields are fields of elements where addition, subtraction, multiplication and division is defined such that these operation on each element ends up as another element in the field. The most common finite fields are integers in modulus p where p is a prime. This finite field have order p because it has p elements. When trying to create a finite field over a multiple of a prime, Galois Fields are needed. This is because if the elements are represented as integers, not all elements have a valid inverse in the field. A finite field can be denoted as $\mathbb{F}_{p^n}$. If we look at an example, a finite field $\mathbb{F}_{p^n}$ where $n > 1$, this field may be constructed from an irreducible polynomial $P(X)$ in $\mathbb{F}_p[X]$ with degree n. The field can be written as:

$$\mathbb{F}_{p^n} = \mathbb{F}_p[X]/(P(X))$$

Finite fields of characteristics 2 is finite fields where the prime in use is 2. These finite fields are what most of the candidates of the NIST PQC project are implemented in. One of the reason finite fields of characteristics 2 is so much used is the structure mirrors the binary structure of computers, and therefore makes binary operations well suited on these fields. There are several libraries made to optimize operations over $\mathbb{F}_{2^n}$ finite fields. Addition and subtraction are identical to the XOR operation of a computer, and therefore very efficient of itself. Most libraries focus on improving the performance of multiplication in these fields.

2.21.1 Multiplication over Finite Fields

There are several ways of optimizing multiplication in finite fields. I will only focus on optimization of finite fields with characteristics 2. Libraries like gf2x and NTL have implemented fast routines for arithmetics in $\mathbb{F}_{p^n}$. These libraries are used by some of the candidates in the NIST PQC. There are also implementations of multiplication over finite fields in hardware like FPGA and ASIC [16].

Modular reduction in a multiplication can be efficiently done by using Montgomery and Barret reduction techniques. These reduction techniques works best at low Hamming weight modular polynomials. So it is important to find low hamming weight irreducible polynomials for the finite fields. For $\mathbb{F}_{2^n}$ where $n < 2048$ it is possible to find irreducible polynomials with order n and Hamming weight either 3 or 5. It is possible to create optimal reduction algorithms which can outperform Montgomery reduction as n increases.

2.22 Quantum Key Distribution

A quantum key distribution is a contender for key distribution between computers. It provides a way for two parties to communicate keys with the possibility of recognising someone eavesdropping by taking advantage of properties in quantum physics. These property are the quantum superposition and quantum entanglement. The idea is that if a third party tries to observe the information in transit, they will have to collapse the superposition to observe the data, which will affect the error rate of the two parties communicating. Although a quantum key distribution would cause extra complexity to encryption process, it is proven to be secure [17].

2.23 RSA

Rivest Shamir Adlema (RSA) is a standardized public key cryptosystem for both public key encryption and signature. RSA is considered broken when it comes to quantum computers running Shor's algorithm. The system is based on the integer factorization problem and the RSA problem. Although RSA is not considered the most efficient public key encapsulating compared to other schemes based on Elliptic Curve Cryptography (ECC), it is one of the most widely used because of how easy it is to understand and the lack of patents on the system.

Security

RSA has been around since 1977 and several cryptanalysis trying to find weaknesses in the system. As RSA is based on the mathematical problems of integer factorization and RSA problem, there is no known efficient algorithms for fully breaking RSA until a large enough quantum computer becomes a reality. However there are weaknesses if wrong parameters and methods are used. An important part of RSA encryption is to pad the data correctly. There are several known attacks on plain RSA, such as the Coppersmith's attack. Plain RSA is susceptible to both chosen plaintext attack and chosen ciphertext attack.

Parameters

Due to padding the keys and plaintext of RSA does not have a fixed size. The ciphertext size is the same as the modulus size n, which vary from 1024 to 15360 bits. For 128-bit security or security level 1, NIST recommends to use 3072 bits. NIST further recommends 15360-bit keys for 256-bit security, or security level 5. The plaintext of 2048-bit RSA is according to Public Key Cryptographic Standards (PKCS) #1 maximum 245 bytes.

Chapter 3

Candidate Cryptosystems

3.1 ROLLO

ROLLO submission is the result of combining three candidates from round one in the NIST Post-Quantum Cryptography project. These candidates were

- LAKE, renamed to ROLLO-I
- LOCKER, renamed to ROLLO-II
- Rank-Ouroboros, renamed to ROLLO-III

The ROLLO algorithms are all based on rank metric codes and share the same decryption algorithm. ROLLO-I and ROLLO-III are key encapsulating algorithms, while ROLLO-II is a public key encryption algorithm. I will focus mostly on ROLLO-II in this thesis. ROLLO-I and ROLLO-III are categorized as IND-CPA KEM. While ROLLO-II is categorized as IND-CCA2 PKE.

The way ROLLO has overcome the problem of a too large public and private key in code based cryptography, is to use *ideal codes*. Ideal codes are a family of codes where the generator matrix is in systematic form with blocks of ideal matrices. The ideal codes only differ from Quasi-Cyclic codes by the irreducible polynomial P of the finite field $\mathbb{F}_{q^m}[X]/\langle P \rangle$. Quasi-Cylic codes have the irreducible polynomial as $< x^n - 1 >$ while ideal codes might have other types of irreducible polynomials.

ROLLO uses a version of Low Rank Parity Check (LRPC) codes like McEliece, since they have weak algebraic structure. However this is something that could use more study according to a report of round 1 submission [8]. The submitters define the ideal LRPC codes to be used by the cryptosystem.

A new version of ROLLO was given after the deadline of submitting new material to round 2 in the NIST PQC project. This version fixes security concerns that I will go into in the security section. The new submission increases the parameters of the scheme which most likely decreases performance of the system with the updated parameters. However the implementation in the new submission makes it more understandable for 3rd parties, and might have some extra optimizations that was not included in the original submission.

3.1.1 How It Works

The bold variables are vectors. The figure 3.1 might be confusing, but is quite easy to

- KeyGen(1^λ): Picks $(\boldsymbol{x}, \boldsymbol{y}) \overset{\$}{\leftarrow} S_d^{2n}(\mathbb{F}_{q^m})$. Sets $\boldsymbol{h} = \boldsymbol{x}^{-1}\boldsymbol{y} \mod P$ and return pk $= \boldsymbol{h}$, sk $= (\boldsymbol{x}, \boldsymbol{y})$.
- Encrypt(M, pk): Picks $(\boldsymbol{e}_1, \boldsymbol{e}_2) \overset{\$}{\leftarrow} S_r^{2n}(\mathbb{F}_{q^m})$, sets $E = Supp(\boldsymbol{e}_1, \boldsymbol{e}_2)$, $\boldsymbol{c} = \boldsymbol{e}_1 + \boldsymbol{e}_2\boldsymbol{h} \mod P$.
  Computes $cipher = M \oplus \boldsymbol{Hash}(E)$ and returns the ciphertext $C = (\boldsymbol{c}, cipher)$.
- Decrypt(C, sk): Sets $\boldsymbol{s} = \boldsymbol{x}\boldsymbol{c} \mod P$, $F = Supp(\boldsymbol{x}, \boldsymbol{y})$ and $E \leftarrow \boldsymbol{RSR}(F, \boldsymbol{s}, r)$.
  Return $M = cipher \oplus \boldsymbol{Hash}(E)$.

Figure 3.1: Figure from ROLLO-II documentation [18].

understand when the symbols are clear what they mean. The symbol $S_w^n(\mathbb{F}_{q^m})$ is the Set of vector with weight w and length n in the field $\mathbb{F}_{q^m}$. So the equation $(\boldsymbol{x}, \boldsymbol{y}) \overset{\$}{\leftarrow} S_d^{2n}(\mathbb{F}_{q^m})$ means that we choose with uniform randomness two vectors from the set. In the encryption we choose two error vectors $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$. *Supp* is the support function, which means we find the support subspace of the error vectors. The hash value of this subspace is XORed with the message and appended with $\boldsymbol{c} = \boldsymbol{e}_1 + \boldsymbol{e}_2\boldsymbol{h}$.

The decryption starts by calculating $\boldsymbol{s} = \boldsymbol{x}\boldsymbol{c} \mod P$. This equation becomes: (The multiplication of vectors are commutative)

$$\boldsymbol{s} = \boldsymbol{x}\boldsymbol{c} = \boldsymbol{x} * (\boldsymbol{e}_1 + \boldsymbol{e}_2\boldsymbol{h}) = \boldsymbol{x} * (\boldsymbol{e}_1 + \boldsymbol{e}_2\boldsymbol{x}^{-1}\boldsymbol{y}) = \boldsymbol{e}_1\boldsymbol{x} + \boldsymbol{e}_2\boldsymbol{y} \mod P$$

With this result, we can find the support of $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$ with an algorithm called Rank Support Recovery (RSR). When this support is found, the hash of it is XORed with the cipher. The ROLLO-II PKE works by using the KEM from ROLLO-I as a one-time pad. The shared secret E is hashed, and XORed with the message.

3.1.2 Security

The difficult problems ROLLO is based on are particular cases of Rank Syndrome Decoding (RSD) problem [19] and Ideal LRPC codes indistinguishability. More precisely it is only ROLLO-I and ROLLO-II that are based on the RSD problems, ROLLO-III only require assumption of Ideal LPRC codes indistinguishability. The primary difference of ROLLO-II to the others is that it claims to have IND-CCA2 security, while the others only claim IND-CPA security. According to NIST it would be valuable for more cryptanalysis against rank-based primitives like ROLLO, especially algebraic attacks on RSD and LRPC key recovery [8].

As of writing this thesis, the security of ROLLO-I, ROLLO-II and ROLLO-III has been broken by [20]. The article mostly focuses on ROLLO-I. This causes ROLLO-I-128, ROLLO-I-196 and ROLLO-I-256 to not achieve the claimed security levels 1, 3 and 5 respectively. The bit security is considered by this new attack to be 70, 86 and 158 for each. The attack is based on a similar attack in [21]. The reduction in security is caused by a term in the exponential exponent when calculating the complexity of breaking the system. The term is in $\mathcal{O}(n^2)$ for a plain RSD. This is reduced to a term in $\mathcal{O}(n^{\frac{3}{2}})$ for [21] and to a term in $\mathcal{O}(n)$ for [20]. Differences between [20] and [21] is that [20] do not rely on Gröbner Basis calculations. These attacks are grouped as algebraic attacks.

Examples for new parameters are included in [20] to make sure security levels are maintained for ROLLO-I. This attack also affects ROLLO-II and ROLLO-III according to table 6.1, but the paper [20] does not focus on them and do not give any examples of new parameters like they do for ROLLO-I. A few days later they submitted updated parameters and implementations for ROLLO-I and ROLLO-II. They decided to only focus on ROLLO-I and ROLLO-II for simplicity and leave out ROLLO-III.

3.1.3 Parameters

The parameters for each algorithm is chosen in function of the best known attack on the hard problems each depend on to achieve the claimed security category. The sizes are derived from the parameters.

ROLLO-I

Old parameters:

Security	q	n	m	r	d	P	DFR
1	2	47	79	5	6	$X^{47}+X^5+1$	2^{-30}
3	2	53	89	6	7	$X^{53}+X^6+X^2+X+1$	2^{-32}
5	2	67	113	7	8	$X^{67}+X^5+X^2+X+1$	2^{-42}

Security category	pk-size	sk-size	ct-size	ss-size
1	465	40	465	64
3	590	40	590	64
5	947	40	947	64

New parameters:

Security	q	n	m	r	d	P	DFR
1	2	83	67	7	8	$X^{83}+X^7+X^4+X^2+1$	2^{-28}
3	2	97	79	8	8	$X^{97}+X^6+1$	2^{-34}
5	2	113	97	9	9	$X^{113}+X^9+1$	2^{-33}

The new parameters are the ones proposed from the updated submission. In [20] they do not present new P values, but in the updated submission of ROLLO they give the new official parameters. The irreducible polynomials needs to be in $\mathbb{F}_2$ with degree n. The computational cost of the system depends on the value of P. To generate a desirable P with degree n, the Magma software can be used [22]. The updated parameters in [23] gives an n that is generally smaller and m is generally larger compared to the example parameters from [20]. The reason they decided not to use the example parameters can be a combination of reasons like performance, security and size.

Security	pk-size	sk-size	ct-size	ss-size
1	696	40	696	64
3	958	40	958	64
5	1371	40	1371	64

The public key size of the new parameters are described in [23]. It is also possible to compute them from the equation $\lceil nm/8 \rceil$. The ciphertext size is equal to the public key size. The size of the example parameters proposed in [20] were larger than the new parameters, suggesting they were optimized for size.

ROLLO-II

The parameters for ROLLO-II are generally much larger than ROLLO-I. In return the DFR is much smaller, which makes it possible to achieve IND-CCA2 security. We will first look at the original parameters proposed in [18], and then the new parameters from [23].

Old parameters:

Security	q	n	m	r	d	P	DFR
1	2	149	83	5	8	$X^{149} + X^{10} + X^9 + X^7 + 1$	2^{-128}
3	2	151	107	6	8	$X^{151} + X^3 + 1$	2^{-128}
5	2	157	127	7	8	$X^{157} + X^6 + X^5 + X^2 + 1$	2^{-132}

Security	pk-size	sk-size	ct-size	pt-size
1	1546	40	1674	64
3	2020	40	2148	64
5	2493	40	2621	64

The size of the public key kan be calculated the same way as in ROLLO-I. By calculating $\lceil nm/8 \rceil$. The ciphertext is the same, but appended with a SHA512 result, which is 128

bytes. In [20] these parameters are broken and does not achieve the claimed security level.
New parameters:

Security	q	n	m	r	d	P	DFR
1	2	189	83	7	8	$X^{189} + X^6 + X^5 + X^2 + 1$	2^{-134}
3	2	193	97	8	8	$X^{193} + X^{15} + 1$	2^{-130}
5	2	211	97	8	9	$X^{211} + X^{11} + X^{10} + X^8 + 1$	2^{-136}

Security	pk-size	sk-size	ct-size	ss-size
1	1941	40	2089	64
3	2341	40	2469	64
5	2559	40	2687	64

These parameters and sizes are from [23]. In [20] they do not give example parameters for ROLLO-II. The new parameters have generally increased, however m has decreased for security level 3 and 5 resulting in the size of the public key not increasing as much as for ROLLO-I.

ROLLO-III

ROLLO-III is not included in the new submission of ROLLO in [23]. The submitters did this for simplicity sake. The parameters below are from the old submission [18]:

Security	q	n	m	w	w_r	P	DFR
1	2	47	101	5	6	$X^{47} + X^5 + 1$	2^{-30}
3	2	59	107	6	8	$X^{59} + X^7 + X^4 + X^2 + 1$	2^{-36}
5	2	67	131	7	8	$X^{67} + X^5 + X^2 + X + 1$	2^{-42}

Security	pk-size	sk-size	ct-size	ss-size
1	634	40	1188	64
3	830	40	1580	64
5	1138	40	2196	64

The security/complexity bits of each security level are 69.5, 88.0 and 137.7 for security level 1, 3 and 5 respectively. This means ROLLO-III-256 could be used as a cryptosystem with claimed security level 1. However it is most likely possible to optimize the parameters further.

3.1.4 Intellectual Property

The schemes of ROLLO-I and ROLLO-II are by the authors not intended to hold any patents on the cryptosystems. The ROLLO-III cryptosystem is patented but the authors agreed to give a non-exclusive license for implementing the system if it were to be standardized. This would be done without compensation and under reasonable terms and conditions [24].

3.2 RQC

Rank Quasi-Cyclic (RQC) is an IND-CCA2 KEM which can also be used as a PKE. It has the same submitters as ROLLO. RQC is based on coding theory with several desirable properties according to the publishers. Some features of RQC are:

- Proven to achieve IND-CPA assuming the hardness of the Syndrome Decoding problem.
- The assumption that all families of codes being used is indistinguishable is not required.
- The Decryption Failure Rate (DFR) is zero, because the decryption algorithm is deterministic.
- Attractive parameters compared to most Hamming based proposals.

The family of codes used by RQC is the Gabidulin codes. These codes are essentially the only known codes in rank metric that have an efficient deterministic decoding algorithm up to a given rank error weight. This means the decryption failure rate is zero. They can decode up to $\lfloor \frac{n-k}{2} \rfloor$ errors in a deterministic way [25]. Features like this, the target of IND-CCA2 security and the fact that the system is not dependant on the ideal LRPC codes indistinguishability leads the system to suffer in performance and size of the implementation compared to ROLLO.

RQC is broken from the same attack as for ROLLO in [20]. This means the parameters in the submission in [25] to NIST is not secure. New parameters was published in [26] after the deadline for new material in round 2 in the NIST PQC project. The new submission comes with an updated implementation using the same library as ROLLO for handling rank based finite field elements, called Rank Based Cryptography (RBC) Library.

3.2.1 How It Works

The IND-CCA2 KEM version of the scheme is a Fujisako-Okamoto transformation of the PKE scheme [9]. The symbols in Figure 3.2 have the same meaning as in Figure 3.1. $\mathcal{G}_g$ is the Gabidulin codes that is used for the decoding and encoding of the message. The code can correct up to δ errors. The generator matrix $\boldsymbol{G}$ of the code $\mathcal{G}_g$ is publicly known. This is the specification in the updated submission of RQC, as w_1 and w_2 has replaced w_r.

- Setup(1^λ): generates and outputs the global parameters $\boldsymbol{param} = (n, k, \delta, w, w_r, P)$ where $P \in \mathbb{F}_q[X]$ is an irreducible polynomial of degree n.
- Keygen(param): samples $\boldsymbol{h} \xleftarrow{\$} \mathbb{F}_{q^m}^n$, $\boldsymbol{g} \xleftarrow{\$} S_n^n(\mathbb{F}_{q^m})$ and $(\boldsymbol{x}, \boldsymbol{y}) \xleftarrow{\$} S_{1,w}^{2n}(\mathbb{F}_{q^m})$, computes the generator matrix $\boldsymbol{G} \in \mathbb{F}_{q^m}^{k \times n}$ of $\mathcal{G}_g(n, k, m)$, sets pk $= (\boldsymbol{g}, \boldsymbol{h}, \boldsymbol{s} = \boldsymbol{x} + \boldsymbol{h} \cdot \boldsymbol{y} \mod P)$ and sk $= (\boldsymbol{x}, \boldsymbol{y})$, returns (pk,sk).
- Encrypt(pk,$\boldsymbol{m}$,θ): uses randomness θ to generate $(\boldsymbol{r}_1, \boldsymbol{e}, \boldsymbol{r}_2) \xleftarrow{\$} S_{(w_1,w_2)}^{3n}(\mathbb{F}_{q^m})$, sets $\boldsymbol{u} = \boldsymbol{r}_1 + \boldsymbol{h} \cdot \boldsymbol{r}_2 \mod P$ and $\boldsymbol{v} = \boldsymbol{mG} + \boldsymbol{s} \cdot \boldsymbol{r}_2 + \boldsymbol{e} \mod P$, returns $\boldsymbol{c} = (\boldsymbol{u}, \boldsymbol{v})$.
- Decrypt(sk,$\boldsymbol{c}$): returns $\mathcal{G}_g$.Decode($\boldsymbol{v} - \boldsymbol{u} \cdot \boldsymbol{y} \mod P$).

Figure 3.2: Presentation of RQC.PKE from RQC documentation [25]. Bold variables are vectors.

When generating $(\boldsymbol{r}_1, \boldsymbol{e}, \boldsymbol{r}_2) \xleftarrow{\$} S_{(w_1,w_2)}^{3n}(\mathbb{F}_{q^m})$ in encryption, it means that $wt(\boldsymbol{r}_1, \boldsymbol{r}_2) = w_1$, $wt(\boldsymbol{e}) = w_1 + w_2$ and $Supp(\boldsymbol{r}_1, \boldsymbol{r}_2) \subset Supp(\boldsymbol{e})$. For the decryption, the equation $\boldsymbol{v} - \boldsymbol{u} \cdot \boldsymbol{y}$ can be extended to:

$$\boldsymbol{v} - \boldsymbol{u} \cdot \boldsymbol{y} = \boldsymbol{mG} + \boldsymbol{s} \cdot \boldsymbol{r}_2 + \boldsymbol{e} - (\boldsymbol{r}_1 + \boldsymbol{h} \cdot \boldsymbol{r}_2) \cdot \boldsymbol{y}$$

$$= \boldsymbol{mG} + (\boldsymbol{x} + \boldsymbol{h} \cdot \boldsymbol{y}) \cdot \boldsymbol{r}_2 + \boldsymbol{e} - (\boldsymbol{r}_1 + \boldsymbol{h} \cdot \boldsymbol{r}_2) \cdot \boldsymbol{y}$$
$$= \boldsymbol{mG} + \boldsymbol{x} \cdot \boldsymbol{r}_1 - \boldsymbol{y} \cdot \boldsymbol{r}_2 + \boldsymbol{e}$$

In this equation, it is $\boldsymbol{x} \cdot \boldsymbol{r}_1 - \boldsymbol{y} \cdot \boldsymbol{r}_2 + \boldsymbol{e}$ which is considered the error. The hamming weight of this error is always equal or smaller than δ. This causes the decoding with the Gabidulin code to be deterministic.

3.2.2 Security

The RQC cryptosystem targets to be IND-CCA2 secure. The security of RQC is based on the Decisional Ideal Rank Syndrome Decoding (Decision IRSD) problem. There is no direct proof of this ideal case, but the best known attacks on the Decision IRSD are direct attacks on the search version of the problem [25]. As with ROLLO according to NIST there would be value for more cryptanalysis of rank-based primitives and especially of the algebraic attack targeting RSD.

In [20] the submitters of a scheme discovered that it was possible to increase the effectivness of an algebraic attack the same way they did with ROLLO. The attack is based on the rank decoding problem without using Gröbner basis. This means the parameters for each security level is broken. The attack causes RQC-128 to have a bit security of 77, RQC-192 with 101 bits and RQC-256 with 144 bits. Example parameters was proposed in [20] to fix RQC. These parameters was changed when the updated submission of RQC was released in [26].

Combinatorial attacks on the system are less efficient, and there has not been found any quantum speed-up on the best known algebraic attack mentioned.

3.2.3 Parameters

The old parameters are from the original submission of RQC [25]. These parameters have been replaced by the new parameters from [26].

Old Parameters

Security	q	m	n	k	w	w_r
1	2	97	67	4	5	6
3	2	107	101	3	6	8
5	2	137	131	3	7	9

Security	P	Π
1	$X^{67} + X^5 + X^2 + X + 1$	$X^{97} + X^6 + 1$
3	$X^{101} + X^7 + X^6 + X + 1$	$X^{107} + X^9 + X^7 + X^4 + 1$
5	$X^{131} + X^8 + X^3 + X^2 + 1$	$X^{137} + X^{21} + 1$

As you can see RQC share many similar parameters as ROLLO. But generally these parameters are larger to account for IND-CCA2 security and deterministic error correction.

Security category	pk-size	sk-size	ct-size	ss-size
1	853	40	1690	64
3	1391	40	2766	64
5	2284	40	4552	64

New Parameters

Security	q	m	n	k	w	w_1	w_2
1	2	127	113	3	7	7	6
3	2	151	149	5	8	8	8
5	2	181	179	3	9	9	7

The irreducible polynomials P and Π is not specified in the new example parameters in [20]. They are however specified in the new submission [26]. The same way with the P parameter from ROLLO we know that P is supposed to be sparse and can be found with the Magma software [22]. The Π parameter is more tricky as this parameter is not explained how it is found. From what it seems in the document it is also a sparse irreducible polynomial, and can be found in the Magma software.

Security	P	Π
1	$X^{113} + X^9 + 1$	$X^{127} + X + 1$
3	$X^{149} + X^{10} + X^9 + X^7 + 1$	$X^{151} + X^3 + 1$
5	$X^{179} + X^4 + X^2 + X + 1$	$X^{181} + X^7 + X^6 + X + 1$

Security	pk-size	sk-size	ct-size	ss-size
1	1834	40	3652	64
3	2853	40	5690	64
5	4090	40	8164	64

The pk-size is given in [20] but it is possible to calculate by the equation $40 + \lceil nm/8 \rceil$. The ciphertext size is calculated by the equation $2\lceil nm/8 \rceil + 64$. The new sizes of the RQC scheme is around doubled from the original sizes. The number of cycles according to the updated submission have also around doubled from the new parameters, but this is something I will look into.

3.2.4 Intellectual Property

The authors do have some patents on the cryptosystem. If the cryptosystem is standardized in the NIST PQC project, a non-exclusive license for implementing the standard without compensation under reasonable terms is given [27].

3.3 HQC

Hamming Quasi-Cyclic (HQC) is a second round submission in the NIST PQC project by the same submitters as ROLLO and RQC. Like ROLLO and RQC, HQC is based on coding theory. Features of HQC are that it is proven IND-CPA security, and can by construction of a hybrid encryption scheme achieve IND-CCA2 security and good efficiency. There are many generic techniques to make a IND-CPA secure scheme to a IND-CCA2, but since HQC got possible decryption failures, the transformation of Hofheinz et al. [9] has to be used. HQC does not need to rely on the assumption of the family of codes in use is indistinguishable among random codes like many other code based scheme does.

The documentation of HQC feature a decryption failure probability analysis which makes it possible to claim IND-CCA2 security. HQC is using Quasi Cyclic codes which means the hard problems like Decision Syndrome Decoding Problem is converted to QC-code versions of these problems.

Improvements of the scheme was presented in an updated submission after the deadline of round 2 in NIST PQC project. The new submission includes an updated DFR analysis making it possible to reduce the public key size. A new algorithm using Reed-Muller and Reed-Solomon codes are also introduced to reduce the public key and ciphertext sizes further. This version of HQC will be referred to as HQC-RMRS.

3.3.1 How It Works

- Setup(1^λ): generates and outputs the global parameters param $= (n, k, \delta, w, w_{\boldsymbol{r}}, w_{\boldsymbol{e}})$.
- KeyGen(param): samples $\boldsymbol{h} \xleftarrow{\$} \mathcal{R}$, the generator matrix $\boldsymbol{G} \in \mathbb{F}_2^{k \times n}$ of $\mathcal{C}$, sk $= (\boldsymbol{x}, \boldsymbol{y}) \xleftarrow{\$} \mathcal{R}^2$ such that $wt(\boldsymbol{x}) = wt(\boldsymbol{y}) = w$, sets pk $= (\boldsymbol{h}, \boldsymbol{s} = \boldsymbol{x} + \boldsymbol{h} \cdot \boldsymbol{y})$, and returns (pk, sk).
- Encrypt(pk,$\boldsymbol{m}$): generates $\boldsymbol{e} \xleftarrow{\$} \mathcal{R}$, $\boldsymbol{r} = (\boldsymbol{r}_1, \boldsymbol{r}_2) \xleftarrow{\$} \mathcal{R}^2$ such that $wt(\boldsymbol{e}) = w_{\boldsymbol{e}}$ and $wt(\boldsymbol{r}_1) = wt(\boldsymbol{r}_2) = w_{\boldsymbol{r}}$, sets $\boldsymbol{u} = \boldsymbol{r}_1 + \boldsymbol{h} \cdot \boldsymbol{r}_2$ and $\boldsymbol{v} = \boldsymbol{m}\boldsymbol{G} + \boldsymbol{s} \cdot \boldsymbol{r}_2 + \boldsymbol{e}$, returns $\boldsymbol{c} = (\boldsymbol{u}, \boldsymbol{v})$.
- Decrypt(sk,$\boldsymbol{c}$): returns $\mathcal{C}$.Decode($\boldsymbol{v} - \boldsymbol{u} \cdot \boldsymbol{y}$).

Figure 3.3: Figure from documentation of HQC [28].

In figure 3.3 the PKE scheme is presented with how keygeneration, encryption and decryption works. The code $\mathcal{C}$ is an instance of tensor codes for HQC and an instance of concatenated Reed-Muller and Reed-Solomon codes for HQC RMRS. The generator matrix $\boldsymbol{G}$ of $\mathcal{C}$ is publicly available. This is a simplified version of the scheme as it is possible to generate codewords without the generator matrix. The $\mathcal{R}$ is the finite field where elements are represented. In this case the finite field is $\mathcal{R} = \mathbb{F}_2[X]/(X^n - 1)$ since it is Quasi-Cyclic.

The equation $\boldsymbol{v} - \boldsymbol{u} \cdot \boldsymbol{y}$ can be expanded to:

$$\begin{aligned}\boldsymbol{v} - \boldsymbol{u} \cdot \boldsymbol{y} &= \boldsymbol{m}\boldsymbol{G} + \boldsymbol{s} \cdot \boldsymbol{r}_2 + \boldsymbol{e} - (\boldsymbol{r}_1 + \boldsymbol{h} \cdot \boldsymbol{r}_2)\boldsymbol{y}) \\ &= \boldsymbol{m}\boldsymbol{G} + (\boldsymbol{x} + \boldsymbol{h} \cdot \boldsymbol{y}) \cdot \boldsymbol{r}_2 + \boldsymbol{e} - (\boldsymbol{r}_1 + \boldsymbol{h} \cdot \boldsymbol{r}_2)\boldsymbol{y}) \\ &= \boldsymbol{m}\boldsymbol{G} + \boldsymbol{x} \cdot \boldsymbol{r}_1 - \boldsymbol{y} \cdot \boldsymbol{r}_2 + \boldsymbol{e}\end{aligned}$$

The $\boldsymbol{e}' = \boldsymbol{x} \cdot \boldsymbol{r}_1 - \boldsymbol{y} \cdot \boldsymbol{r}_2 + \boldsymbol{e}$ is considered the error vector to be corrected. The hamming weight of this error can be too large for the decoding to handle, causing a decryption failure. An analysis of the distribution of $\boldsymbol{e}'$ is done to find the DFR.

3.3.2 Security

The hard problem HQC is based on versions of the Quasi-Cyclic Syndrome Decoding (QCSD) problem. The security target is IND-CCA2 security. According to NIST further analysis of the relation between the decision and search variants of QCSD problem would be valuable. It would also be valuable to investigate the the quasi-cyclic code structure to check if it affects security.

The IND-CPA security of HQC is achieved by using a randomness generator θ for the encryption algorithm `Encrypt(pk,m,θ)`. To convert this to IND-CCA2 with [9] the idea is to make the encryption deterministic by changing θ to G(m) where G is a secure hash function so that `Encrypt(pk,m,G(m))`. This would require a hash H(c, m) to be appended to the ciphertext to detect decryption failures.

There are some known attacks against quasi-cyclic codes called Specific structural attacks. One of these are the attack DOOM which have a gain of $\mathcal{O}(\sqrt{n})$. However this becomes inefficient when there are only two irreducible factors on the form $x - 1$ and $x^{n-1} + x^{n-2} + ... + x + 1$. Attacks directly on the HQC cryptosystem are reduced to the best attacks on the Decisional 2-QCSD and Decisional 3-QCSD problems. In the old implementation of HQC there is a known side-channel attack, more specifically a timing attack. This attack takes advantage of the BCH code implementation. A constant time BCH decoding is therefore also implemented and mandatory for real world use.

3.3.3 Parameters

HQC only list parameters for the HQC.KEM scheme, but since HQC.KEM is just a transformation of HQC.PKE, they have many of the same parameters. The public and secret key can be reduced to seeds at the expense of some calculation to recover the real public and secret key. We will first look at the old parameters from [28].

Old parameters

Instance	Security	n_1	n_2	n	k	δ	w	$w_r = w_e$	p_{fail}
hqc-128-1	1	796	31	24677	256	60	67	77	$< 2^{-128}$
hqc-192-1	3	766	57	43669	256	57	101	117	$< 2^{-128}$
hqc-192-2	3	766	61	46747	256	57	101	117	$< 2^{-192}$
hqc-256-1	5	766	83	63587	256	57	133	153	$< 2^{-128}$
hqc-256-2	5	796	85	67699	256	60	133	153	$< 2^{-192}$
hqc-256-3	5	796	89	70853	256	60	133	153	$< 2^{-256}$

Security	pk-size	sk-size	ct-size	ss-size
1	6,170	252	6,234	64
3	10,918	404	10,981	64
3	11,688	404	11,749	64
5	15,898	532	15,960	64
5	16,926	566	16,984	64
5	17,714	566	17,777	64

New parameters

Instance	Security	n_1	n_2	n	k	δ	w	$w_r = w_e$	p_{fail}
hqc-128	1	766	31	23869	256	57	67	77	$< 2^{-128}$
hqc-192	3	766	59	45197	256	57	101	117	$< 2^{-192}$
hqc-256	5	796	87	69259	256	60	133	153	$< 2^{-256}$
hqc-rmrs-128	1	80	256	20533	-	-	67	77	$< 2^{-128}$
hqc-rmrs-192	3	76	512	38923	-	-	101	117	$< 2^{-192}$
hqc-rmrs-256	5	78	768	59957	-	-	133	153	$< 2^{-256}$

The tensor product code used in hqc is uses the parameters n_1, n_2, k and δ. While the Reed-Muller and Reed-Solomon code uses the parameters n_1, n_2 and k. The parameters with - are not specified for the Reed Muller Reed Solomon scheme. I did not figure out why k is not specified for HQC-RMRS.

Instance	pk-size	sk-size	ct-size	ss-size
hqc-128	3,024	40	6,017	64
hqc-192	5,690	40	11,364	64
hqc-256	8,698	40	17,379	64
hqc-rmrs-128	2,607	40	5,191	64
hqc-rmrs-192	4,906	40	9,794	64
hqc-rmrs-256	7,535	40	15,047	64

The reason of a significant decrease of key sizes of the new parameters compared to the old are because the secret key is represented as the seedexpander. This means to get the x and y in the secret key you have to compute parts of the keygeneration.

3.3.4 Intellectual Property

There are some patents on the cryptosystem. However if HQC is chosen for standardization a non-exclusive license under reasonable terms and conditions would be provided for implementing the standard [29].

3.4 LAC

LAC stands for Lattice-based Cryptosystem, and is as the name says, a cryptosystem based on lattices. The submission consists of four public key cryptography primitives:

- **LAC.CPA** - a PKE scheme with IND-CPA claimed security
- **LAC.CCA** - a KEM scheme with IND-CCA claimed security, which is based on LAC.CPA
- **LAC.KE** - a key exchange protocol which is passively secure, converted from LAC.CPA
- **LAC.AKE** - an authenticated key exchange protocol obtained from LAC.CCA and LAC.CPA

The main focus of the submitters of LAC was reducing size of keys while achieving the claimed security categories. This has resulted in a modulus of the system that can fit in a single byte, which is in many ways desirable to reduce sizes of the cryptosystem. The small modulus allows each element of the polynomial to fit in a single byte which leads to performance increase by using instruction sets like AVX2.

To maintain the security levels with the small modulus size, it is needed to add larger random errors on the ciphertext which affects the error rate. This causes a need for error correcting when sending a message. BCH code is therefore used to correct most of these errors and to make the real error rate manageable for the standards of the NIST PQC project.

These properties leads LAC to be an efficient and generate small public keys, however there are some vulnerabilities which will go through in the security section. The LAC team updated their submission 15th of April 2020 to round 2 in the NIST PQC project [30]. This update include changes to the implementation where a new polynomial multiplication and noise generation was added. According to the submitters this makes the system 2-4 times faster than the original submission to round 2. New parameter sets was included for IoT devices and for $q = 256$.

3.4.1 How It Works

I will go through how LAC.CPA system works with algorithm 1, 2 and 3 from [31]. To start we will look at some notation for the algorithms. First we will define the message space $\mathcal{M}$ to be $\{0,1\}^{l_m}$ for a positive integer l_m. $\mathcal{S} = \{0,1\}^{l_s}$ is the space of seeds for randomness, where l_s is a positive integer. Ψ_σ is a binomial distributions and $\Psi_\sigma^{n,h}$ is the n-ary centered binomial distribution, where the hamming weight of the elements in the distribution is h. The modulus is defined as q and the polynomial ring is defined as $R_q = \mathbb{Z}_q/(x^n + 1)$. ECCEnc and ECCDec are the error correcting encoding and decoding routines used in the scheme. They convert a message between $\boldsymbol{m} \in \mathcal{M}$ and a codeword $\hat{\boldsymbol{m}} \in \{0,1\}^{l_v}$, where l_v is the length of the encoded message. The U routine is creating a uniform distribution over the set that is taken as input. The choices of the parameters will be given in the parameters section for each security level.

Algorithm 1 LAC.CPA.KG()

```
Ensure:
    A pair of public key and secret key $(pk, sk)$
1: $\text{seed}_a \xleftarrow{\$} \mathcal{S}$
2: $\boldsymbol{a} \xleftarrow{\$} \text{Samp}(U(R_q);\text{seed}_a) \in R_q$
3: $\boldsymbol{s} \xleftarrow{\$} \Psi_\sigma^{n,h}$
4: $\boldsymbol{e} \xleftarrow{\$} \Psi_\sigma^{n,h}$
5: $\boldsymbol{b} \leftarrow \boldsymbol{as} + \boldsymbol{e} \in R_q$
6: return $(pk := (\text{seed}_a, \boldsymbol{b}), sk := \boldsymbol{s})$
```

The keygeneration algorithm 1 of LAC is returning a public and private key pair. The algorithm uses a seed as randomness. First the algorithm chooses $\boldsymbol{a}$ randomly from the samples of the R_q polynomial ring. The value of $\boldsymbol{s}$ and $\boldsymbol{e}$ is randomly chosen from the n-ary centered binomial distribution $\Psi_\sigma^{n,h}$. The value of $\boldsymbol{b}$ is calculated, and the public key is seed_a and $\boldsymbol{b}$, while the private key is $\boldsymbol{s}$.

Algorithm 2 LAC.CPA.Enc(pk =(seed_a, $\boldsymbol{b}$), $\boldsymbol{m} \in \mathcal{M}$; seed $\in \mathcal{S}$)

```
Ensure:
    A ciphertext $\boldsymbol{c}$
1: $\boldsymbol{a} \leftarrow \text{Samp}\ (U(R_q);\text{seed}_a) \in R_q$
2: $\hat{\boldsymbol{m}} \leftarrow \text{ECCEnc}(m) \in \{0,1\}^{l_v}$
3: $(\boldsymbol{r}, \boldsymbol{e}_1, \boldsymbol{e}_2) \leftarrow \text{Samp}(\Psi_\sigma^{n,h}, \Psi_\sigma^{n,h}, \Psi_\sigma^{l_v}; \text{seed})$
4: $\boldsymbol{c}_1 \leftarrow \boldsymbol{ar} + \boldsymbol{e}_1 \in R_q$
5: $\boldsymbol{c}_2 \leftarrow (\boldsymbol{br})_{l_v} + \boldsymbol{e}_2 + \lfloor \frac{q}{2} \rceil \cdot \hat{\boldsymbol{m}} \in \mathbb{Z}_q^{l_v}$
6: return $\boldsymbol{c} := (\boldsymbol{c}_1, \boldsymbol{c}_2) \in R_q \times \mathbb{Z}_q^{l_v}$
```

The encryption algorithm is defined in algorithm 2. The algorithm takes pk, $m \in \mathcal{M}$ and seed $\in \mathcal{S}$ as input. The algorithm first samples a polynomial in R_q and assigns it to $\boldsymbol{a}$. The message $\boldsymbol{m}$ is encoded with error correcting code to $\hat{\boldsymbol{m}}$. The variables $(\boldsymbol{r}, \boldsymbol{e}_1, \boldsymbol{e}_2)$ are chosen from the samples of $(\Psi_\sigma^{n,h}, \Psi_\sigma^{n,h}, \Psi_\sigma^{l_v})$. The value of $\boldsymbol{c}_1$ is calculated from an equation of $\boldsymbol{a}$, $\boldsymbol{r}$ and $\boldsymbol{e}_1$, where $\boldsymbol{e}_1$ is considered the error vector. The value of $\boldsymbol{c}_2$ is calculated from an equation of $\boldsymbol{b}$, $\boldsymbol{r}$, $\boldsymbol{e}_2$, q and $\hat{\boldsymbol{m}}$. Here $\boldsymbol{e}_2$ is considered the error vector. The output is the ciphertext $(\boldsymbol{c}_1, \boldsymbol{c}_2)$.

Algorithm 3 LAC.CPA.Dec($sk = \boldsymbol{s}$, $c = (\boldsymbol{c}_1, \boldsymbol{c}_2)$)

```
Ensure:
    A plaintext $\boldsymbol{m}$
 1: $\boldsymbol{u} \leftarrow \boldsymbol{c}_1\boldsymbol{s} \in R_q$
 2: $\tilde{\boldsymbol{m}} \leftarrow \boldsymbol{c}_2 - (\boldsymbol{u})_{l_v} \in \mathbb{Z}_q^{l_v}$
 3: for $i = 0$ to $l_v - 1$ do
 4:     if $\frac{q}{4} \leq \tilde{\boldsymbol{m}}_i < \frac{3q}{4}$ then
 5:         $\hat{\boldsymbol{m}}_i \leftarrow 1$
 6:     else
 7:         $\hat{\boldsymbol{m}}_i \leftarrow 0$
 8:     end if
 9: end for
10: $\boldsymbol{m} \leftarrow$ ECCDec($\hat{\boldsymbol{m}}$)
11: return $\boldsymbol{m}$
```

The decryption algorithm of LAC is defined in algorithm 3. The input of the algorithm is the secret key $sk = \boldsymbol{s}$ and the ciphertext $\boldsymbol{c} = (\boldsymbol{c}_1, \boldsymbol{c}_2)$. First step is to recover the encoded message $\hat{\boldsymbol{m}}$. For recovering this, computing two equations using $\boldsymbol{c}_1$, $\boldsymbol{s}$ and $\boldsymbol{c}_2$ will result in the encoded message with errors and in $\mathbb{Z}_q^{l_v}$. The result of this will be called $\tilde{\boldsymbol{m}} \in \mathbb{Z}_q^{l_v}$. To convert $\tilde{\boldsymbol{m}}$ to be $\hat{\boldsymbol{m}} \in \{0,1\}^{l_v}$ we need to remove the error added by the error vectors in the encryption. This is done by converting a number that is close to 0 to 0 in the modular space, and otherwise 1. This means in $\mathbb{Z}_q$ a number that is smaller than $\frac{q}{4}$ and larger than $\frac{3q}{4}$ is converted to 0. When this is done the error correcting code decoding is used so that bits that failed the conversion from $\mathbb{Z}_q$ to $\{0,1\}$ is discovered and corrected. If there is a decryption failure, then $\boldsymbol{m} \notin \mathcal{M}$.

3.4.2 Security

LAC is based on a variant of the Learning with Errors problem, specifically poly-LWE. According to NIST they lacked full confidence in the security category 5 parameters because some vulnerabilities had been discovered. This was according to the LAC team fixed by changing from a standard centered binomial distributions to a fixed hamming weight centered binomial distributions for choosing parameters in the scheme. This makes LAC immune to high hamming weight CCA attacks [31].

In [32] there was discovered a Chosen Ciphertext Attack on the LAC.CCA system. This is a multi-target CCA, where the problem is that the decryption failure rate is too high. The attack was resolved in the update [30]. To decrease the decryption error rate they reduced the message space for LAC128, use the same error correcting code for all security levels, decrease the error vector hamming weight in LAC256 and generate the noise vector from the public key and seed together. I did not have time to take a closer look at this update.

3.4.3 Parameters

The old parameters will be presented first.

Old parameters

Security	n	q	**dis**	**ecc**	l_m	**bit-er**	**dec-er**
1	512	251	$\Phi_1^n, \Phi_1^{n,\frac{n}{2}}$	[511,256,33]	256	$2^{-12.61}$	2^{-116}
3	1024	251	$\Phi_{\frac{1}{2}}^n, \Phi_{\frac{1}{2}}^{n,\frac{n}{4}}$	[511,256,17]	256	$2^{-22.27}$	2^{-143}
5	1024	251	$\Phi_1^n, \Phi_1^{n,\frac{n}{2}}$	[511,256,33]+D2	256	$2^{-12.96}$	2^{-122}

These parameters can replace the placeholders in algorithm 1, 2 and 3 to show how each security level works. The **dis** column contains the secret and noise distribution. The **ecc** contains the error correcting codes used. The **bit-er** is the bit error rate when converting from $\mathbb{Z}_q$ to $\{0, 1\}$. The **dec-er** is the decryption error rate for the cryptosystem with corresponding parameters. The l_m parameter is message size in bits.

Security	pk-size	sk-size	ct-size	pt-size	Error rate
1	544	512	712	32	2^{-116}
3	1056	1024	1188	32	2^{-143}
5	1056	1024	1424	32	2^{-122}

New parameters

The new parameters are specified in [33]. In the article, the submitters adds versions of the scheme using the modulus $q = 256$, but also keeping the old modulus $q = 251$. Therefore creating two versions of the scheme per security level. These are separated by v3a for $q = 251$ and v3b for $q = 256$. The rest of the parameters for each v3a and v3b pair are the same.

Instance	n	q	**dis**	**ecc**	l_m	**bit-er**	**dec-er**
LAC128-v3a	512	251	Φ_{512}^{256}	[255,128,17]+D2	128	$2^{-22.26}$	2^{-151}
LAC128-v3b	512	256	Φ_{512}^{256}	[255,128,17]+D2	128	-	$> 2^{-151}$
LAC192-v3a	1024	251	Φ_{512}^{256}	[511,256,17]+D2	256	$2^{-42.24}$	2^{-324}
LAC192-v3b	1024	256	Φ_{512}^{256}	[511,256,17]+D2	256	-	$> 2^{-324}$
LAC256-v3a	1024	251	Φ_{1024}^{384}	[511,256,41]+D2	256	$2^{-20.01}$	2^{-302}
LAC256-v3b	1024	256	Φ_{1024}^{384}	[511,264,41]+D2	256	-	$> 2^{-302}$

The decryption error rate of LAC-v3b is slightly smaller than its LAC-v3a counterpart.

Instance	pk-size	sk-size	ct-size	pt-size
LAC128-v3a	544	512	704	16
LAC128-v3b	544	512	704	16
LAC192-v3a	1056	1024	1344	32
LAC192-v3b	1056	1024	1344	32
LAC256-v3a	1056	1024	1448	32
LAC256-v3b	1056	1024	1448	32

The size of keys and messages are not directly specified in the documentation for the system. However it is possible to calculate this from the parameters. The public key is just a 32 byte seed, and an n byte vector. The secret key is an n byte vector, but can also be represented only as a 32 byte seed with some tradeoff in performance when decrypting. The ciphertext $(\boldsymbol{c}_1, \boldsymbol{c}_2)$ consists of an n bytes vector $\boldsymbol{c}_1$ and $l_m + l_r * 8$ bytes in $\boldsymbol{c}_1$, where l_m is the message size in bits and l_r is bits of rendundant data in the BCH code. For LAC128 and LAC192 it is $l_r = 8$, and for LAC256 it is $l_r = 21$. The sizes are the same for v3a and v3b. I did not have time to look closer at the performance of the new implementation and new parameters.

3.5 LEDAcrypt

LEDAcrypt is a merger of LEDAcrypt KEM and LEDAcrypt PKC from round 1 submissions. The candidate therefore contains both a key encapsulation mechanism and public key encryption scheme. The KEM is built from the Niederreiter cryptosystem, and the PKC is built from the McEliece cryptosystem. They are both based on linear error-correcting codes [34]. The LEDAcrypt team claims the KEM can provide both IND-CPA security with ephermal keys and IND-CCA2 security with long-lasting keys, and the PKC can provide IND-CCA2 security with long-lasting keys.

An advantage LEDAcrypt PKC have over the other PKE schemes is the ability to encrypt a larger message. This means that a system like this can have other use cases, e.g. be used in a hybrid PKE solution.

LEDAcrypt uses Quasi-Cyclic Low Density Parity Check (QC-LDPC) codes for error correcting to the McEliece cryptosystem. The encryption algorithm adds error to the codeword to make it a ciphertext. This causes the decryption algorithm to have decryption failures.

The LEDAcrypt team updated their submission a month before the deadline for round 2 in the NIST PQC project. The updated submission includes some handling of security concerns, but also improvements of decryption failure rate. This causes the public key and ciphertext sizes to decrease between 5%-25% for IND-CPA solution, and 30%-50%

for IND-CCA2 solution. I did not have the time to test performance of this updated implementation.

3.5.1 How It Works

I will go through how LEDAcrypt PKC works. This is the basics of how the McEliece system works. LEDAcrypt PKC do have some extra procedures which contains converting and padding of the message, initializing the error vector and check for decryption failure. These extra procedures will not be included in the specification below.

Keygeneration

Algorithm 4 LEDAcrypt.PKC.Keygen; From [35]

```
    Output: $(sk, pk)$
    Data: $p > 2$ prime, $ord_p(2) = p - 1$, $n_0 \geq 2$
 1: seed $\leftarrow TRNG()$
 2: $\{H, Q\} \leftarrow GENERATEHQ(seed)$
 3: $L \leftarrow HQ$
 4: if $\exists 0 \leq j < n_0$ s.t. $wt(L_j) \neq d_v \times m$ then
 5:     goto 2
 6: end if
 7: $LInv \leftarrow COMPUTEINVERSE(L_{n_0-1})$
 8: for $i = 0$ to $n_0 - 2$ do
 9:     $M_i \leftarrow LInv\, L_i$
10: end for
11: $Z \leftarrow diag([I, ..., I])$
12: $pk \leftarrow [\, Z \,|\, [\, M_0 \,|\, ... \,|\, M_{n_0-2} \,]^T \,]$
13: $sk \leftarrow \{H, Q\}$
14: return $(sk, pk)$
```

The keygeneration specified in algorithm 4 of LEDAcrypt uses a prime p and an positive integer n_0 as parameters. These parameters depend on the security level of the scheme.

The keygeneration algorithm first generates a seed from a True Random Number Generator (TRNG). This is used to generate the H and Q binary matrices. The $p \times pn_0$ quasi-cyclic parity-check block matrix H can be written as:

$$H = \left[H_0, ..., H_{n_0-1}\right]$$

where each H_i is a $p \times p$ circulant block. The hamming weight of each H_i is denoted as d_v so that there is a fixed odd number of non-zero elements per row/column. Since the elements are binary there are d_v elements that have value 1. The $pn_0 \times pn_0$ quasi-cyclic sparse matrix Q is structured as a $n_0 \times n_0$ block matrix. The elements in Q have size

$p \times p$ and are circulant matrices. Each of these blocks in Q can be written as $Q_{i,j}$ where $0 \leq i < n_0$ and $0 \leq j < n_0$. The hamming weight of each of these $Q_{i,j}$ blocks have to be constrained to make sure the invertibility of Q.

The multiplication on line 3 of H and Q creates a $p \times pn_0$ matrix L. The matrix L can be structured as a $1 \times p$ matrix:

$$L = \left[L_0, L_1, ..., L_{n_0-1}\right]$$

Each L_j $0 \leq j < n_0$ has a hamming weight equal to $d_v m$, where m is the hamming weight of each row of Q.

The next step at line 7-10 is to create a $p \times p(n_0 - 2)$ matrix. The structure of this matrix called M is $1 \times n_0 - 2$ where each element is a $p \times p$ circulant block. Each M_i block where $0 \leq i < n_0 - 1$ is the product of $L_{n_0-1}^{-1} L_i$.

$$M = \left[M_0, ..., M_{n_0-2}\right] = \left[L_{n_0-1}^{-1} L_0, ..., L_{n_0-1}^{-1} L_{n_0-2}\right]$$

This matrix is converted to a parity-check matrix in systematic form and used as the public key. The public key can easily be converted to a systematic generator matrix. The private key is the matrices H and Q.

Encryption

Algorithm 5 LEDAcrypt.PKC.Encrypt; From [35]

```
Input:
    $u = [u_0, ..., u_{n_0-2}]$: plaintext message;
    $e = [e_0, ..., e_{n_0-1}]$: error message with $wt(e) = t$;
    $pk = [Z \mid [M_0 \mid ... \mid M_{n_0-2}]^T]$: public key;
Output:
    $c = [c_0, ..., c_{n_0-1}]$: error affected codeword;
Data: $p > 2$ prime, $ord_p(2) = p - 1$, $n_0 \geq 2$
1: $blk \leftarrow 0$
2: for $j = 0$ to $n_0 - 2$ do
3:     $blk \leftarrow blk + u_j^T M_j$
4: end for
5: $c \leftarrow [u_0, ..., u_{n_0-2}, blk]$
6: for $j = 0$ to $n_0 - 1$ do
7:     $c_j \leftarrow c_j + e_j$
8: end for
9: return $c$
```

The input of the encryption specified in algorithm 5 is the plaintext u, the error vector e and the public key pk. The plaintext u is a $n_0 - 1$ long vector with $1 \times p$ binary vectors as elements. The error vector e is a n_0 long vector with $1 \times p$ binary vectors as elements.

The hamming weight of e is exactly t. The public key is the same as the output from the keygeneration. The output of the encryption algorithm is:

$$c = \Big[u_0 \,|...|\, u_{n_0-2} \,|\, \sum_{j=0}^{n_0-2} u_j M_j\Big] + [e_0 \,|...|\, e_{n_0-2} \,|\, e_{e_0-1}]$$

The plaintext message blocks is added with the corresponding error vector blocks. Since these are binary vectors, we can look at it as XORing. The last block $\sum_{j=0}^{n_0-2} u_j M_j$ is to create a codeword in the QC-LDPC code. This is needed to decrypt and error correct the message.

Decryption

Algorithm 6 LEDAcrypt.PKC.Decrypt; From [35]

```
Input:
    $c = [c_0, ..., c_{n_0-1}]$: error affected codeword;
    $sk = \{H, Q\}$: private key;
Output:
    $u = [u_0, ..., u_{n_0-1}]$: message;
    $e = [e_0, ..., e_{n_0-1}]$: error;
Data: $p > 2$ prime, $ord_p(2) = p - 1$, $n_0 \geq 2$
 1: $L \leftarrow HQ$
 2: $s \leftarrow Lc^T$
 3: $\{e, res\} \leftarrow LEDADECODER(s, sk)$
 4: if $res = true$ then
 5:     for $j = 0$ to $n_0 - 1$ do
 6:         $u_j \leftarrow c_j + e_j$
 7:     end for
 8: else
 9:     $e \leftarrow \perp$; $u \leftarrow \perp$
10: end if
11: return $(u, e, res)$
```

The input of the decryption specified in algorithm 6 is c as the error affected codeword and the secret key sk. The first steps of the decryption algorithm is to compute the parity check matrix L from H and Q. This matrix is used to find the syndrome Lc^T of the error affected codeword. With the syndrome it is possible to recover the error vector e with the weight the same as the syndrome. This is done with the function $LEDADECODER$. If the decoding is successful the res variable is $true$, otherwise $false$. If it was successful, the error vector is added to the ciphertext, or XORed since they are binary vectors. This means the original error vector is cancelled out and the result of this is $u = [u_0 \,|...|\, u_{n_0-2} \,|\, \sum_{j=0}^{n_0-2} u_j M_j]$.

3.5.2 Security

The submission for round 1 the team did not provide low enough error rate to claim it is IND-CCA2 secure. However with the merger and round 2 submission, the team proposed parameters with low enough DFR so that IND-CCA2 security could be guaranteed for long term keys. According to NIST a possible area for further analysis is if the security is affected by the extra structure in QC-LDPC compared to Quasi-Cyclic Moderate-Density Parity-Check (QC-MDPC).

In [36] an attack on LEDAcrypt is found by looking for weak private keys that occure 1 in $2^{47.72}$ times when generating keys. At security level 5, the private key can be found from the public key using only $2^{18.72}$ guesses. The attack takes advantage of product structure in the public key. The updated version of LEDAcrypt in [35] addresses the weak keys by changing the parameters to avoid the possibility to find these weak keys.

3.5.3 Parameters

I will go through the parameters for longterm LEDAcrypt PKC version.

Security	n_0	p	t	d_v	m	b_0	DFR
1	2	35,899	136	9	[5, 4]	44	2^{-64}
1	2	52,147	136	9	[5, 4]	43	2^{-128}
3	2	57,899	199	11	[6, 5]	64	2^{-64}
3	2	96,221	199	11	[6, 5]	64	2^{-192}
5	2	89,051	267	13	[7, 6]	89	2^{-64}
5	2	142,267	267	13	[7, 6]	88	2^{-256}

The parameters provided give a choice of having a decryption error rate at 2^{-64} or $2^{-\lambda}$ where λ is the bits of security for each security level. To achieve IND-CCA2 security, the parameters with DFR at $2^{-\lambda}$ must be chosen.

Security	pk-size	sk-size	sk-size memory	min ct-size	max ct-size	Error rate
1	4,488	25	468	4,521	8,967	2^{-64}
1	6,520	25	468	6,554	13,040	2^{-128}
3	7,240	33	660	7,283	14,480	2^{-64}
3	12,032	33	660	12,077	24,064	2^{-192}
5	11,136	41	884	11,189	22,272	2^{-64}
5	19,040	41	884	19,095	38,080	2^{-256}

The sk-size is the size of the seed used to generate H and Q matrices. This can be used as the private key, therefore reducing the size compared to storing H and Q. This will affect performance once a decryption is needed. The minimum and maximum ciphertext size is because the message size in LEDAcrypt PKC is variable. The specification does not clearly define the maximum size of the plaintext for the scheme.

The new parameters in [35] reduces the parameters by 30%-50% compared to the parameters specified above. I did not have time to test the performance of these new parameters.

Chapter 4

Methods

My project is about the performance and security of Post-Quantum cryptographic algorithms. To begin with I started researching around the NIST Post-Quantum Cryptography project. This project contains exactly what I needed for my project with implementations of many post-quantum cryptographic algorithms in a relatively uniform way, and documentation for each of them. This lead me to some research questions I wanted to answer:

- What is considered bottlenecks in a computer when looking at performance measurements of post-quantum cryptographic algorithms?
- What is the performance of candidates in the NIST Post-Quantum Cryptography project and how do they compare to each other when resources are constrained?
- What kind of performance increase or decrease does updated versions of the cryptographic algorithm candidates have compared to the original versions in round 2 of the NIST Post-Quantum Cryptography project?
- What does the resources and components in a computer that affect the performance of post-quantum cryptographic algorithms say about the optimization of these algorithms?

To answer these question, I chose to use quantitative methods, but with a some qualitative analysis of code. This means I needed to create performance measurements and analyse this data.

To find information about the subject I had to learn about two main subjects. These are post-quantum cryptography and performance optimization of algorithms. They do overlap in the sense that most implementations of cryptographic algorithms has the need of more optimization than the average algorithm. In the start I worked on the project for gathering performance measurements and being able to use the cryptographic algorithms.

The quantitative methods consists of collecting reliable and diverse results. To collect diverse results I have researched about what affects the performance of a computer. When finding new components or resources that has the potential of affecting performance, I tried finding different methods of either disabling features of the component or limit the resource. This creates the possibility of collecting performance data of the same algorithms but with different resources or components. The ideal case would be to test the candidates on different devices with different resources, but I only had a single computer to do the performance measurements. I also looked into new components or resources that can be added to a device to affect the performance of the algorithms if they were to be chosen as a standard public key encryption scheme. Much like the AES-NI instruction set that was introduced to increase the speed of AES.

I ended up looking at 5 parameters for limiting and disabling resources and components. These are:

- CPU clock speed
- Memory clock speed
- Cache pressure
- Core count
- C/C++ optimization flags

The CPU clock speed is changed by adjusting the speed of each core in the linux virtual filesystem. The speed of the processor used for measurments is between 800 Mhz and 4300 Mhz with a interval of 100 Mhz. The speed above 3400 Mhz is using Intel Turbo Boost technology which can affect the performance measurements, but in the results I did not find any results that appear affected by this. The memory clock speed has to be changed in the bios software. The speed of the memory is changed between 800 Mhz and 1600 Mhz. The reason for only two different speeds for measurements of the memory is because there was not a noticeable difference to defend looking at smaller intervals between these two values. We will look closer at the performance difference between these two values in the results and discussion. The cache pressure is a parameter that is possible to change in the system catalogue of Linux. This parameter can be changed from 0 to 200. The interval I chose between the values are 20. The core count can be changed the same way as the CPU clock speed, in the linux virtual filesystem. The core count is changed between one, two and four cores. The hyperthreading option of the computer is only one thread per core and is therefore not a factor. The last parameter is the C/C++ optimization flags. This has to be changed during the compilation of the project. There are a lot of different optimization flags that can be used, but for simplicity the flags that are compared are `-O3`, `-O2` and `-O1`.

For the qualitative analysis I decided to use a profiling tool for profiling each implementation of the candidates to look at the functions with the highest time consumption. With this information I could look closer at functions bottlenecking the system and maybe find optimization that can be done or have already been implemented.

To collect reliable results I have been identifying and removing errors and incorrect measurements. For avoiding outliers, a reasonable option is to run the same test multiple times and use the median of the results. This means that in the project, I run each procedure a number of times, often based on a time limit. The median of these results is used as the data for analysis. The time limit is set to 300ms, which I have found is a good balance between time used and removal of outliers in the dataset. A similar method is used for performance measurements in SUPERCOP. To make sure the data is useful I have used python to visualize the data while creating the project. To avoid other errors in the dataset I had to make sure there were no memory leaks or buffer overflows in the program. A memory leak or buffer overflow from one candidate could affect another and create errors in the results. I therefore had to test each algorithm separately to check for memory leaks, and find a uniform way of measuring the performance of the candidates. For analyzing the final dataset I used python with different analyzing and visualization tools. The libraries used in python are pandas and matplotlib. For exporting plots and other visualizations in python to the thesis PDF, I saved the plots as PGF files and used these as input in the LaTeX file.

To generate data I created a single C++ file and header. The data is generated as an `unsigned char`-array, because this is the input type of the plaintext expected for most of the candidates of the NIST PQC project. Because the array can be a variable size, it is returned as a vector. To extract the `unsigned char`-array, you can use the function `data()`. I also included some helping functions for working on the data, like checking two arrays if they have equal content, finding median of arrays and time difference between two clock points.

4.1 Performance measurements

The procedures for encryption, decryption and keygeneration for each cryptosystem has been streamlined to make it possible to make a generic performance measurement algorithm. The streamlining has made it possible for the performance measurement algorithm to take just an object as input. This object must contain three functions and four variables. The first function is `keygen` which needs `sk` and `pk` as input references to the allocated memory blocks. These arguments represent the secret key and public key respectively. Output of the function is written to these arguments. The variables in the object corresponding to this function is the length of the secret and public key. The next function is `encryption`. This function takes the arguments `ct`, `ctlen`, `pt`, `ptlen` and `pk`. These input variables represent the ciphertext, ciphertext length, plaintext, plaintext length and public key respectively. The output is written to the allocated `ct` and `ctlen` memory blocks. The final function is `decryption`. This function takes the input `pt`, `ptlen`, `ct`, `ctlen` and `sk`. These input variables represent the same variables as the `encryption`-function. Output of this function is written to allocated `pt` and `ptlen` memory blocks. The other variables in the object are ciphertext max size and plaintext max size.

Most of the cryptosystem used in the performance measurement algorithm only works with a fixed sized plaintext. Therefore the algorithm only generates data for the plaintext max size. Each procedure (keygeneration, encryption and decryption) is executed in a loop until the time limit is reached. The encryption and decryption is executed until one of them reach the time limit.

4.2 Implementation of PKE schemes

Most of the implementation of the post-quantum algorithm candidates does not directly come with a implementation of the public key cryptosystem, only a specification in the documentation. They do however come with a implementaion of the corresponding key encapsulating mechanism. For each algortihm I had to go through and find the functions that makes the PKE work, and check with the specification that it is the correct way the PKE scheme should work.

4.2.1 LEDAcrypt

LEDAcrypt does have the possibility to be used as a public key encryption scheme, but the documentation could be more clear on how to use it and with what parameters. When looking through the submission of implementation to the NIST PQC you find the PKC in its own folder. You'll find the implementation of LEDAcrypt is following the specifications of the NIST PQC API [37]. This is one of the few candidates in the project that follows the PKE specification. When running the `make`-command for compiling LEDAcrypt, `SL` and `DFR_SL_LEVEL` has to be defined. `SL` stands for Security Level and is set as one of the integers between 1 and 5. This definition decides the security level of the compiled functions. `DFR_SL_LEVEL` stands for Decoding/Decryption Failure Rate Security Level. This definition is either set to 0 or 1, where 0 means the likelihood of a decryption failure is 2^{-64}. If the `DFR_SL_LEVEL` is set to 1, it means the likelihood of a decryption failure is equal to 2^{-SL_BITS} where SL_BITS are the number of bits security for each security level (128 bits for level 1 and so on).

When compiling the candidate separate it does so with no problems. However some problems arise when it is implemented as library in other projects. When using the optimized LEDAcrypt implementation as a library with other candidates it causes undefined behaviour if function and variable names are not prefixed. This is most likely caused by the fact that functions are not defined in a namespace or correct naming conventions for functions in a library, as the usage of this implementation was not meant to be used as a library in the first place. The problem becomes even more clear when libraries of LEDAcrypt with different security levels are used in the same program. This will cause each security level to use the same functions. If the `-flto` option is used when compiling, the compiler will complain about multiple definition of functions. If this flag is not used, the functions with the same names will overwrite each other, and the last function linked will be used for all libraries. This means if the functions with security level 1 is linked last, The implementation with security level 5 will actually just be level 1. If level 5 is linked last, a buffer overflow with either a crash or undefined behaviour when running will happen. To avoid this the implementation were cloned as three different implementation for security level 1, 3 and 5. Then I added a prefix on function names and global variables for each of these to identify each security level separately. The prefix for security level 1 is `leda_1_`.

The final solution still have some warnings about conditional jumps in memory after running tools like valgrind, but I did not figure out why. The naming of the LEDAcrypt libraries are `libLEDApkc_sl$(SL)_DFR$(DFR_SL_LEVEL)` where `$(SL)` and `$(DFR_SL_LEVEL)` are replaced by security level and desired DFR respectively.

Encryption

The function used for encrypting data is called `leda_pkc_crypto_encrypt`. The arguments of the function are `c` for the ciphertext, `clen` for the ciphertext length, `m` for the message, `mlen` for the message length and `pk` for the public key. The output is the status of the encryption, where -1 is fail and 0 is success. Since the message length can be changed, the encryption function pads the message before it's encrypted. If the output is 0, then the ciphertext will be written in the `c` variable, which has to be allocated before it is used as an argument.

The maximum size of the message will vary based on the security level and the failure rate level set at compile time. Ciphertext size will also vary based on these parameters and the message that is encrypted. This function will always return 0 as long as the message size is in scope.

Decryption

The function used for decrypting the ciphertext is called `leda_pkc_crypto_encrypt_open`. The arguments are `m` for message, `mlen` for message length, `c` for ciphertext, `clen` for ciphertext length and `sk` for secret key. The return value is the status of the decryption, where -1 is a failed decryption and 0 is a success. The decryption function will output the corresponding message which was encrypted with the corresponding public key. The likelihood of the function returning -1 is based on the definitions of `DFR_SL_LEVEL` and `SL` set at compile time.

Keygeneration

The `leda_pkc_crypto_encrypt_keypair` function is used for keygeneration in the scheme. The arguments are `pk` for public key and `sk` for secret key. The function will always return 0 for success. The public key will be written into `pk`-variable and the secret key will be written into `sk`-variable. Both variables have to be allocated before used as arguments.

4.2.2 HQC

Only the KEM version of the scheme is included in the optimized implementation of HQC in round 2 of the NIST PQC project. Since the PKE version is a building block of the KEM it is possible to find it in the implementation. The security levels and decryption failure parameters are split in different folders. Most of the functions are similar in each folder, causing a problem when combining the security levels into a single project. This is overcome by changing the names of functions with the prefix `hqc_SL_DFR_` where `SL` is the security level in bits and `DFR` is the decryption failure rate parameter of the scheme. The functions and parameters used is for the HQC-PKE IND-CPA scheme and not HQC-PKE IND-CCA2 scheme. It would be interesting to look at the performance hit the IND-CCA2 scheme has. The HQC-PKE IND-CPA does not follow the NIST PQC API in its current implementation, although it could easily be converted to do so.

Encryption

The encryption of the implementation is done with the function called `hqc_pke_encrypt`. The arguments for this function does not follow the NIST PQC API as mentioned. The arguments are:
`u` - for the u-vector.
`v` - for the v-vector.
`m` - for the message.
`theta` - for randomness seed.
`pk` - for the public key.

The return type is void for the function. The arguments `u` and `v` are written to when encrypting and have to be allocated in memory before used. The reason `u` and `v` are outputs of the encryption, is that the decryption is computing $v - u * y$ where y is part of the private key. The implementation does come with a helper function called `hqc_ciphertext_to_string` which converts `u`, `v` and the hash of the message `d` to a string. This is done by appending each variable to the same address in memory. The reason the hash is included is because of the probabilistic decryption algorithm and it is needed in the KEM. Although this makes the PKE scheme susceptible to a Chosen Plaintext Attack variant, since the hash of the plaintext would make it obvious. To make the implementation follow NIST PQC API I created a function called `encrypt_hqc_SL_DFR` where `SL` is the security level in bits and `DFR` is the decryption failure rate level. The length of the message has to be fixed, as a padding scheme is not implemented and variable sized plaintext is not supported in the encryption. The function will therefore return -1 if message size is not equal to maximum message size.

Decryption

The decryption of HQC is done with the function called `hqc_pke_decrypt`. The arguments for this function does, like the encryption, not follow NIST PQC API specifications. The arguments are:
`m` - for the message.
`u` - for the u-vector.
`v` - for the v-vector.
`sk` - for the secret key.

The return type is void, even though a decryption failure may happen. This is because in the implementation, the decryption failure is not handled here. The probability of a decryption failure is low and would most likely not affect the performance measurements. Therefore I have to implement a function to handle this called `decrypt_hqc_SL_DFR` where `SL` is the security level in bits and `DFR` is the decryption failure rate level.

Keygeneration

The keygeneration function is called `hqc_pke_keygen`. This function does not follow the NIST PQC API because the return type is void. Therefore I created a function called `keygen_hqc_SL_DFR` which will always return 0.

Updated implementation

The updated implementation of HQC have many of the similar functions. Therefore the method of implementing the PKE into the project is similar to the original HQC.

4.2.3 RQC

The RQC implementation does not have any dedicated functions for directly encrypting and decrypting data like the documentation is specifying. However there does exist function that expects finite fields vectors as argument for encryption and decryption. There is no documentation I can find on what parameters to use when converting a character array to a finite field vector. This makes it difficult to find what size the plaintext should be.

As it is the IND-CCA2 secure KEM that is implemented, the seed used is based on the message. This has to be changed to achieve IND-CPA secure PKE scheme. The `randombytes` function from NIST can be used for generating a random seed. The finite fields vectors `u` and `v` are input and outputs in the encryption and decryption. These are described in the specification of the PKE scheme. To decrypt the system you only need to do decode the equation:

$$\vec{v} - \vec{u} * \vec{y}$$

Where $\vec{y}$ is part of the secret key. The decoding of Gabidulin codes is for correcting errors that are intentionally added in the encryption.

I created functions in C++ which follows the NIST PQC API, although there is only one valid message size since a secure padding is not implemented. To avoid name collisions, all functions and global variables are named with the prefix `rqc_SL_`, where `SL` is the security level in bits.

Encryption

The encryption algorithm in the implementation is called `rqc_pke_encrypt`. The arguments of the function are `u` for the u-vector, `v` for the v-vector, `m` for the message-vector, `theta` for the seed and `pk` for the public key. For converting the message to a message-vector, the function `ffi_vec_from_string_compact` is used. To convert the u- and v-vector to ciphertext, I use the function `hqc_ciphertext_to_string`. This function appends the vectors together in memory, but also requires the hash of the message since it is intended for the KEM. It is possible to just send a vector with random values for the hash as this will not be used in the decryption. An idea to fix this is to create a similar function which only appends u and v together in memory. The function I created to handle encryption is called `encrypt_rqc_SL`, where `SL` is the security level in bits.

Decryption

The decryption algorithm is called `rqc_pke_decrypt`. The arguments of the function are `m` for the message-vector, `u` for the u-vector, `v` for the v-vector and `sk` for the secret key. To convert a ciphertext of bytes to u- and v-vector, the function `hqc_ciphertext_from_string` is used. To handle this I created the function `decrypt_rqc_SL`.

Keygeneration

The keygeneration in the KEM is the same as the PKE scheme. The function used is called `rqc_pke_keygen` with `pk` and `sk` as arguments. The function I created to handle keygeneration is called `keygen_rqc_SL`, where `SL` is security level in bits.

Updated implementation

The updated implementation of RQC had some significant changes in the implementation. However the functions used for the implementation of the PKE scheme only changed names. The implementation of the PKE scheme for the updated RQC is therefore similar to the original. I tried changing parameters in the old implementation to match the example parameters given in [20], but it is not trivial to change the parameters of the original implementation. This is caused by finite field elements hardcoded in the implementation, but not in specified in the documentation. The comparison between the old and new implementation will therefore be hard as the parameters are different.

4.2.4 ROLLO

When I refer to ROLLO, I mostly refer to ROLLO-II, as this is the part which contains the PKE scheme. The implementation of the PKE scheme in ROLLO is not implemented as a standalone version from the KEM version. This means there are no functions that follows the NIST PQC API for PKE, but hidden in the functions for KEM scheme. Unlike RQC, the ROLLO implementation does not have encryption and decryption directly implemented with finite field vectors as input. With the specification of the PKE scheme and the implementation of the KEM, it is possible to create the PKE scheme. ROLLO-II has an implementation of each security level in different folders. Each folder contains many similar functions with similar names. This will cause name conflict when included in the same program. Therefore a prefix is added for each folder, the prefix is `rollo_ii_SL_`, where `SL` is the security level in bits.

Encryption

Since ROLLO-II does not have a dedicated function for encryption, I created one in C++ with the name `encrypt_rolloII_SL` where `SL` is the security level in bits. The arguments are following the NIST PQC API. To create the encryption algorithm, I followed the documented specification in the submission of ROLLO.

Decryption

The decryption does not have a dedicated function either. I created a function named `decrypt_rolloII_SL` where `SL` is the security level in bits. The decryption function follows the NIST PQC API. The function recovers the shared secret by using the rank support recover algorithm [18]. Then the hash of the shared secret and cipher of the message is XORed to recover the message.

Keygeneration

The keygeneration of the PKE scheme is similar to the KEM of ROLLO-II. Therefore they use the same function, named `crypto_kem_keypair`, and follows the NIST PQC API with `pk` and `sk` as arguments.

Updated implementation

In the midst of writing, a security concern of the parameters were discovered [20]. This results in the need to change parameters in the scheme to make sure it is not broken. Before the new implementation was submitted to the round 2 in [23], I tried to implement the example parameters specified for ROLLO-I in [20]. To change the ROLLO-I parameters, the files that had data that needed to be changed was:

- `api.h`
- `ffi.h`
- `ffi_elt.c`
- `parameters.h`

In the beginning it looks like `api.h` and `parameters.h` files were the only ones that needed to be changed. However as I discovered, parameters for finite field arithmetic were hardcoded into `ffi_elt.c`. In `ffi.h` there was a duplicate definition of `PARAM_M` called `FIELD_M`. Complications arose in `ffi_elt.c` when trying to use the updated parameters, and it was at this point the updated implementation was posted.

The updated implementation of ROLLO contains a new library for finite fields arithmetics called RBC, hash and updated parameters. The rest of the code is similar to the original with some changed names of functions and variables. Similar procedure was used to be able to make the PKE scheme available.

4.2.5 LAC

The implementation LAC provides is different from the other candidates. In their optimized implementation they provide a header file called `lac_param.h` where you either define LAC128, LAC192 or LAC256 for each security level. In the folder they also include examples and tests for both the KEM and PKE scheme. The PKE scheme is placed in an appropriately named file `encrypt.c`, where the functions follows the NIST PQC API to a certain extent, which will be discussed below. The PKE scheme implemented is the LAC.CPA with IND-CPA security. The files in the implementation are all located in a single folder. Although this makes it easier when configuring the compilation paths of each file, it does decrease the readability of the implementation. If the number of files is increased in the implementation it would quickly become chaotic for someone using the implementation. The LAC team does implement possibilities for activating constant time function, but these are not set as default.

Encryption

The encryption algorithm is called `crypto_encrypt` and follows the NIST PQC API. This means the arguments are `c` for ciphertext, `clen` for ciphertext length, `m` for message, `mlen` for message length and `pk` for public key. The function returns -1 if any of the references are null, otherwise it returns 0 and the ciphertext is written to `c` with length to `clen`. The function is probabilistic by generating a random seed and passing this seed to a deterministic encryption function called `pke_enc_seed`. This function is also used for the LAC.CCA scheme, where the seed is replaced by a hash of the plaintext, as described in [9].

Decryption

The decryption algorithm is `crypto_encrypt_open` and also follows the NIST PQC API. So the arguments are `m` for message, `mlen` for message length, `c` for ciphertext, `clen` for ciphertext length and `sk` for secret key. If any of the references are null, the function will return -1, otherwise 0. Although a decryption failure can occure, it does not return -1 if this happens. This breaks the NIST PQC API, because it should return -1 if a decryption failure happened. It is therefore impossible to know if a decryption failed unless some sort of extra error detection or change in the implementation is done. This error rate is minimal, so it will most likely not affect the performance tests in any way.

Keygeneration

The keygeneration function is called `crypto_encrypt_keypair` with the arguments `pk` and `sk` for public key and secret key. The function will return -1 if either argument is a null reference.

4.3 Hybrid PKE/Composite Mode

Each candidate in the NIST PQC project are targeted to be a post-quantum cryptographic algorithm. As the word ”candidate” indicates, they are not verified or standardized yet. The implementations are often more insecure than modern cryptographic algorithms because of side-channel attacks on the implementation and broken parameters. To find the best middle ground I decided to also look at temporary solution like combining the candidates with RSA, called hybrid PKE/composite mode.

In the implementation of the hybrid PKE/composite mode I decided to implement a solution only using RSA and the candidates. This was just to encrypt the output of RSA with the post-quantum algorithm. There are of course other possibility to implement this, like using Elliptic Curve Cryptography and AES. It was implemented just to get an idea of the performance loss of combining 2 public key encryption schemes. The Inner Level Encryption (ILE) result might have to be encrypted multiple times by the Outer Level Encryption (OLE), therefore the ILE ciphertext is split into the number of blocks to be able to encrypt each separate block. A padding is not used as the size of ILE ciphertext is fixed. This might not be a secure procedure, however it is just supposed to be a performance measurement. I did not have time to look closely on the measurements of these solutions.

4.4 Compilation

For compiling the project, I used the `make` command with a Makefile for the instructions. The implementations of the candidates in NIST PQC project had to use make for the compilation, therefore for consistency and simplicity I decided to use the same. Since the data I was interested in included different C/C++ optimization flags, I had to create a method for compiling all the implementation of the candidates at the same time as the main project is compiled. This was done by including all the implementation as subfolders of the project. This way `make` can be used recursively through the subfolders, compiling each candidate. To avoid an overflow of object files from each candidate used in the compilation of the main program, each candidate for each security level creates it's own static library. These libraries are included when compiling of the main file.

For using different optimization flags, the user has to specify a Makefile-flag called `FLAGS`. This flag can be three different values: 1, 2, 3 or 4. Each of these values represent `-O3`, `-O2`, `-O1` and debugging flags respectively. The default value if not specified is 1. This flag will be passed the `make` procedure for each subfolder and compiled with the corresponding optimization flag. Debugging flags are not used for performance measurements, only when profiling.

4.5 Profiling

For profiling I use a script for compiling the project with the correct debugging flags. It also runs the compiled project with candidate and procedure (keygen, encryption or decryption) names that are to be profiled. This will make the program only call the procedure chosen until the time limit is reached. When choosing decryption or encryption the results will be a bit skewed as they will need to call keygeneration once to generate a keypair for use. The debugging flag will also impact optimization and therefore not give an accurate time consumption for each function with different optimization flags.

4.6 Common function and libraries

The external libraries used are libraries that the candidates use for their implementations. According to NIST these libraries that can be any libraries that is easily installed with a `make` command. Some candidates have decided to use external libraries for finite field arithmetics, which often is done by using NTL and gf2x. Most of the candidates uses OpenSSL for cryptographic and hash algorithms, and XKCP for other hash algorithms.

4.6.1 FFI and RBC

ROLLO and RQC share the same named folder ffi for the old implementations and rbc for the new implementations. The FFI and RBC library handle all operations and objects that is related to finite fields. Since the submitters of these candidates are the same, it makes sense they use the same resources. Other candidates has used 3rd party libraries to represent finite fields and the operation on the finite fields.

The implementation of the finite fields looks at first glance to be similar. However the different folders use different parameters and irreducible parameters, causing the implementation of these libraries to be different. It is easier to tell apart with RBC since the folders have the name `rbc-N`, where N is a number identifiend the specific finite field elements and parameters in the library. So instead of combining the libraries to a single folder, each implementation have their own ffi/rbc folder where each function has a prefix to identify each candidate.

4.6.2 NTL and gf2x

The Number Theory Library (NTL) and the gf2x library are used by some candidates in their implementation of operations on finite fields. The libraries are used because they are known for their optimizations of multiplication and other operations. NTL and gf2x uses some hardware optimized operation that may give some advantages to the candidates using it over others. The libraries did cause some problems as they would not work properly on the computer used for performance measure. This problem was caused by using the C++ version of the ROLLO implementation. However ROLLO did also have a C implementation not using the libraries.

4.6.3 XKCP

The eXtended Keccak Code Package (XKCP) is a library for hash algorithms. The library is used by some candidates who has the need for SHA256 and other hash algorithms.

4.6.4 OpenSSL

OpenSSL contains functions for different needs of the candidate cryptosystems. The library contains hash algorithms, randomness generators and cryptographic functions. The library is widely used over TCP and SSL protocols. If a candidate algorithm is standardized, it could be implemented into OpenSSL and make it easily adapted by many users. The Open Quantum Safe project has already created the possibility of beeing implemented into OpenSSL through a branch on GitHub.

Chapter 5

Results and discussion

5.1 Overall findings

The overall findings is the result of the measurments of all cryptosystems. In my opinion it helps to first look at the performance of the cryptosystems compared to each other. The graphs for the performance comparison will be in logarithmic scale, because the speed difference between the different algorithms do not allow for a linear scale to be visualized well. In figure 5.1, 5.2 and 5.3 we find the comparison between each of the cryptosystem for each security level. To get a better comparison between the different cryptosystem, I decided to divide the cycles by the size of the plaintext for each system. This will give a more fair evaluation for the system with large plaintext size compared to looking at raw performance data.

There are of course different use cases for these system. Most of them will be converted to a key encapsulating mechanism and used this way. In a symmetric encryption scheme the need for a key or seed larger than 32 bytes (256 bits) is unusual. In these cases a PKE scheme with the possibility to encrypt a 6.5kB will be wasted potential. Other use cases for a public key encryption scheme is sending a small messages to multiple recipients. In a use case like this, an efficient encryption scheme and small public key would be preferable. While for a conversion to a KEM, a small public key and efficient encryption and decryption will be just as important. Another use case is for IoT-devices. These devices often have limited memory and processing power and would therefore require the cryptosystem not exceed a certain size with regard to key sizes and code size. Different use cases are something NIST has taken into account when creating the NIST PQC project, where not a single winner will be chosen.

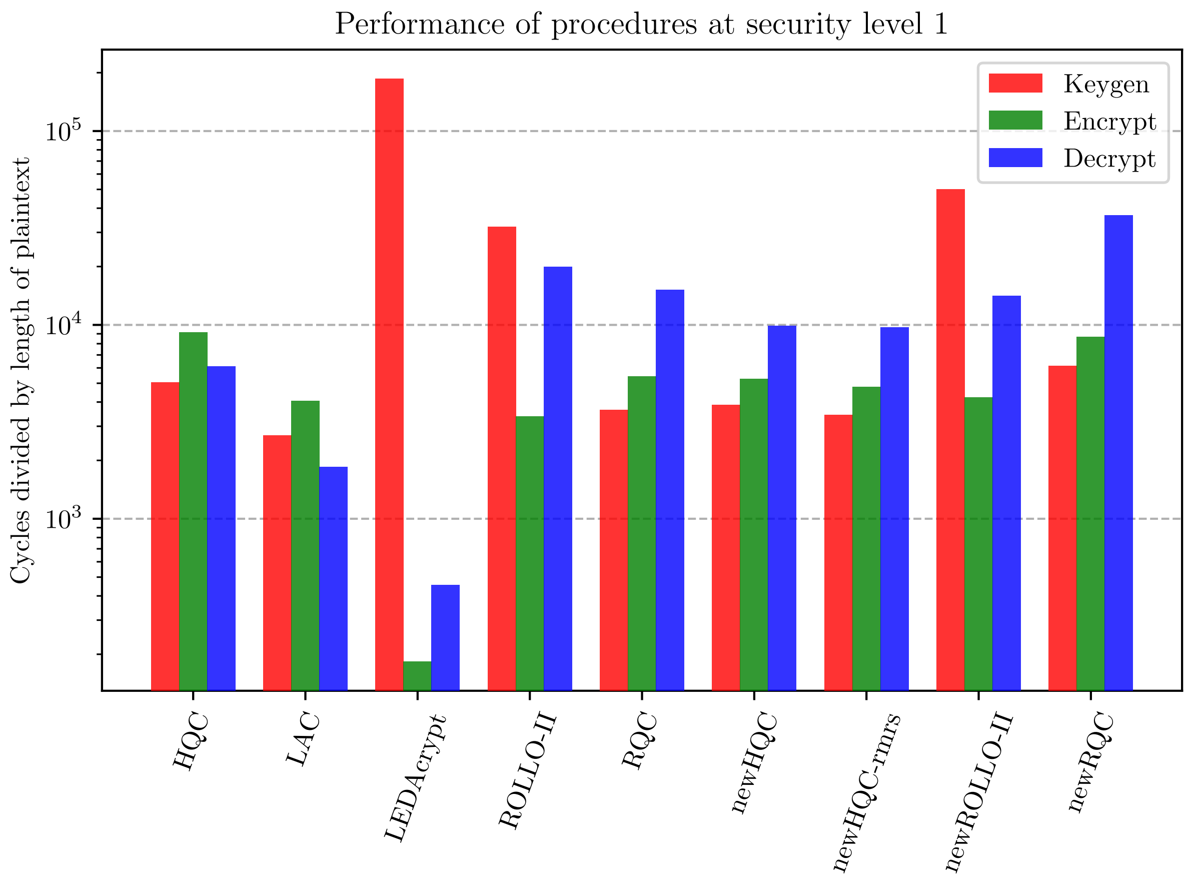

Figure 5.1: The figure shows the performance results between the different candidates at security level 1. The HQC, ROLLO-II and RQC are old implementations using broken parameters which are no longer safe for the security level.

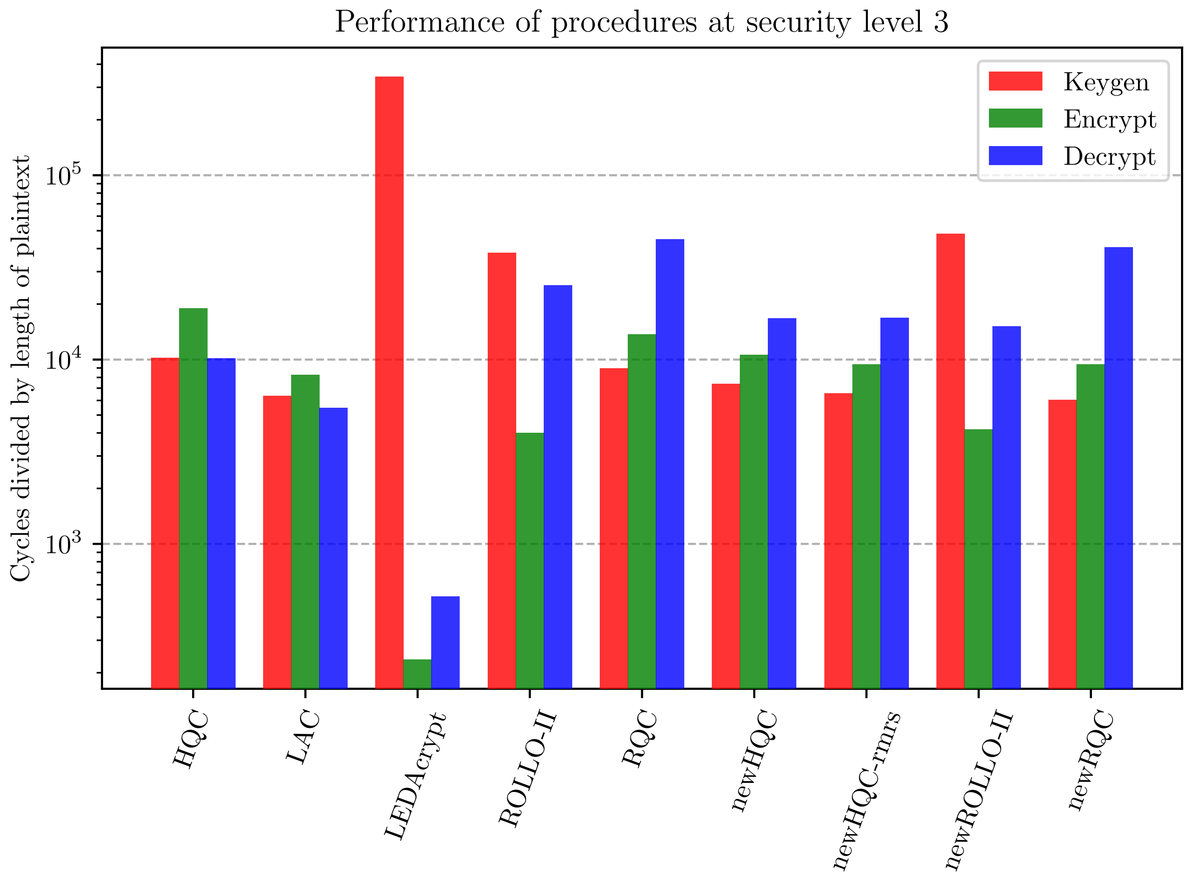

Figure 5.2: The figure shows the performance results between the different candidates at security level 3. The HQC, ROLLO-II and RQC are old implementations using broken parameters which are no longer safe for the security level.

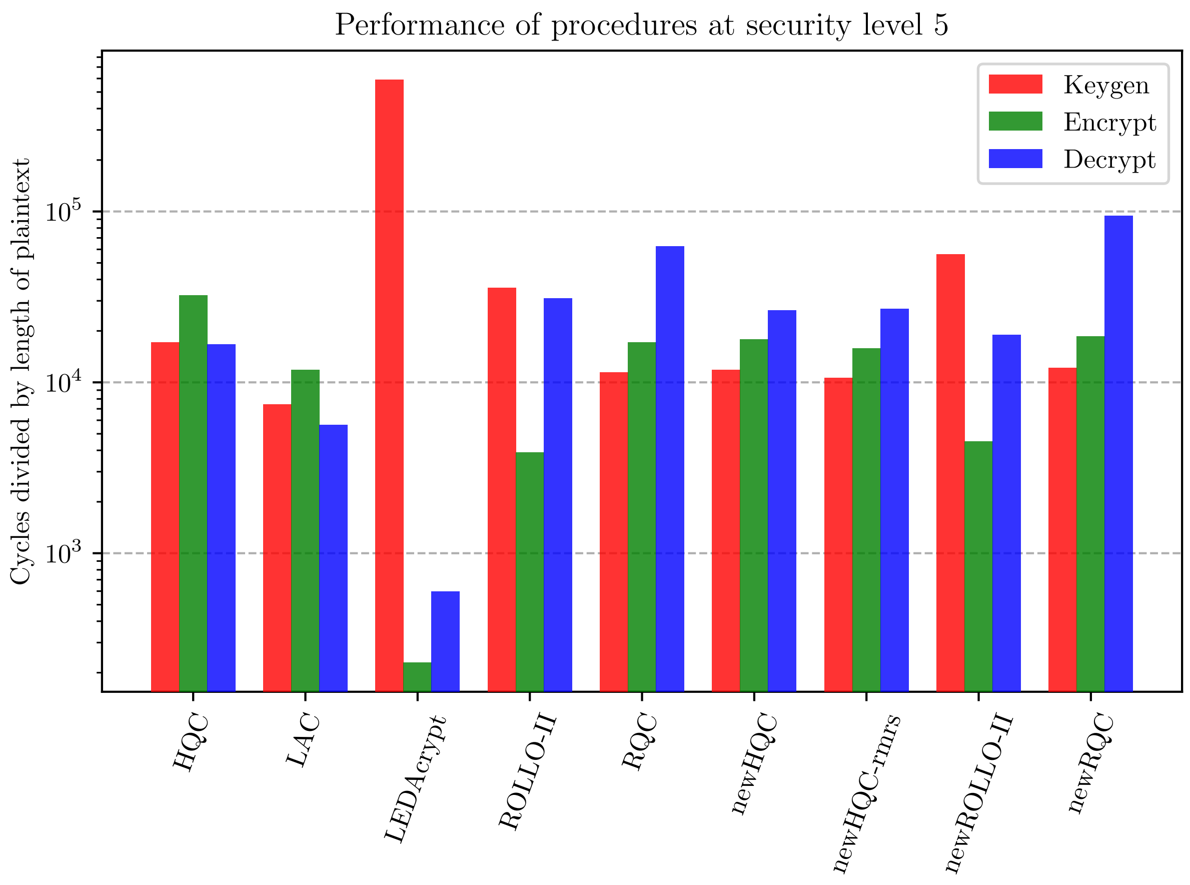

Figure 5.3: The figure shows the performance results between the different candidates at security level 5. The HQC, ROLLO-II and RQC are old implementations using broken parameters which are no longer safe for the security level.

	Keygen cycles	Encryption cycles	Decryption cycles
Clock speed	-0.7772	-0.7659	-0.7458
Memory speed	0.001350	0.001507	0.001231
Cache pressure	-0.002228	-0.001617	-0.001690
Core count	-0.002504	-0.001411	-0.001612
C/C++ flags	0.1182	0.1393	0.1621

Figure 5.4: The correlation matrix between the parameters and performance measurements. Although these numbers are affected by the number of cryptosystem in the data, they do give a basic idea of what the affect of each parameter has compared to each other.

In figure 5.4 a negative value means when the higher parameter value, the faster the procedures run. If the value is zero, then the parameter does not weigh towards better or worse performance of the cryptosystems. If there is a positive value, then the increase of the parameter affects the performance in a negative way. The values are relative to each other, which means that if the CPU clock speed has a large correlation with the keygeneration cycles then the correlation value with the other parameters will decrease.

When looking at the combined data of all cryptosystems it is important to normalize it. This means to find the smallest number of cycles for keygeneration, encryption and decryption for each cryptosystem. This data is then used to normalize the performance results of the cryptosystem it is from. The reason this is important is to make sure each cryptosystem has the same affect on the aggregated graphs and correlations between performance and the components that affect performance. This is especially noticeable on the correlation technique I chose. The normalized data can also be used when looking at each system separately, as it is easier to see the performance difference when working with percentages instead of cycles.

The correlations is found by using the Kendall method. Correlation is often used as a method to test for the association between parameters. However it does not make any assumptions on causation. The reason for the choice of Kendall method over the standard Pearson method is because we can encounter non-linear relations. Kendall can measure all types of monotonically correlations. This means however it is only monotonic correlation that is measured. A non-monotonic correlation like a quadratic correlation can be hidden with a correlation value close to zero.

If we look at the parameter C/C++ optimization flags, with different flags categorized as 1 for `-O3` flag, 2 for `-O2` flag, and 3 for `-O1` flag. It means the correlation method is looking for monotonic correlation between performance and three different categories. This can be fixed by displaying them in a bar graph in figure 5.5.

The core count seems according to the correlation figure 5.4 to have a small affect on the performance measurements. The different number of cores tested are 1, 2 and 4 cores.

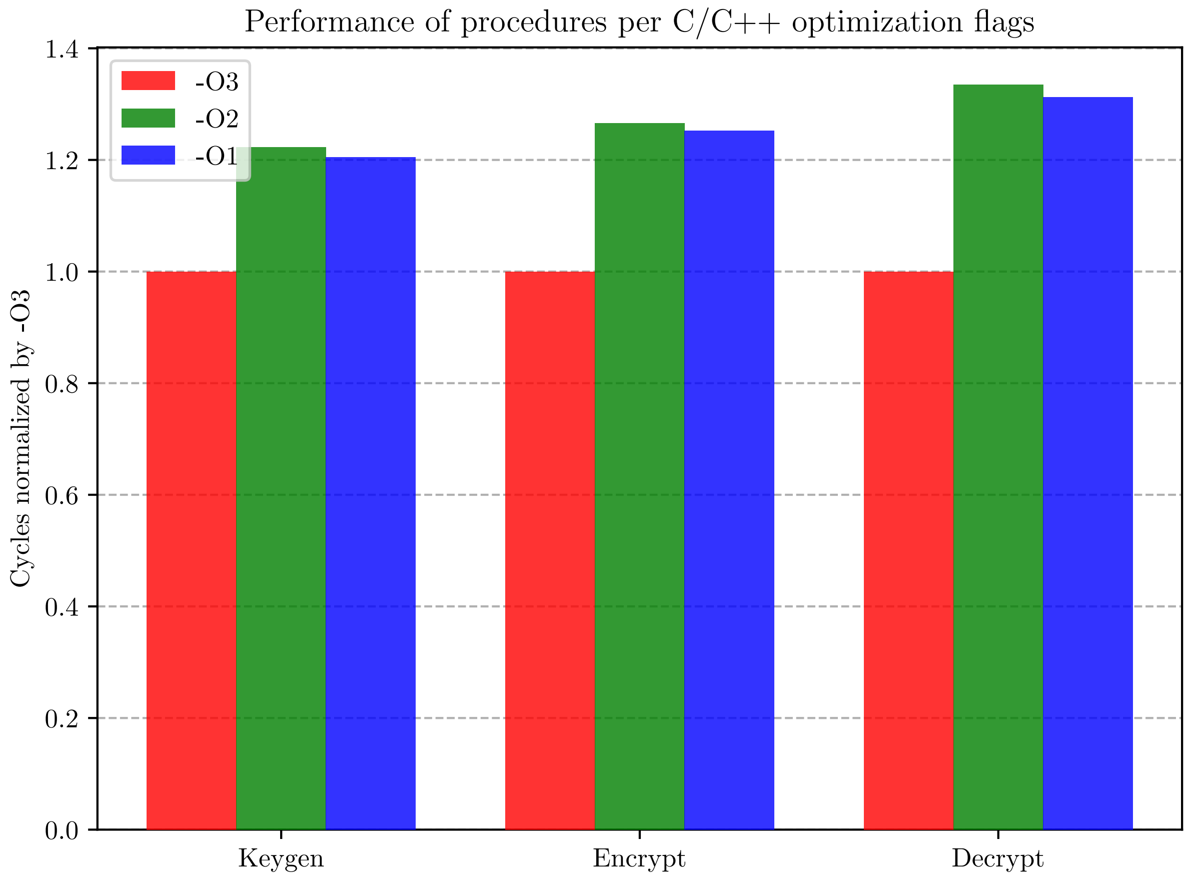

Figure 5.5: The bars represent the time or cycles difference between each optimization flag. The bars are normalized from the number of cycles of the `-O3` parameter. This means if a bar has a height of 1.2, the average runtime is 20% slower than `-O3` flag.

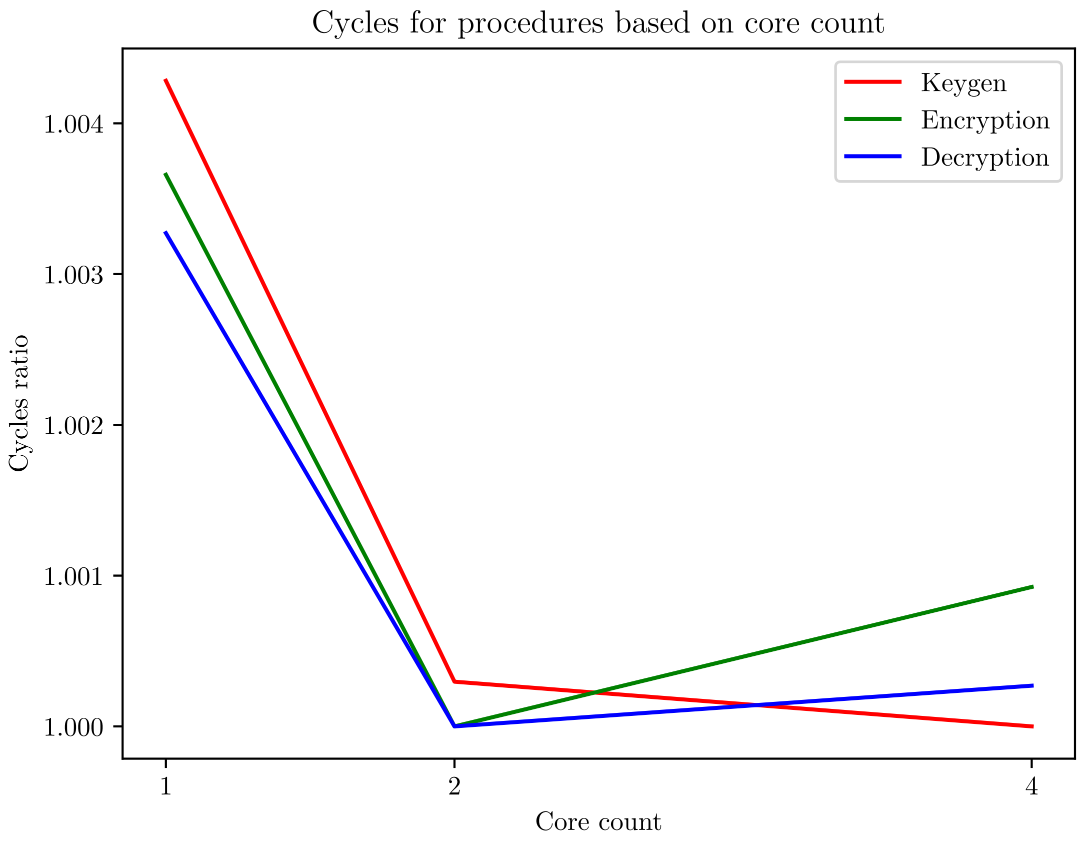

Figure 5.6: The difference in the median between two and four cores are within 0.1%. While the difference to one core is up to 0.4%. The plot is calculated from the mean of the normalized performance measurements of each core count. The value for each core is then divided by the smallest value among them to find the performance difference between each other. The algorithms that had higher runtime did notice more of an affect of running on one core, suggesting CPU utilization of other processes.

The performance results of one core will likely be slower than the other parameter values as the results will most likely be more affected by background processes. This is because all background processes have to run on the same core as the performance measurement program. To avoid as much errors as possible I let the program run on the computer without any user interactions. In figure 5.6 we see that the difference of less than 0.2% between two and four cores are within my opinion the margin of error. However if we look at the relatively significant performance decrease of around 0.6% with just one core. This is most likely due to background processes forced to utilize the same core the main program running performance measurements.

The cache pressure seems to be affecting the encryption algorithms generally more than the keygenerations and decryptions. The problem with measuring the correlation like this is that the parameter is not necessarily monotonically. The default value of the cache pressure is 100, while the measurements are between 0 and 200 with an interval of 20. It seems reasonable to believe that the most efficient value for the cache pressure would be the default value. Therefore it would not be surprising to see a quadratic or some other non-monotonically regression.

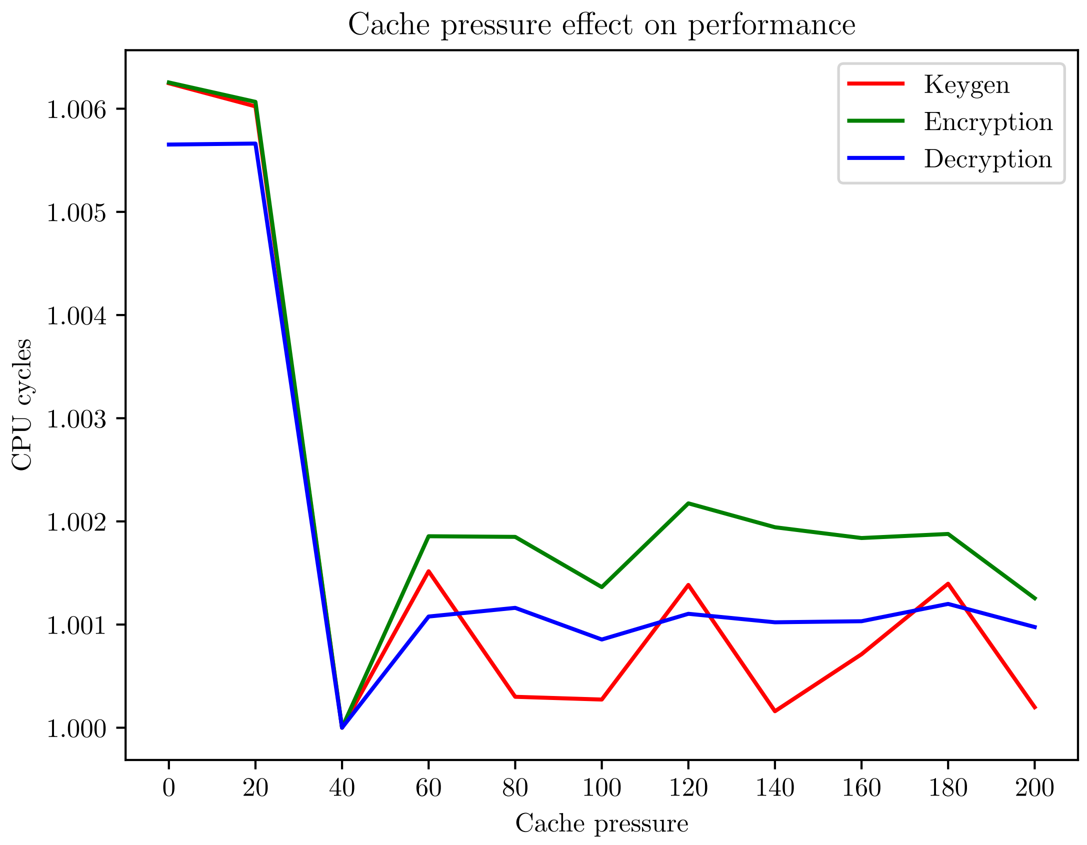

Figure 5.7: The cache pressure affect on the performance measurements is most noticable when it is below 40. The cache pressure at these values are up to 0.6% slower than the rest of the cache pressure values. The cache pressure values of 40 and above is within 0.2% of each other, which could be considered within margin of error.

In figure 5.7 we see that the cache pressure below 40 will give performance decrease up to 0.6%. This can be caused by the feature cache pressure is providing. What happens when the cache pressure is too low is that Linux will not be as active removing memory pages hardly used in the cache space to the slower swap space memory. Therefore memory pages more frequently used does not fit in the remaining cache space. When the cache pressure is 40 or above it seems this problem disappears. With the cache pressure increasing it will utilize the CPU more. This will in theory lead to a performance decrease with a higher cache pressure, but any clear trend outside the margin of error is not visible.

Looking more specifically on the performance effect of the cache pressure with only `-O3` flag in figure 5.8 we see that the cache pressure has a much more significant impact.

This is interesting as we do not find the same curve on `-O1` and `-O2` flags. I do not have much experience with cache and memory behaviour, but if I were to try to find a reason it would be that `-O3` do some extra cache optimization that is affected when the cache pressure is too low.

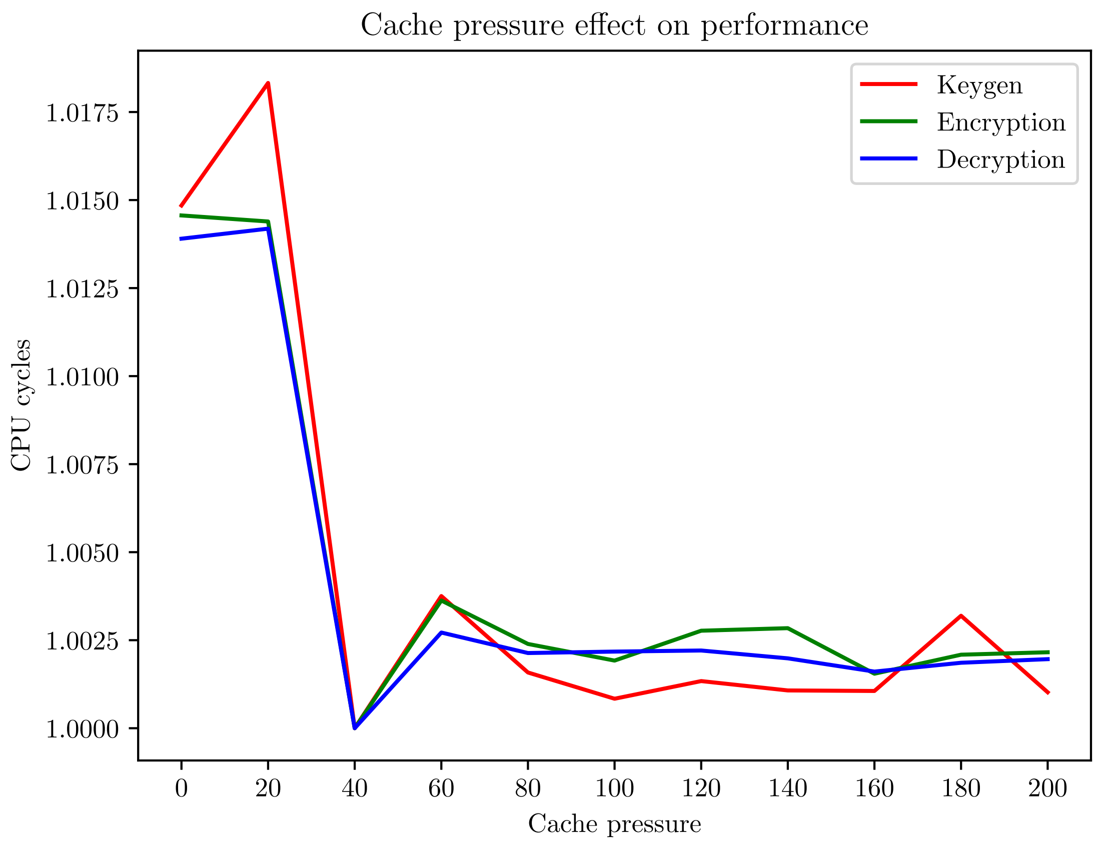

Figure 5.8: The cache pressure affect on the performance measurements with only `-O3` flag. This same curve is not visible when looking at the cache pressure with `-O1` and `-O2`.

The memory speed gives a interesting results. According to the data in 5.4 it seems like there is a correlation with slower memory speed gives higher performance. Common sense would say that when the memory speed is slower, the code in the memory would be faster to load, and therefore run faster. The average decrease in performance from 800MHz to 1600MHz is around 0.2%. This can be considered within margin of errors, but it is still interesting that the performance did not increase with the increase of memory speed. To my knowledge this can be caused by four factors:

1. All code that is used is stored in cache that is not affected by the RAM clock speed.
2. The calls to the memory and waiting for an answer is not counted in the cycles measured.

3. The change in RAM speed in the BIOS is not working correctly.
4. Change in speed is only a bottleneck if the maximum bandwidth of the memory is reached.

The CPU clock speed is as expected the big bottleneck of the system. Every algorithm is dependant on the CPU clock speed in the same degree. If I plot normalized performance measurements of candidates on top of each other, the result ends up like in figure 5.9. All measurements are overlapping within margin of error except some outliers for the keygeneration of LEDAcrypt. The reason LEDAcrypt is not accurate is because the keygeneration procedure takes too long time for a 300ms runtime to find a representative median result. The regression line is calculated from a simple linear regression on the logarithmic values of the performance measurements except the keygeneration of LEDAcrypt. The formula for the regression line is:

$$f(x) = \frac{4305.77}{x^{1.00024}}$$

If we find the root mean squared error of the test data (which is 20% of the original data), we find it to be 0.4. A root mean squared error of 0.4 means the function is good at predicting performance measurements from the function. The equation $\lim_{x\to 0^+} 4305.77x^{-1.00024} = \infty$ makes sense as a clock speed of 0 would never finish the procedures. However the $\lim_{x\to\infty} 4305.77x^{-1.00024} = 0$ does not make sense since when the clock speed increases there are still other parameters such as cache response time and bandwidth that will bottleneck the speed. Although this is just theory and not something that could happen in the real world. It would make sense to add another parameter to the equation so that $f(x) = 4305.77x^{-1.00024} + d$.

The formula found looks surprisingly close to:

$$g(x) = \frac{4300}{x}$$

This formula would suggest a full bottleneck of the system based on the CPU clock speed. This can be explained by a CPU clock speed reduction of 50% (e.g. from 4300MHz to 2150MHz) will result in a reduction in performance of 50%.

Although it is good to have some general idea of what the aggregated performance based on the different components looks like. It makes much more sense to look at the different performance of each post-quantum cryptosystem separately.

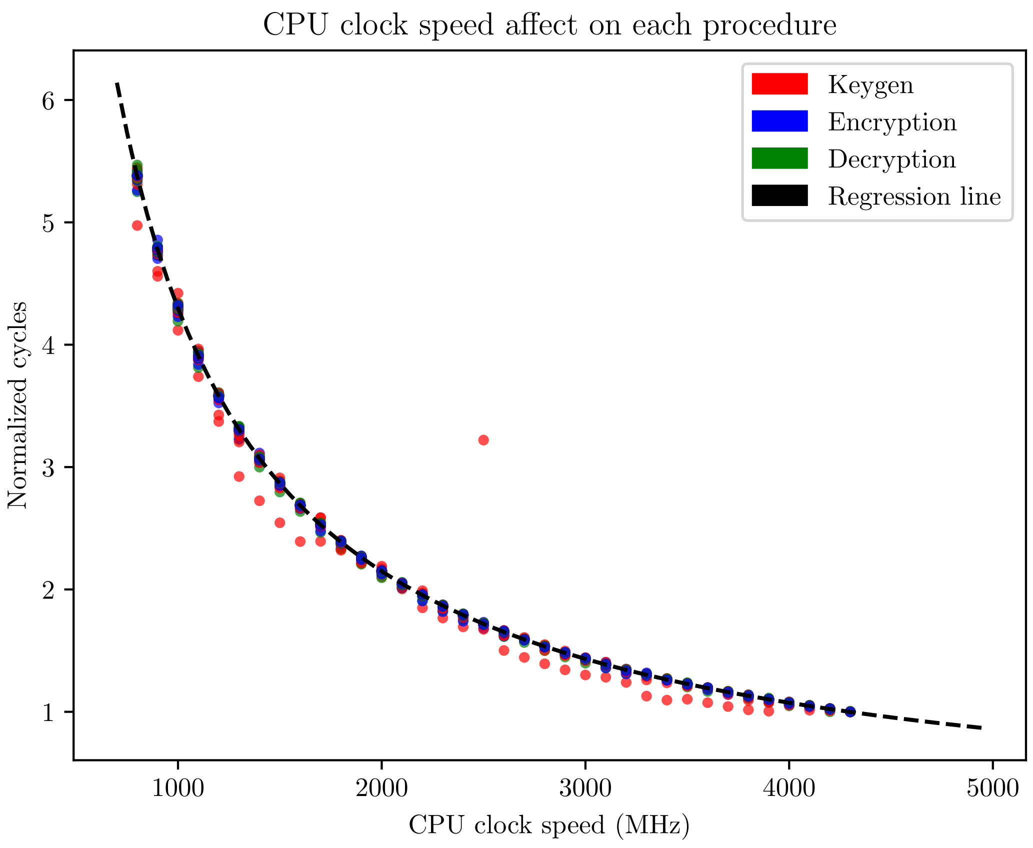

Figure 5.9: The points show the performance difference between different CPU clock speed for each procedure. The points may be hard to see, but graph is just to point out how much the performance overlap if you normalize the data. The regression line is calculated from the points with Python.

5.2 HQC

5.2.1 Old implementation

For the old implementation of HQC we can see in figure 5.10 that the C/C++ flags affects the encryption and decryption more than the keygeneraion. This can be further investegated by looking at figure 5.11. Here we see that there is a significant performance increase when enabling the `-O3` option compared to `-O2` and `-O1`. This performance increase for the encryption procedure is more significant on security level 1, and decreases further when looking higher security levels. A interesting part is that the optimization done by `-O2` has a negative effect on the performance of the encryption and keygeneration compared to `-O1`, but not to the decryption.

The core count has minimal effect on the performance, with a difference between two and four cores of within 0.25% and one core of 0.40%. The cache pressure has a similar

	keygen	encrypted	decrypted
clock speed	-0.9766	-0.7414	-0.8601
memory speed	0.001670	0.002293	0.001866
cache pressure	-0.002170	-0.002489	-0.002404
cores	-0.001858	-0.002638	-0.001021
C/C++ flags	0.005958	0.2982	0.1708

Figure 5.10: The correlation matrix for HQC-128. Keygeneration is close to only be affected by the CPU clock speed. When it comes to the encryption and decryption, the system is much more affected by the C/C++ optimization flags. The other parameters have a small but noticable correlation with the performance.

curve as the overall data, with a performance drop of up to 0.7% when the cache pressure is below 40. The CPU clock speed has the same affect as found in the overall data, following the regression line. These results are similar for all security levels.

Profiling

	Keygen	Encryption	Decryption
`shiftXor`	76.57%	28.35%	21.06%
`lfsr_encode`	0%	41.10%	0%
`repetition_code_decode`	0%	5.15%	31.04%
`vect_fixed_weight`	14.69%	3.46%	2.66%
`array_to_rep_codeword`	0%	17.13%	0%
`gf_mod`	0%	0%	12.20%
`gf_mul`	0%	0%	10.64%

Looking at the profiling of the system we find the function `shiftXor` to be the most time-consuming function for keygeneration, and in the top 2 for encryption and decryption. The percentage increases when the security level increases for this function. This function can therefore be considered the reason keygeneration does not have a significant performance increase with the flag `-O3` over `-O2` and `-O1`. The reason for this is that the function contains binary operations and a variable length for loop. These operations are hard for a compiler to increase the performance of and not much more optimization that can be done. However the AVX operations are not used in the optimized solution for HQC which can increase the performance. This is changed in newer implementations

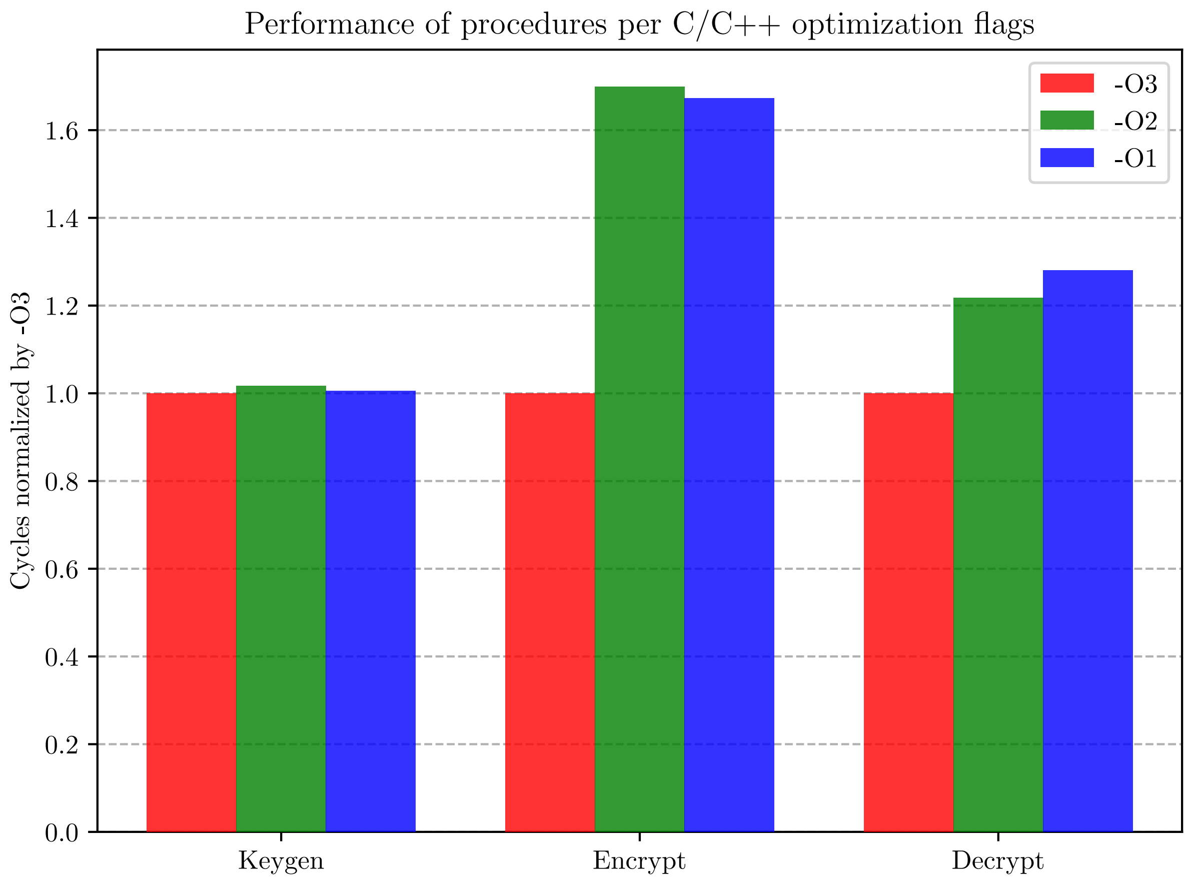

Figure 5.11: The difference each C/C++ optimization flags makes on the procedures for HQC-128. The keygeneration algorithm is barely affected, while the encryption algorithm has a large performance increase between `-O3` and `-O2`. This data is from security level 1. The performance increase for `-O3` in encryption decreases relative to the increase in security level.

of HQC. An implementation of this function in FPGA would most likely be possible and beneficial with a software/hardware solution. The function is called relatively often compared to other functions, so inlining it could be beneficial as long as the cache is not overwhelmed.

The most time-consuming function for encryption is `lfsr_encode`. This function uses binary operation on integers in a fixed size for-loop. This function could take advantage of loop unrolling which can be added manually with the flag `-funroll-all-loops`. As the performance of the encryption procedure increases over 40% from `-O2` to `-O3`, there is a large probability of optimization in this function. The share of the total time consumption decreases for this function when the security level is increased. Optimization of `array_to_rep_codeword` is another function with fixed sized for loops which could be optimized. The `shiftXor` becomes a more significant time consumer when the security levels increases, which in turn decreases the difference in performance between `-O3` and `-O2`.

For the decryption procedure, `repetition_code_decode` is the most time-consuming. The function consists of fixed sized for loops. The loops does contain a `continue` statement which may have an effect on the optimization possibilities. Functions like `gf_mod` and `gf_mul` only contains binary operations, and can therefore take advantage of an FPGA implementation. These functions has a similar share of the total time consumption for the different security levels.

5.2.2 New implementation

	keygen	encrypted	decrypted
clock speed	-0.9739	-0.9721	-0.9568
memory speed	0.001907	0.001279	0.001287
cache pressure	-0.002325	-0.002380	-0.002025
cores	-0.002479	-0.001771	-0.002536
C/C++ flags	0.008674	0.01911	0.05092

Figure 5.12: new-HQC-128 correlation matrix.

The updated submission of HQC comes with a new implementation of the scheme. This implementation is a lot less reliant on the C/C++ optimization flags. It is actually the decryption procedure that is most affected by the flags. In figure 5.13 we see that keygeneration and encryption procedure are similarly affected by the flags.

The minimal affect the C/C++ flags has on the performance may suggest that more optimizations have been done in the code instead of the compiler having to do the optimizations. The higher the security level is, the less affect the flags has on the performance. A reason for these results can be because the flag `-funroll-all-loops` is declared in the compilation of the system. The flag is often known to increase performance as long as the resulting code does not become too large to fit in cache or memory.

The core count have a minimal effect on the performance, with only a 0.3% increase with one core, which suggest there are no parallelized implementation of any functions. The difference between two and four cores is only 0.05%. When looking at the cache pressure I could not find anything standing out to the overall findings. A cache pressure below 40 results in a performance increase of up to 0.6%. The CPU clock speed also have no notable results standing out to the overall data, except the encryption at 800MHz. This value is most likely an outlier and is close to be considered within margin of error. This outlier is not present when looking at higher security levels. The memory speed decreases performance by up to 0.25% from 800MHz to 1600MHz.

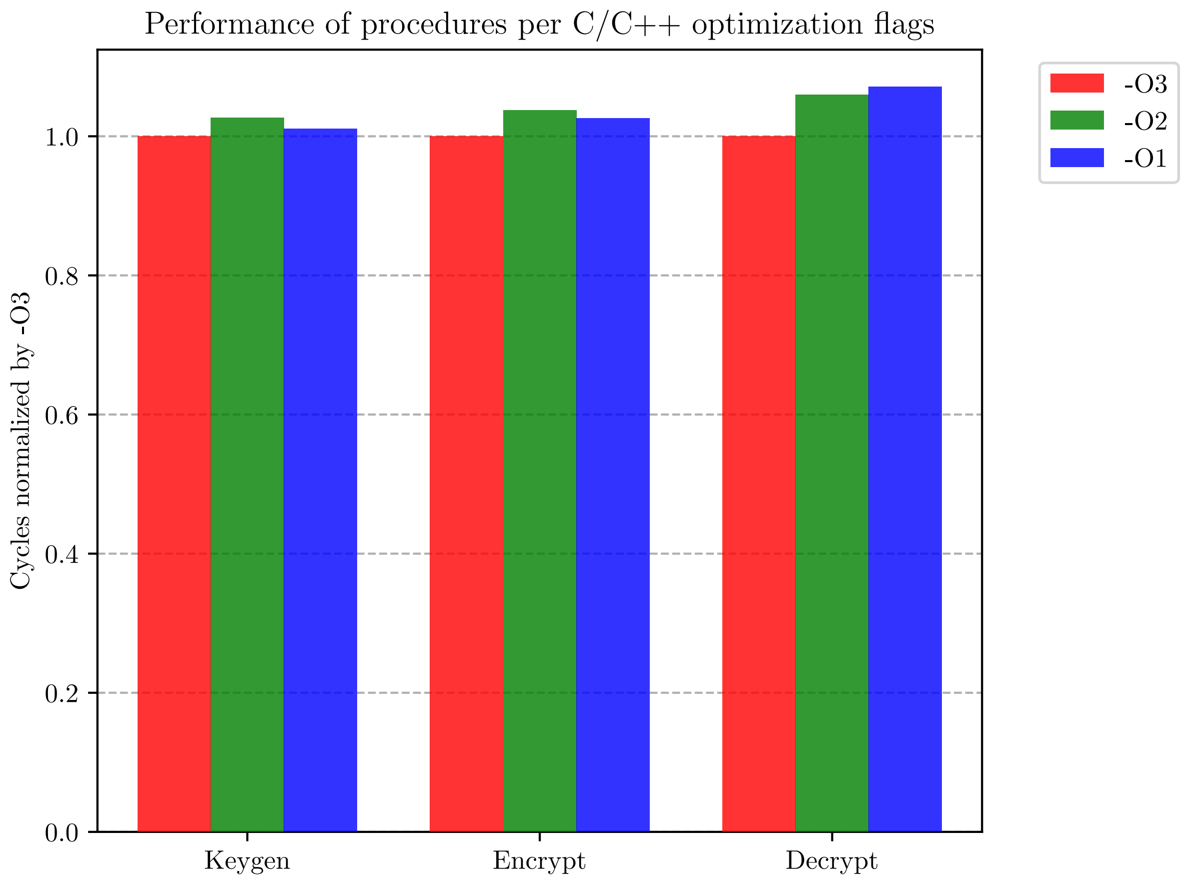

Figure 5.13: The affect of optimization flags for each procedure for new-HQC-128. The performance differences are significantly smaller than original HQC implementation.

Profiling

	Keygen	Encryption	Decryption
`fast_convolution_mult`	71.72%	67.73%	33.97%
`code_encode`	0%	9.28%	2.39%
`vect_set_random_fixed_weight`	7.35%	7.76%	5.26%
`gf_mul`	0%	0%	15.31%
`gf_mod`	0%	0%	7.18%

The function that uses a major part of time consumption of all procedures of the updated HQC is `fast_convolution_mult`. It is clear that this function is therefore one of the reasons why C/C++ optimization flags do not have such a big impact. The task of `fast_convolution_mult` is to compute the product of a polynomial with a sparse polynomial. It consists of multiple for-loops where many of them are fixed size. These will with the `-funroll-all-loops` flag unroll making the function more optimized. The operations in the function are diverse but many of them are binary operations using AVX instructions. These AVX instructions are very efficient way of doing the binary operations. This function has been declared as a `static inline` function. The performance benefit of this is minimal, as the function is only called once for keygen, two times for encryption, and three times for decryption. The share of total time consumption for this function increases when security level is increased.

Another function with a major share of time consumption for encryption is `code_encode`. The goal of this function is to encode the message to a codeword in the BCH code. To solve this goal the function has multiple fixed sized for loops while doing different operations where a big portion of them are binary operations. AVX instructions are also used in this function to optimize the performance. As the C/C++ optimization flag results indicate, there are probably not much more the compiler can do to optimize the function further. The share of total time consumption for this function is reduced to only 2.17% when the security level is increased to 5.

`vect_set_random_fixed_weight` is present in all three procedures with minor time consumption. The function is responsible for generating a random vector of a specific hamming weight. The function consist of for loops with a variable length based on the weight of the vector generated. This makes it hard for the `-funroll-all-loops` to optimize the function. The percentage of total time consumption for the functions stays relatively the same when the security level is increased.

The functions `gf_mul` and `gf_mod` makes up a large portion of the decryption procedure, and is two of the few functions that makes decryption stand apart from keygeneration and decryption. These functions have the purpose of multiplication and modulo. The functions are likely to be optimized in some sort, at least the function calling them, because in a single decryption they will be called approximately 8,300 times each. This makes different optimizations like in-lining and branching optimization more desirable for the compiler. Each of these calls takes around 2-5 nanoseconds to compute. The percentage of total time consumption for these two functions are reduced when security level is increased.

5.2.3 New rmrs implementation

The rmrs HQC cryptosystem is similar to the new HQC with the exception of using different parameters and different codes for error correcting. The implementation of the system is also very similar except the extra functions needed for the error correction. Therefore we see a similar correlation matrix in figure 5.14 to new HQC. Although there are some changes when it comes to C/C++ optimization flags.

The C/C++ optimization flags effect on the performance figures in 5.15 show that the encryption procedure is close to being as affected by the optimization flags as the decryption procedure. In the new-HQC case the decryption procedure had higher performance differences for decryption than encryption. The performance figures with respect to the other variables like CPU clock speed, memory speed, cache pressure and cores are within

	keygen	encrypted	decrypted
clock speed	-0.9742	-0.9683	-0.9696
memory speed	0.001408	0.001727	0.001611
cache pressure	-0.002509	-0.002482	-0.002332
cores	-0.002605	-0.002041	-0.002120
C/C++ flags	0.01238	0.02731	0.03827

Figure 5.14: New rmrs HQC correlation matrix.

margin of error to have the same affect on the system as the new-HQC cryptosystem.

Profiling

	Keygen	Encryption	Decryption
`fast_convolution_mult`	71.12%	56.29%	30.32%
`reed_muller_decode`	0%	0%	28.06%
`vect_set_random_fixed_weight`	6.31%	6.40%	3.20%
`vect_set_random_fixed_weight_by_coordinates`	7.04%	4.48%	1.69%
`gf_mul`	0%	8.96%	7.91%
`gf_mod`	0%	5.33%	5.65%
`compute_syndromes`	0%	0%	7.91%

The functions with similar names as new HQC have the same implementation and I will therefore skip them. The `reed_muller_decode` is the function for decoding the message from the codeword using Reed-Muller decoding. The function consists of a fixed size for-loop and calls to other functions. At first glance it doesn't make sense that this function should have high time consumption as it is only a small for-loop with some light operations and calls to other functions. As it turns out these functions are inline functions, which will expand into the function for avoiding using the processing power of calling the function and returning. These inline functions consists of fixed sized for-loops and binary operations where some are implemented in AVX instruction, increasing the optimization of the code. The share of the total time consumption decreases for this function when the security level is increased.

The `vect_set_random_fixed_weight_by_coordinates` function is similarly implemented as `vect_set_random_fixed_weight`, and has therefore similar optimization done.

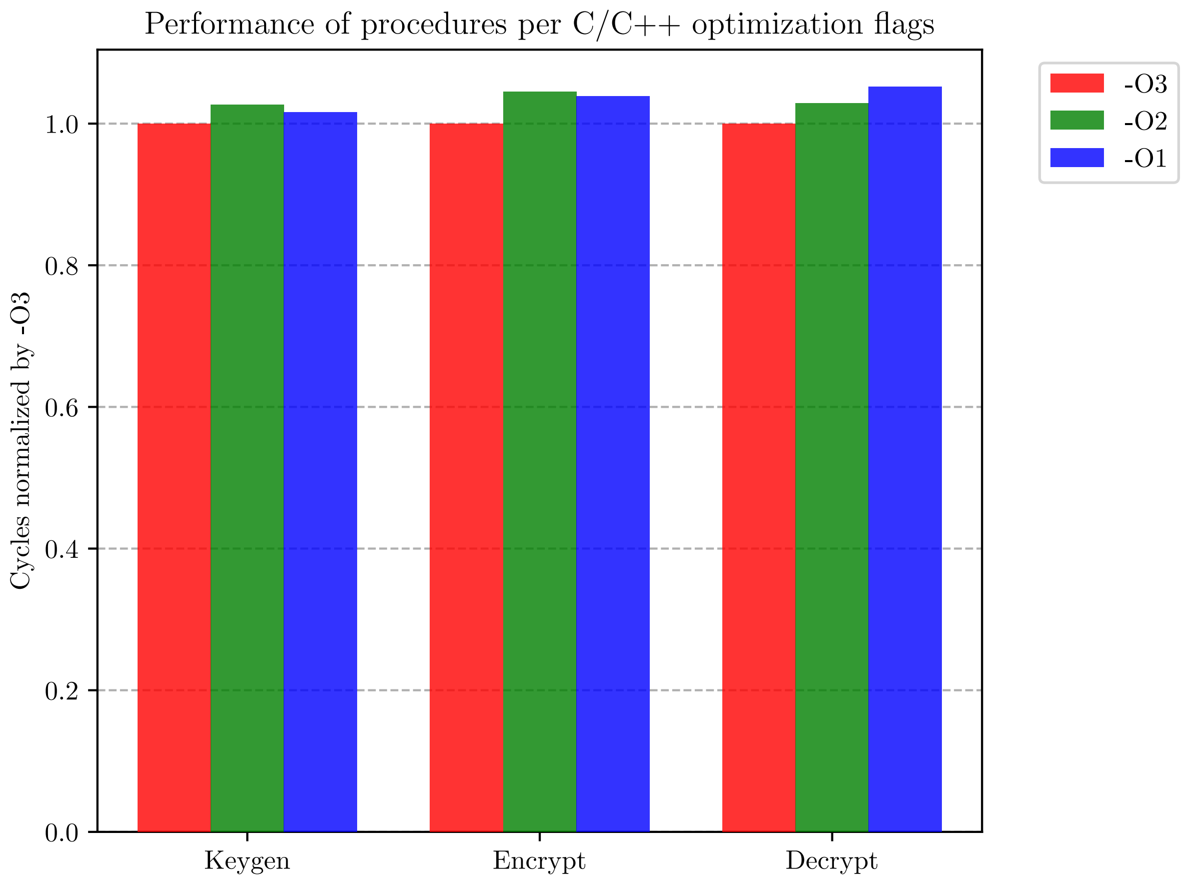

Figure 5.15: The performance differences when looking at optimization flags for new-HQC-rmrs.

The last function named `compute_syndromes` computes syndromes array of a codeword. It only contains for-loops of up to 1920 iterations. In the loops there are binary and integers operations on arrays. The percentage of total time consumption decreases for this function when the security level is increased.

5.2.4 Comparison

The updated submission with the new implementation is clearly more optimized than the original implementation. The original optimized submission do not use AVX instructions, which is standard on most modern processors. The `-funroll-all-loops` may give a performance boost when the implementation is used as a standalone solution. However when used as a library in a larger project this may become a problem as the extra code in the compiled program can push more important parts of the program out of the cache.

	Keygen	Encryption	Decryption
HQC-128	-23%	-43%	62%
HQC-rmrs-128	-32%	-48%	59%
HQC-192	-28%	-44%	64%
HQC-rmrs-192	-36%	-50%	66%
HQC-256	-31%	-45%	58%
HQC-rmrs-256	-38%	-51%	61%

As the old parameters are not broken, the old implementation is still usable, even though the parameters are generally larger than the new submission. The performance of keygeneration and encryption has increased performance, while the decryption has become slower. This can be caused by a new decoding algorithm that is introduced in HQC-rmrs, or that the parameters generally require more computation to decrypt. This is still suprising as using AVX operations are supposed to increase performance significantly. The public key size has decreased for normal HQC by around 50%, and HQC-rmrs has reduced the public key by around 60%.

5.3 LAC

	keygen	encrypted	decrypted
clock speed	-0.6773	-0.6476	-0.6116
memory speed	0.001698	0.001686	0.001423
cache pressure	-0.002712	-0.001948	-0.002150
cores	-0.001324	-0.002354	-0.001468
C/C++ flags	0.2510	0.2764	0.3011

Figure 5.16: LAC-128 correlation matrix.

The LAC cryptosystem has a high correlation of optimization flag and performance. This suggests that there are large parts of the code which can be optimized by the compiler. The LAC system is significantly affected if we look at the results in 5.17. We see that the different procedures have a performance decrease of up to 250%. This performance difference increases with the security level. Interesting point is that the `-O2` flag decreases performance significantly compared to `-O1`. The `-O2` flag have the same optimization as `-O1` but with some extra optimization that does not increase size of the compiled program. This means some of these optimizations done by the compiler have a negative effect on the performance.

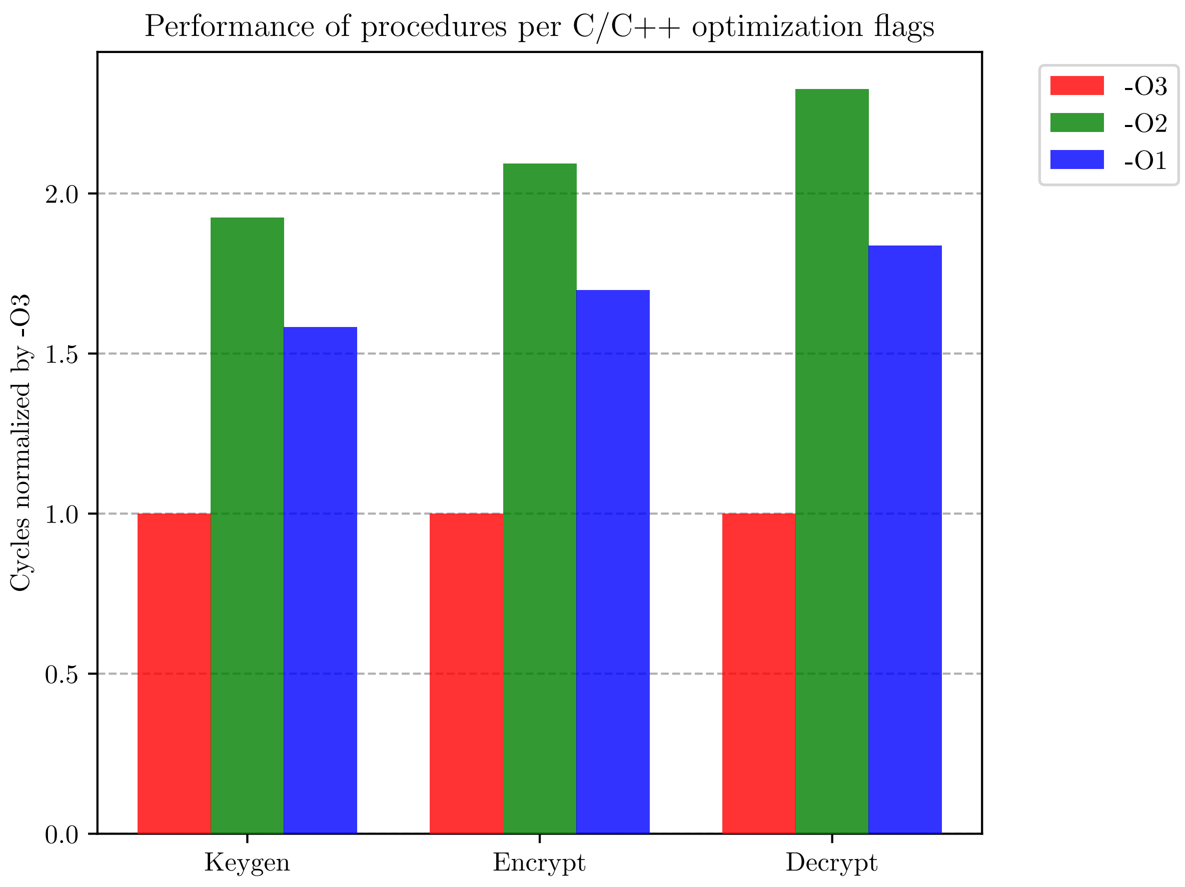

Figure 5.17: Performance measurements for each optimization flag for LAC-128.

The core count has the same effect on all security levels, where the procedures has a 0.3-0.4% decrease in performance with a single core compared to multiple. The difference between two and four cores are within margin of error, suggesting there is not any parallelization of the code. The cache pressure seems to have a similar effect on LAC as it has on the overall data. The performance decrease with a cache pressure below 40 is found to be between 0.6% and 0.8%. Looking at the cache pressure with only `-O3` the perfomance hit below 40 is between 1.5% and 2.0%. The CPU clock speed has a similar curve as the regression line with some small outliers when the security level increases. This is most likely because it was not able to generate a large enough pool of measurements in 300ms to return without outliers.

Profiling

	Keygen	Encryption	Decryption
`poly_aff`	91.66%	92.81%	0%
`poly_mul`	0%	0%	95.77%
`gen_psi_fix_ham`	7.37%	4.12%	0%

The whole runtime of the LAC implementation can almost be specified by only three functions. This means most of the optimization needed is to focus on these functions. The `poly_aff` function has the goal of computing $as+e$ where a, s and e are input arguments. This function contains some fixed sized for-loops, and both memory and binary operations. There is no specified flag for unrolling the for loops in the compilation file. Since the function is only called once or twice during the procedures, I would imagine it is not much performance increase of adding the `-funroll-all-loops`. The binary operation in the function seems to be what is slowing the function down. Most of the binary operation in the function are bitwise AND operations. These are easily implemented in an FPGA and could increase the performance of the system by using a software/hardware solution. The share of the total time consumption increases for this function when the security level increases.

Function responsible for 95.77% of time consumption of decryption is `poly_mul`. This function is similar to `poly_aff`, but instead of computing $as + e$ it computes as from input arguments. The operations in the function is therefore very similar and could also have a performance increase with a implementation of bitwise AND in an FPGA using a software/hardware solution. The share of total time consumption increases also for this function when security level is increased.

`gen_psi_fix_ham` is responsible for generating small random error and secret vectors with fixed hamming weight. This function consists of variable sized for-loops and while-loops. The operations in the function are integer operations with a few calls to functions like hash-functions. The share of total time consumption is reduced when security level is increased for this function.

Updated implementation

LAC does have an updated submission with updated implementation. I did not have time to make performance measurements and compare the original implementation with the new.

5.4 LEDAcrypt

LEDAcrypt is the cryptosystem with the highest time consumption for each procedure. In return it has one of the largest messages possible to encrypt in a single ciphertext. This could for example make it useful for broadcasting a message to different computer

	keygen	encrypted	decrypted
clock speed	-0.7521	-0.9432	-0.8728
memory speed	0.001071	0.001508	0.001995
cache pressure	-0.002175	-0.001913	-0.002201
cores	-0.01039	-0.002907	-0.004545
C/C++ flags	0.2545	0.05708	0.1567

Figure 5.18

with different public keys. The system is one of the most efficient when it comes to performance per byte encrypted. However the high time consumption makes it harder to get performance measurements correct as the time frame of 300ms becomes thin for finding a representative result. The longer a procedure runs, the higher possibility is for another process to affect the result, but also fewer results to find the median from.

The correlation of the optimization flags varies between the procedures. Keygeneration have a correlation value of 0.2545 and encryption has 0.0571. This suggest that the time consuming functions for each procedure are different and may have different optimizations.

In figure 5.19 we see why we get a big difference in correlation between the procedures. The keygeneration procedure has a large performance penalty of compiling with `-O1`. This penalty is there in encryption and decryption procedures also, but smaller. The most interesting part of the figure is the fact that `-O2` is faster than `-O3` for encryption. This is the first example we've found for this case. There are several reasons why the results ended up like that. A reason can be that the compiled code size with `-O3` becomes too large to fit everything in the CPU cache and therefore has to move some of the code to the slower RAM. It may also be that `-O3` just optimized some code the wrong way causing it to run slower. Another reason can be just pure random error, although this would be more likely to happen on keygeneration since this procedure has significantly higher runtime and therefore room for errors. This advantage for `-O2` is not present in higher security levels. For security level 3 the `-O2` flag is more efficient than `-O3` for the keygeneration procedure.

The affect of core count is most noticeable on keygeneration algorithm, with a performance decrease of up to 2% as we can see in figure 5.20. Compared to the overall data only have a 0.4% decrease in performance, this is a significant difference. It might seem like the procedure has some sort of parallelization, but this is not necessarily true. The difference may be the result of the long runtime each keygeneration procedure have. The combination of long runtime per procedure and running on only a single core might be the reason results are more affected than other systems. This is similar for all security levels.

The cache pressure effect on performance is similar to the overall findings for security

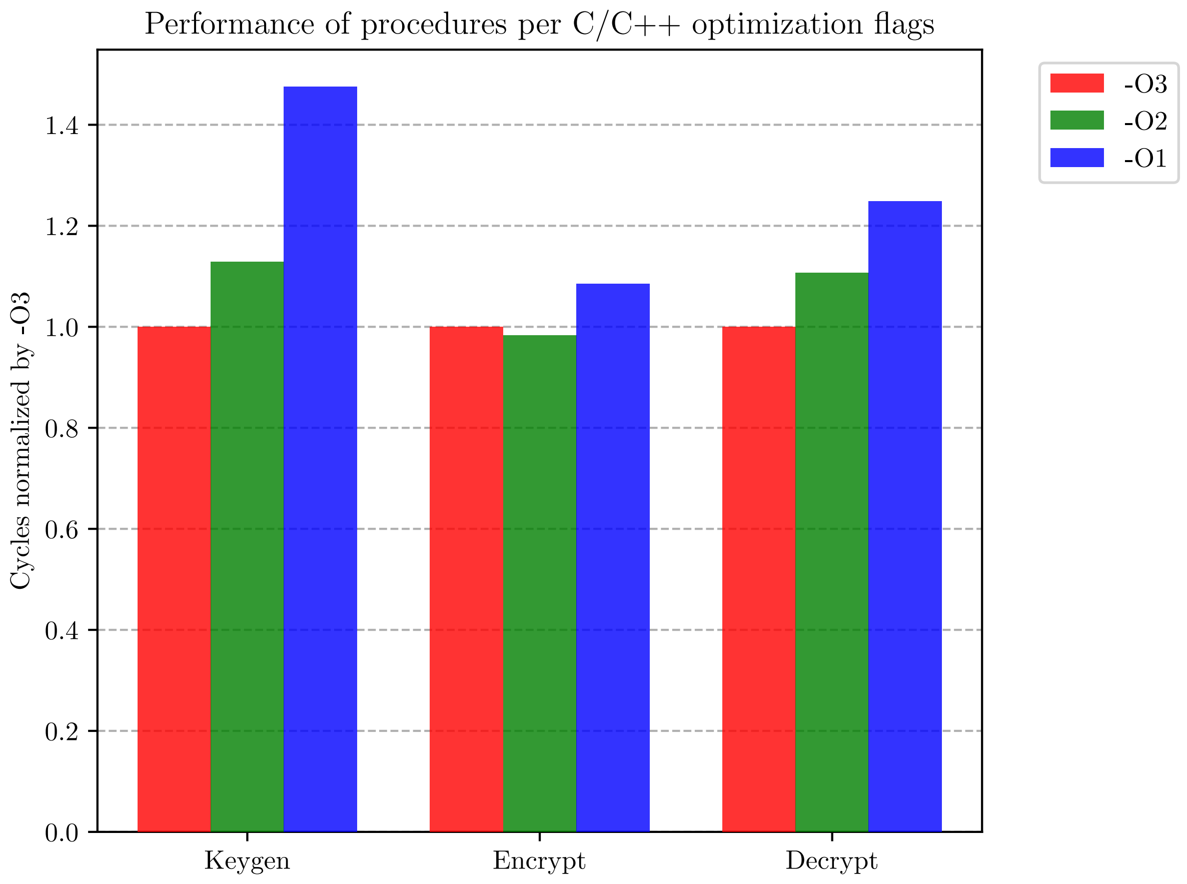

Figure 5.19: Performance measurements for each optimization flag for LEDAcrypt-PKC-1.

level 1, however this is not the case for security level 3 and 5. The keygeneration in these security levels fluctuates significantly with up to 2.5% performance decrease. Looking at each separate optimization flag the fluctuation increases to 8%. It is hard to conclude anything from this fluctuation, but it could be outliers as the sample size was not large enough, causing erroneous results. The fluctuations of encryption and decryption however stayed consistent with the overall findings.

Except some outliers, the performance based on CPU clock and memory speed follows the curve from the overall findings.

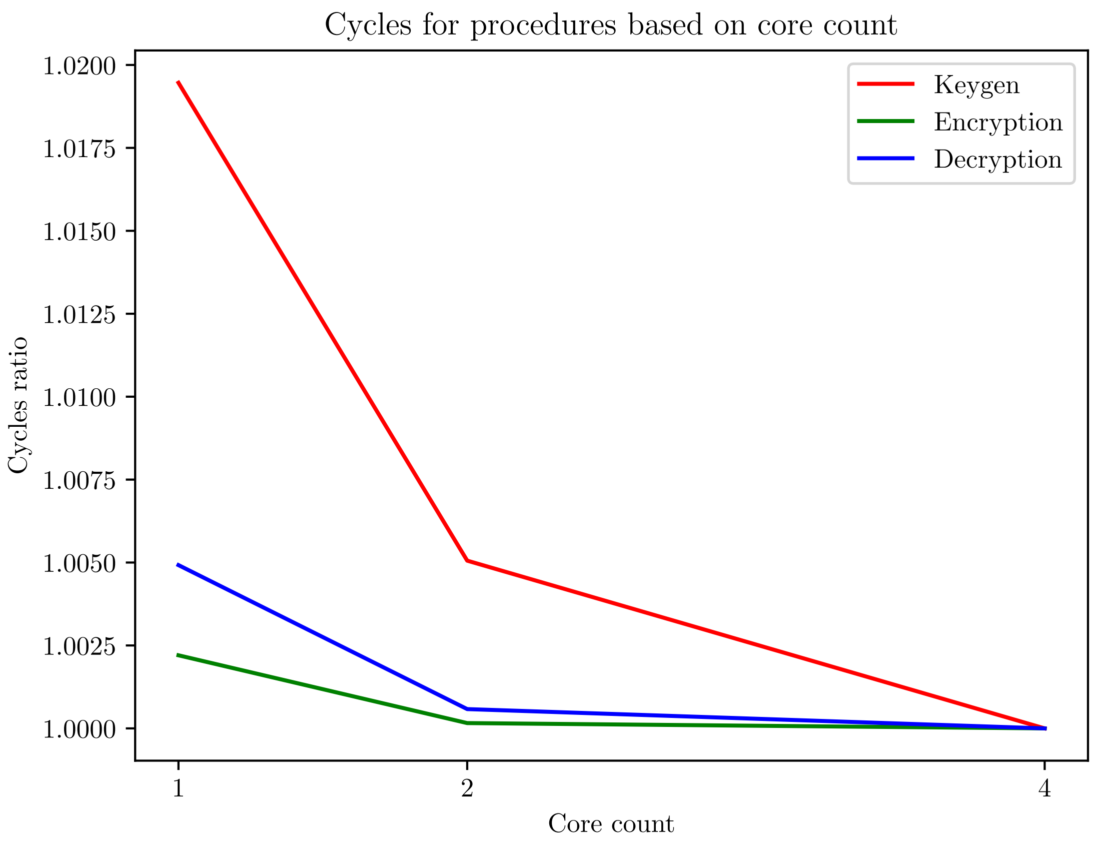

Figure 5.20: Performance difference at different core count for LEDAcrypt-PKC-5.

Profiling

	Keygen	Encryption	Decryption
`DFR_test`	97.13%	6.93%	4.87%
`bf_decoding`	0%	0%	32.94%
`bytestream_into_poly_seq`	0%	23.44%	0%
`constant_weight_to_binary_approximate`	0%	0%	17.45%
`bitstream_read`	0%	17.06%	0%
`gf2x_mul_TC3`	0%	10.54%	0%

In all security levels it is `DFR_test` that is the major time consumer in keygeneration procedure. The function consists of multiple fixed sized for-loops. The function is a test to check if the code attains the desired decryption failure rate. It is therefore not a determined number of calls to this function, but until it returns true. This is might be why we get such difference between the security levels. The operations in the function are integers operations on arrays, and calls to sorting algorithm quicksort. The reason the function has such high time consumption for encryption and decryption is that keygeneration has to run once before we can use each of them. With a timelimit of 15000ms it was still not enough to limit the effect of the one keygeneration call.

The decryption has many different major time consumer, but the most dominant is `bf_decoding`. This function will change based on the computer to get the optimal optimization based on specification of the computer. The optimization is just checking that the computer running has the correct instruction set and that it is a 64-bit system. The goal of the function is to decode a codeword to the correct message. The function uses multiple fixed sized for-loops with operations using the AVX instruction set. The share of total time consumption increases for this function in decryption if security level is increased and remove the `DFR_test` time consumption.

The function `bytestream_into_poly_seq` consists of both a variable and fixed sized for-loop. The function is only called once per encryption, so a unroll of the fixed sized loop would not give much performance gains. The operations in the function are both on integers and binary data. There are some multiplication of fixed integers inside the for-loops which could be computed before running the loop. Although this would probably not increase the performance much and the compiler may even do the optimization itself with the different optimization flags. The share of total time consumption decreases for this function when security level increases.

`constant_weight_to_binary_approximate` is another major time consumer in decryption. The goal of the function is to encode a bit string to a constant weight polynomial vector. The function consists of two fixed sized for-loops with function calls and integer operations. The function is only called once per decryption and therefore do not have much performance boost of inlining or unrolling loops. The share of the total time consumption is reduced when security level is increased.

A function that is called around 52,000 times for security level 1 and 150,000 for security level 5 each encryption is `bitstream_read`. The function consists of mostly binary operations and a while loop. Since the function is called so many times in an encryption, small changes can have a relatively large impact on the speed of the encryption. Inlining this function could be beneficial, as it is only called from three different places. The share of total time consumption is increased for the function when security level is increased.

The `gf2x_mul_TC3` function is implemented from the Toom-Cook 3 algorithm and is computing a multiplication of two polynomials. The function consists of multiple calls to subfunctions used for operations of the data. These subfunctions are static inline functions using AVX instructions for optimization. The function also consists of some variable sized for-loops with binary operations when combining the results of subfunctions. Percentage of total time consumption in encryption increases for this function when security level increases.

Updated implementation

LEDAcrypt also has a updated submission with a new implementation. However I did not have time to look at the performance of the updated implementation.

5.5 ROLLO-II

5.5.1 Old implementation

	keygen	encrypted	decrypted
clock speed	-0.9369	-0.9257	-0.8329
memory speed	0.001536	0.002231	0.001652
cache pressure	-0.002233	-0.002554	-0.002186
cores	-0.003272	-0.003358	-0.002884
C/C++ flags	0.07961	0.09030	0.1811

Figure 5.21: The correlation matrix for ROLLO-II-128.

The old implementation of ROLLO-II comes with flags in the compilation enabling instruction set that improve performance. These flags are `-mpclmul`, `-msse4.1` `-mavx` and `-mavx2`. According to the correlation matrix 5.21 we see that the C/C++ optimization flags has a relatively low correlation with keygeneration, while it increases for encryption and especially decryption. In figure 5.22 we see that decryption is between 30-40% slower with `-O1` and `-O2` compared to `-O3`. The difference of keygeneration and encryption is less substantial. If we look at different in security level, we find the `-O3` flag to get a larger performance difference to the other flags. This means functions that have more optimization potential are more used at higher security levels.

The cache pressure has a performance decrease of up to 0.6% when cache pressure is below 40. When looking at the performance at 40 and above, it is possible to see a linear performance decrease when increasing the cache pressure. However this can be considered to be within margin of error. Looking at the cache pressure while separating the optimization flags, we see that the `-O3` flag is the only flag having a performance decrease below 40 of up to 1.7%. The `-O2` flag looks like a linear decrease in performance the higher the cache pressure is, but the fluctuations are too large to say something for certain. Last flag `-O1` has most of the performance differences within 0.2% which can be considered within margin of error.

The effect of core count is close to the same as the overall findings, with a performance decrease of about 0.3%-0.4%. The encryption seems to be negatively affected by increasing the core count to 4 cores, but this is most likely erroneous results. CPU and memory

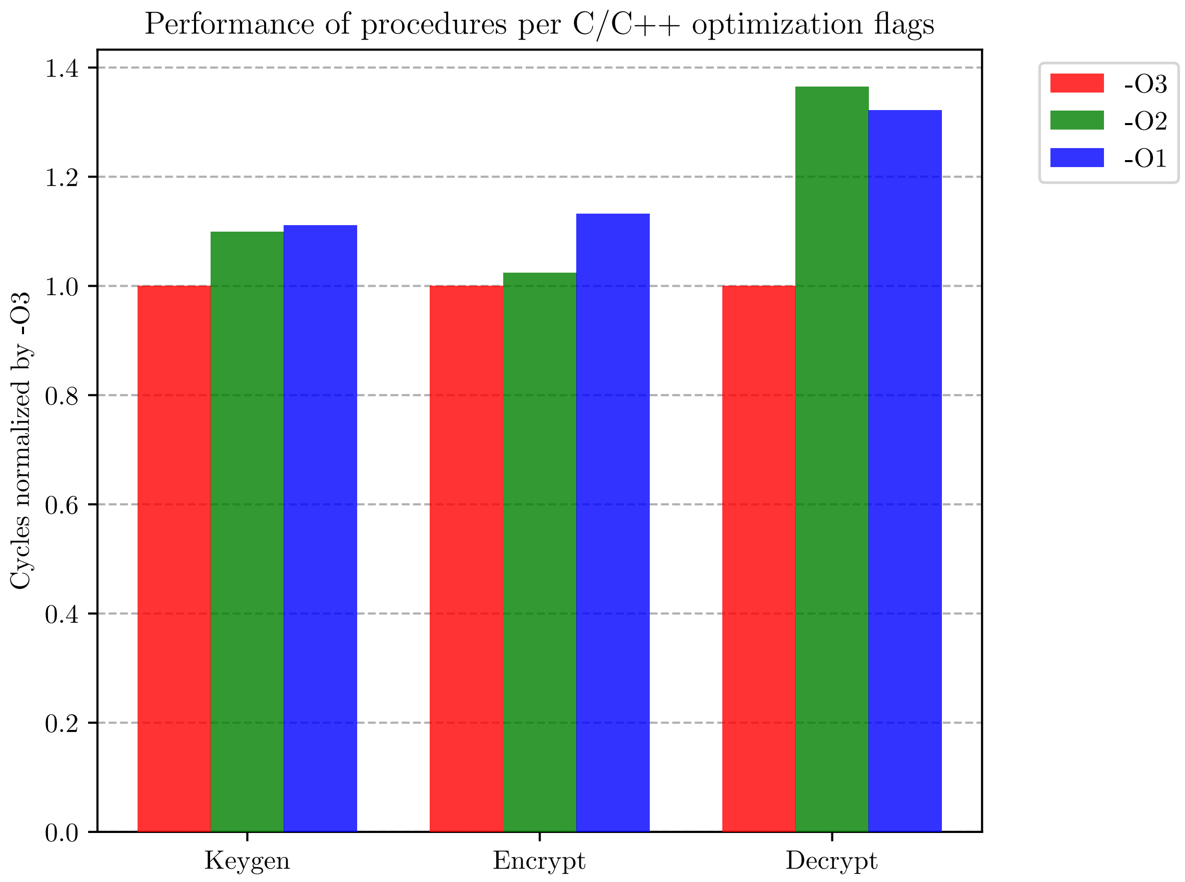

Figure 5.22: Performance measurements for each optimization flag for ROLLO-II-128.

speed seems to affect the system in the same manner as the overall findings, and do not have much deviations that could indicate more or less bottleneck on the components.

Profiling

	Keygen	Encryption	Decryption
`ffi_elt_mul`	15.89%	7.37%	3.50%
`ffi_elt_ur_mul`	12.92%	8.71%	1.17%
`ffi_elt_reduce`	15.68%	10.27%	5.61%
`ffi_elt_add`	7.84%	13.39%	12.62%
`ffi_elt_get_degree`	2.33%	1.79%	21.03%
`ffi_vspace_intersection`	0%	0%	21.03%
`kar_mul`	3.81%	31.25%	7.48%

It is interesting that `ffi_elt_mul` is one of the major time consumer in the system. The function consists of creating a temporary variable and calls to two different functions. The reason it has high time consumption is most likely because it is called almost 50,000 per keygeneration procedure. The goal of this function is to multiply two finite field elements and reduce the result to within the finite field. To increase performance it could be beneficial to inline this function to where it is called from. This is most likely part of what the compiler optimization is doing with the function for `-O3`. There are 11 different places this function is called, which would increase the compiled code size by a small amount. Percentage of total time consumption decreases for this function when the security level is increased.

The functions `ffi_elt_mul` is calling are `ffi_elt_ur_mul` and `ffi_elt_reduce`. The goal of `ffi_elt_ur_mul` is just to multiply two finite field elements. In the optimized implementation, AVX instructions are used to achieve this result. The optimization flags probably do not increase the performance of this function much, unless inlining is done. The operations in the function can be implemented in an FPGA to increase the performance further if a software/hardware solution is used. Percentage of total time consumption is reduced for this function when the security level is increased.

The goal of `ffi_elt_reduce` is to reduce a finite field element based on an irreducible element f. This function is different for each security level because f is hardcoded into the function. Having the reduction in the function be based on f as a variable would most likely decrease performance. The function consists of variable initialization and binary operations. Percentage of total time consumption is also reduced for this function when security level is increased.

For `ffi_elt_add` the function is optimized by using AVX instructions. The goal is to add two finite field elements. This function is called 70,000-80,000 times per keygeneration and decryption, while around 22,000 for encryption. It would benefit by inlining the function to the places where it is called as long as it is room in the memory or cache. The function is called from 19 different places and would therefore increase the size of the compiled code a small amount. It could also be implemented in an FPGA with a software/hardware solution. Percentage of total time consumption increases when security level increase, except for in keygeneration.

The major time consumers for decryption are `ffi_elt_get_degree` and `ffi_vspace_intersection`. The `ffi_elt_get_degree` is a function for finding the degree of the finite field element. It is called around 83,000 times each decryption, which means the decryption can get a significant performance increase if the function is inlined. The function uses assembly commands to find the degree of the finite field. A developer using assembly in C usually do not increase performance, and often actually decreases performance as the compiler is not able to optimize it itself. However there are cases where it can increase performance, and the command used in assembly is bit scan reverse (`bsr`). The other function `ffi_vspace_intersection` is only called around 14 times per decryption and will therefore not have much of a performance increase by inlining, even decrease performance if there is not enough memory or cache. The percentage of time consumption for these functions are reduced when security level is increased.

The `kar_mul` function is the largest time consumer in encryption. It contains calls to other functions and itself recursively. It is the recursive calls making this function have such high time consumption. When a function is recursive, it is often possible to parallelize it. The percentage of time consumption is increased when security level is increased.

5.5.2 New implementation

	keygen	encrypted	decrypted
clock speed	-0.9745	-0.9291	-0.8799
memory speed	0.001632	0.001649	0.001712
cache pressure	-0.002149	-0.002478	-0.002371
cores	-0.003419	-0.002208	-0.002699
C/C++ flags	0.01928	0.08841	0.1489

Figure 5.23: The correlation table for newROLLO-II-128.

The updated submission of ROLLO came with new parameters and new implementation. This makes it hard to directly compare performance of the old and new implementation, because the affect of the parameters on performance. The new implementation comes with the same `-mpclmul`, `-msse4.1`, `-mavx` and `-mavx2` flags for compiling the code. This new implementation has even lower correlation between C/C++ optimization flags and performance. As we see in figure 5.24 it shows that keygeneration procedure is barely affected by the optimization flags. The encryption has a small performance increase when changing the optimization flag from `-O1` to `-O3` of about 12%. However the decryption have the largest performance difference of about 23%. If we look at each security level, the benefit of `-O3` increases for keygeneration procedure, while encryption and decryption have close to the same benefit as in security level 1.

Performance with different core count is similar to the overall data, with some outliers. The single core performance is up to 0.4% slower while the difference of two and four cores are normally within 0.1%. There is however an outlier at security level 5 where the encryption is almost 0.5% slower at four cores. The cache pressure has more variations when the value is below 40 than the overall findings, the performance decrease is mostly at 0.5% and above. The rest of the cache pressure measurements seem to have a slow linear growth, but this is hard to tell as most of the measurements is within 0.2% difference. Looking at only the `-O3` flag this growth becomes more clear at security level 5 in figure 5.25. Interesting fact is that encryption seems to be generally more affected by the cache pressure than the other procedures. This fact is however only prevalent in security level 5. At the other optimization flags the measurements are so volatile it is hard to find any

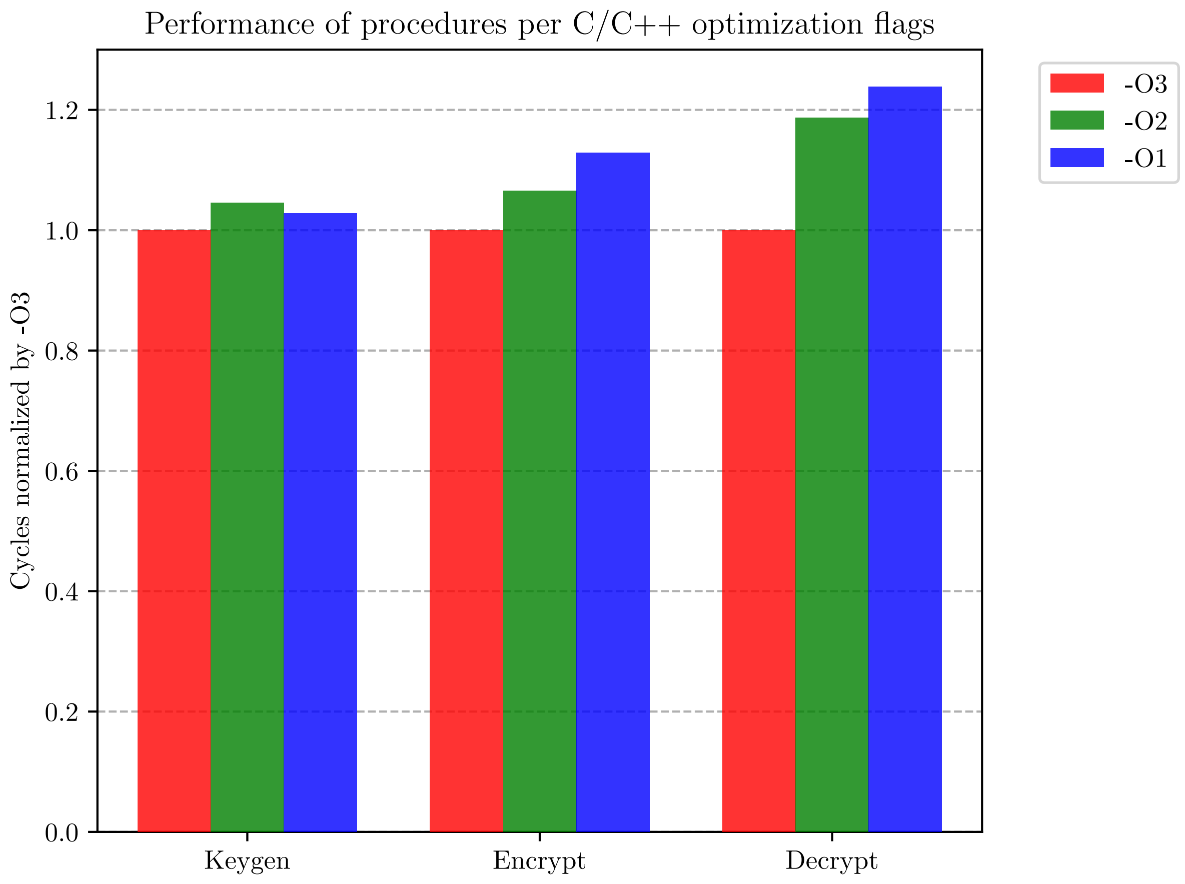

Figure 5.24: Performance measurements for each optimization flag for newROLLO-II-128.

pattern except for the fact that encryption seems to be more affected than keygeneration and decryption.

The memory speed and CPU clock speed seem to have a similar affect on the system as found in the overall findings.

Profiling

	Keygen	Encryption	Decryption
`rbc_kar_mul`	3.60%	29.27%	22.89%
`rbc_elt_reduce`	14.86%	8.13%	5.97%
`rbc_elt_add`	10.81%	16.67%	18.91%
`rbc_elt_mul`	12.16%	9.76%	5.47%
`rbc_elt_ur_mul`	14.64%	6.50%	4.73%

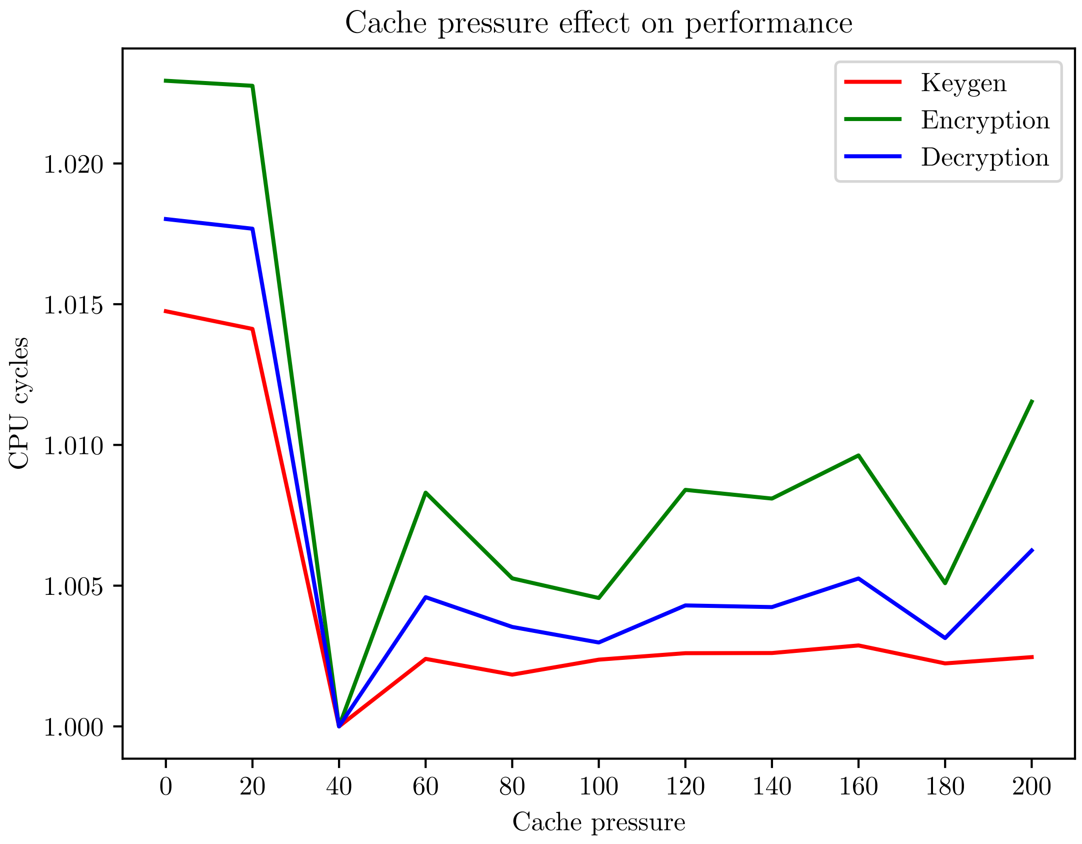

Figure 5.25: Performance measurements based on cache pressure for newROLLO-II-128 with `-O3` optimization.

The function `rbc_kar_mul` is clearly the largest time consumer for encryption and decryption. This function does not contain any large multiplications or additions, but a number of calls to other functions handling binary and integer operations. This means the way this function consumes time is by function calls and iterating through for-loops. These are areas where the compiler often optimize when an optimization flag is used. This can be seen in figure 5.24 as encryption and decryption are the procedure having most performance gain from `-O1` to `-O3`. Percentage of total time consumption increases for encryption and decreases for decryption when the security level is increased.

`rbc_elt_reduce` consists of binary operation where the goal is to reduce a finite field element. The function is the largest time consumer in keygeneration, and is called almost 50,000 times per keygeneration. This makes small improvements in the function actually major improvements for the whole procedure. The operations done in the function are mostly XOR and bitshift operations, and is different for each security level because the irreducible element is hardcoded into the function. The function can be implemented in an FPGA to create a software/hardware solution, increasing performance. Inlining the function could also increase performance without a significant increase in code size, as the function is only called from two different places. The share of time consumption is decreased when security level is increased.

The goal of the function `rbc_elt_add` is to compute the addition of two finite field elements. The function consists of operations using the AVX instruction set for optimizing the procedures of the system. The function is called around 82,000 times for the decryption procedure. There is not much optimization the compiler can do with the AVX instructions, but since the function is called so many times, it is possible there could be performance gains by inlining the function to remove the function call and return overhead. This could affect the performance negatively if there is not enough space in cache or memory. Percentage of total time consumption decreases when the security level increases.

`rbc_elt_mul` and `rbc_elt_ur_mul` are the replacements of `ffi_elt_mul` and `ffi_elt_ur_mul`. The functions have the same purpose as the corresponding function in the old implementation. The `rbc_elt_mul` function is only creating a temporary variable and calling three different functions. As this function is called 80,000 times in keygeneration, small improvements can have a great effects on the whole system. It would most likely give a performance boost if this function were to be inlined places where it is called, however this is not certain. It would increase the compiled code size a small amount as it is called from 11 different places. The function `rbc_elt_ur_mul` is implemented using SSE instruction set from the `-mpclmul` flag for optimized operations. I did not look closer at why they did not use AVX instead. The compiler probably is not able to do much optimization of the function except for inlining. The percentage of time consumption decreases for `rbc_elt_mul` and increases for `rbc_elt_ur_mul` when the security level is increased.

5.5.3 Comparison

The old implementation of ROLLO-II is considered broken, as the parameters for each security level does not achieve the claimed security level. This causes the new implementation of ROLLO-II to come with new parameters, making a direct comparison of the implementations hard, because different parameters will give different performance. The update does come with some noticeable changes, like the ffi library has been extracted and renamed to the rbc library. This library is now a public library making it easier for optimization which would benefit all cryptosystems using it.

	Keygen	Encryption	Decryption
ROLLO-II-128	56%	25%	-29%
ROLLO-II-192	27%	5%	-40%
ROLLO-II-256	57%	16%	-39%

The total size of the parameters did not increase much from the updated parameters. As we can see the performance comparison, the keygeneration and encryption dropped in performance, while the decryption performance increased. Especially for security level 5, where the public key only increased 2.6%. This leads to the question if increase in parameters can lead to a faster decryption, or if optimization done in the new implementation have increased performance. Most of the time-consuming functions in each implementation seem to be similar and not much different optimization between them. Functions increasing in time consumption is the `rbc_kar_mul` and `rbc_elt_add` functions, where `kar_mul` and `ffi_elt_add` are the corresponding functions in the old implementation. It may be this reason the decryption is more efficient.

5.6 RQC

5.6.1 Old implementation

	keygen	encrypted	decrypted
clock speed	-0.9413	-0.9350	-0.8986
memory speed	0.001858	0.002247	0.001832
cache pressure	-0.002113	-0.002108	-0.002272
cores	-0.002810	-0.003686	-0.002771
C/C++ flags	0.06336	0.06325	0.1260

Figure 5.26: The correlation matrix for RQC-128.

The implementation of the old RQC uses many of the same components as the old implementation of ROLLO-II. This means we will most likely get many of the same correlations and graphs. However there might be a difference in what functions are the major time consumers. The correlation between C/C++ optimization flags and performance is quite low, similar to ROLLO-II. In figure 5.27 we see that the affect of optimization flags have the most impact on decryption procedure, with keygeneration and encryption affected similarly. The affect of these flags decrease when the security level is increased.

The affect of cache pressure on performance is more volatile at lower security levels, especially for the encryption procedure. When looking at the cache perssure with `-O3` flag in figure 5.28 it is possible to see this trend. The performance hit of keygeneration and decryption is around 1.5% when cache pressure is below 40, and is almost within 0.5% performance difference at 40 and above. The encryption procedure however has a 2.5% performance hit at cache pressure below 40, and above 40 has a performance hit of above 1.0%. This result is not visible for security level 3 and 5, which follows the performance difference like the overall findings. Affect of cache pressure when using `-O1` and `-O2` did not show any particular pattern, but the result were more volatile than overall findings with performance differences of up to 0.5%.

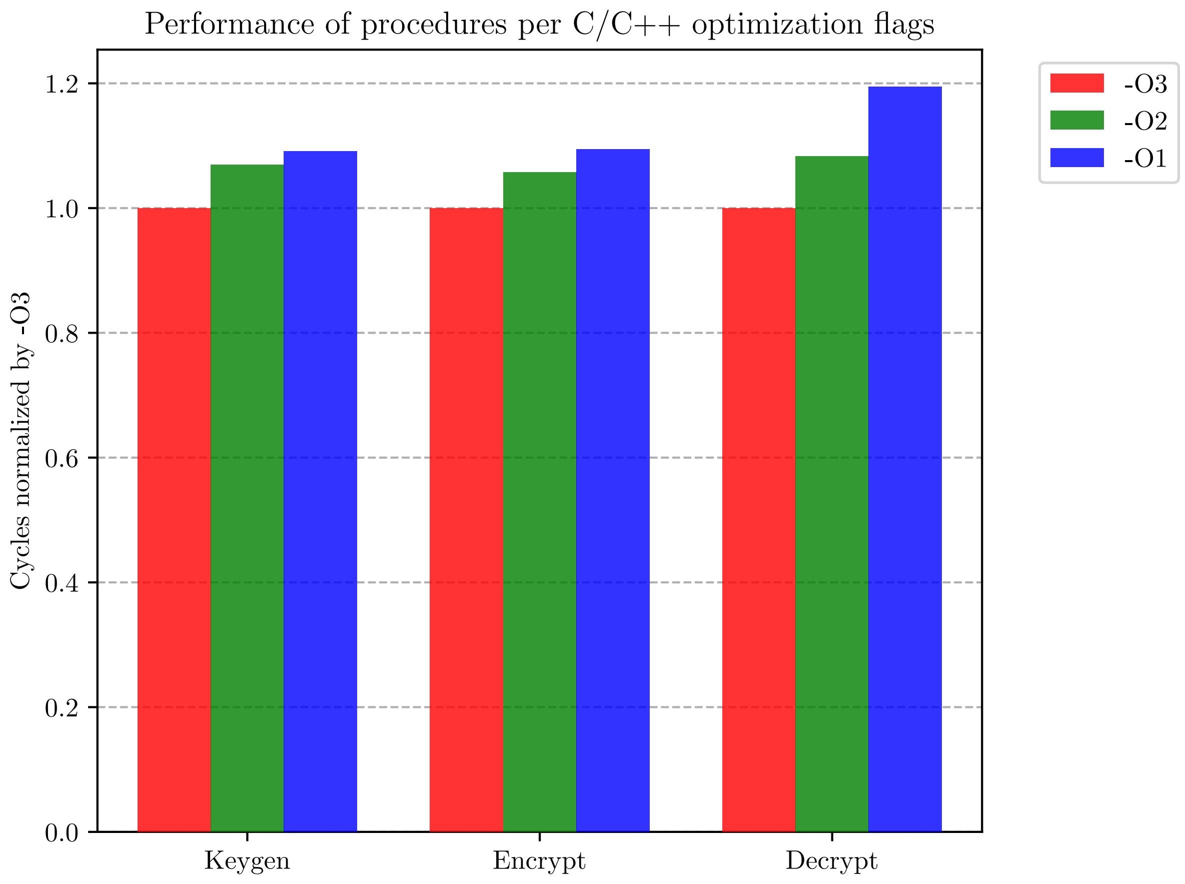

Figure 5.27: Performance measurements for each optimization flag for RQC-128.

The core count has a performance drop of around 0.3%-0.4% when using a single core compared to two and four cores. The difference between two and four cores are within margin of error. Memory and CPU clock speed follows the overall data within margin of error, where the performance from CPU clock speed follows the regression line.

Profiling

	Keygen	Encryption	Decryption
`kar_mul`	23.50%	21.55%	3.62%
`ffi_elt_reduce`	4.70%	10.78%	15.38%
`ffi_elt_add`	14.53%	12.50%	4.07%
`ffi_elt_inv`	0.43%	1.72%	11.76%

The old RQC implementation have some similar functions to the old ROLLO-II, but a new one is `kar_mul`. The new function is a major time consumer in keygeneration and

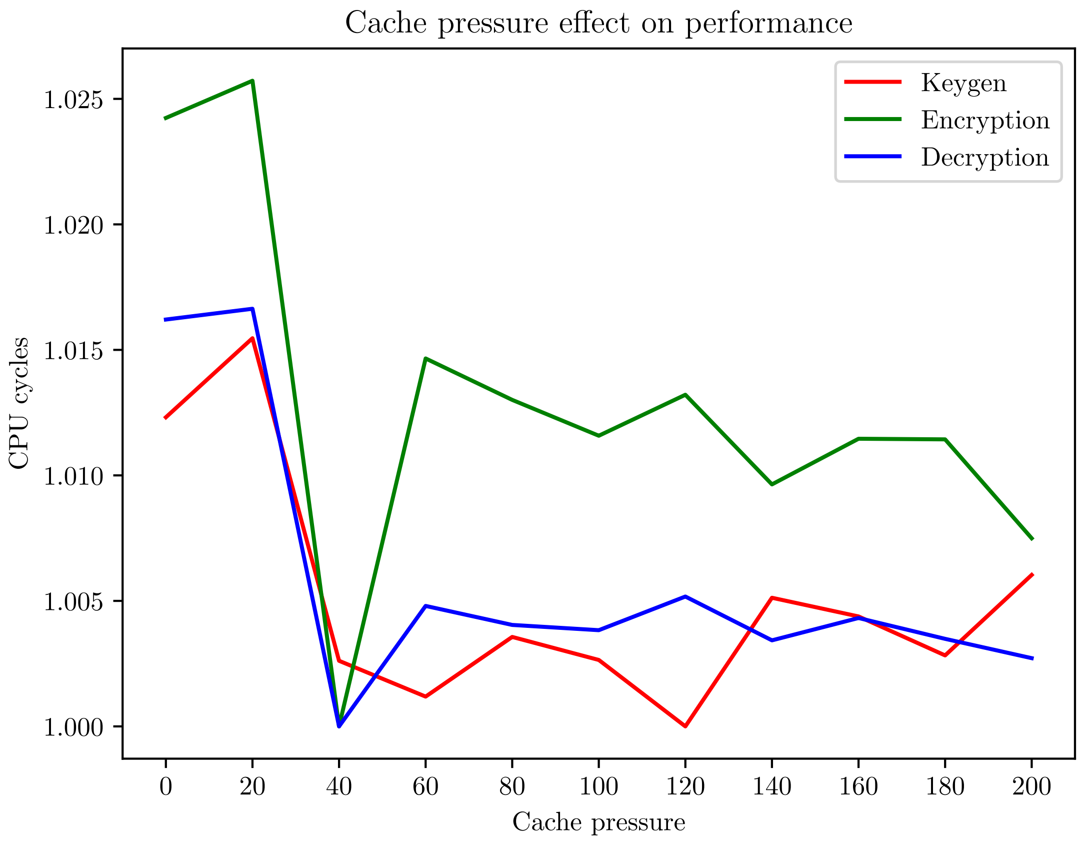

Figure 5.28: Performance measurements based on cache pressure for `-O3` RQC-128.

encryption procedures. The function consists of multiple calls to other functions and itself recursively, but also variable sized for-loops. The functions called are often functions using AVX or other instruction to optimize binary operations. The function is only called once per keygeneration and twice per encryption. With the recursive calls in keygeneration, we find the function to be called around 556 times per keygeneration. A recursive function is often considered to be possible to parallelize. However it is often not a good idea as creating new threads often come with an overhead, and in the real world, creating 556 threads would probably affect the procedure in a negative way. Parallelization is however one of the strong suit of FPGA, where the function could be implemented in a software/hardware solution. The share of the total time consumption reduces when the security level is increased.

`ffi_elt_reduce` has the same goal as it had in ROLLO-II, where the goal is to reduce a finite field element based on an irreduceble element f. In the case of RQC the f element is different for each security level, and the function is therefore different since the element is hardcoded in the function for an optimized solution. The share of the total time consumption increases for the function when the security level is increased.

The goal of `ffi_elt_add` is to compute the addition of two finite field elements.

This is done by using AVX instruction like in ROLLO-II. The percentage of total time consumption for this function reduces a small amount when the security level is increased.

The function `ffi_elt_inv` is a function that was not among the major time consumers in ROLLO-II. It computes the inverse of a finite field element by using an extended euclidean based algorithm. The function uses hardcoded finite field elements to compute the result efficiently. The percentage of time consumption reduces for this function when security level is increased.

5.6.2 New implementation

	keygen	encrypted	decrypted
clock speed	-0.8827	-0.8661	-0.8904
memory speed	0.001359	0.002133	0.001801
cache pressure	-0.002546	-0.002505	-0.002228
cores	-0.002808	-0.003105	-0.002848
C/C++ flags	0.1407	0.1596	0.1362

Figure 5.29: The correlation matrix for newRQC-128

The updated submission of RQC came with a new implementation of the system using new parameters. This implementation uses functions and libraries similar to the new implementation of ROLLO-II. Therefore I would expect many similar correlation to this system. However looking at figure 5.29 we see that the performance has higher correlation with C/C++ optimization flags. Looking at figure 5.30 the performance difference is above 20% between `-O1` and `-O3` for all procedures. This difference increases with the security level, and for level 5 the `-O2` flag is slower than `-O1` for keygeneration. This is most likely caused by the major time consuming functions are different for the security levels.

The cache pressure affect on performance is more volatile than the overall findings, especially for encryption. At security level 1 the cache pressure affect seems to be close to normal affect on the system, where a cache pressure below 40 means performance is decreased by up to 0.6%. For security level 3 and 5 the keygeneration and decryption follows a similar model, however as you can see in figure 5.31 the performance difference of encrypion with respect to cache pressure is up to 1.75%. This could be because the whole procedure does not fit in the L2 or L3 cache of the processor, and has to rely on the RAM memory which is significantly slower. The affect only with `-O3` flag is up to 4% for encryption with security level 3 and 5. Security level 1 follows the model from overall data.

The other parameters have close to the same affect as you find for the overall results. Running on a single core the performance decreases by up to 0.4% for security level 1 and

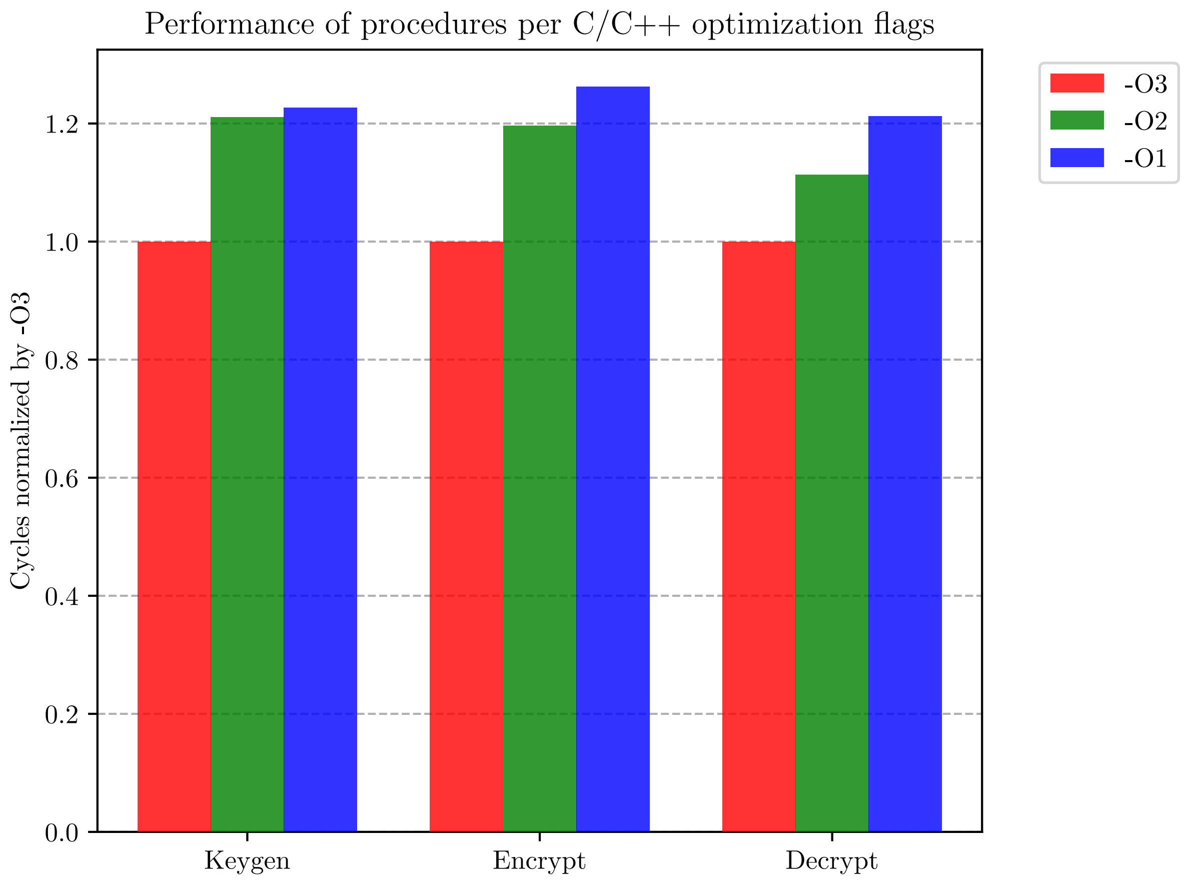

Figure 5.30: Performance measurements for each optimization flag for newRQC-128.

3, while up to 1.0% for security level 5.

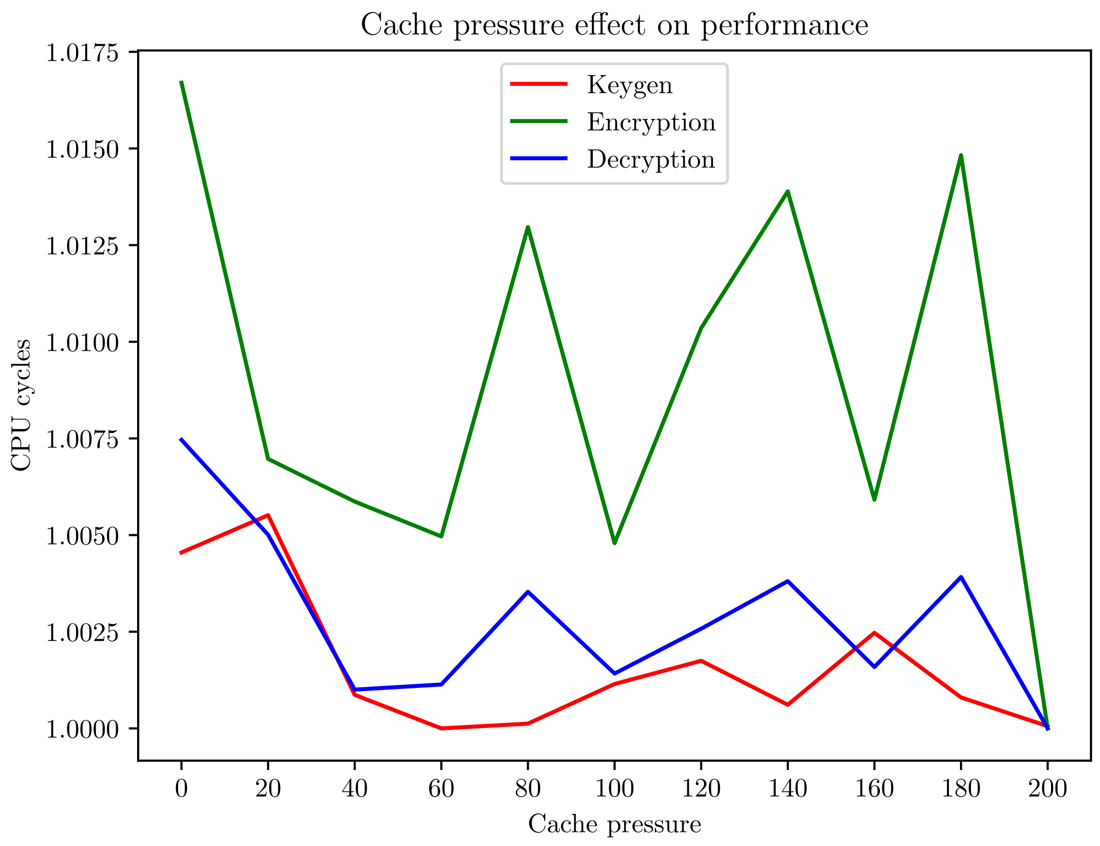

Figure 5.31: Performance measurements based on cache pressure for newRQC-192.

Profiling

	Keygen	Encryption	Decryption
`rbc_kar_mul`	18.02%	23.18%	1.61%
`rbc_elt_reduce`	4.95%	4.72%	14.06%
`rbc_elt_add`	15.77%	11.37%	6.68%
`rbc_elt_mul`	4.50%	8.58%	12.90%
`rbc_elt_ur_mul`	7.21%	7.51%	6.22%

The functions are many of the same as in updated ROLLO-II but with different irreducible polynomials for the finite fields. `rbc_kar_mul` is very similar to the corresponding function in the old RQC implementation called `kar_mul`, and there has not been much more optimization of this function. The percentage of time consumption for `rbc_kar_mul` reduces when the security level is increased. The `rbc_elt_reduce` functions goal is to reduce a finite field element. The operations are binary and it is very similar to the corresponding function in the old implementation, but with some differences as the irreducible polynomial is different. The function is called around 42,000 times per decryption and therefore small optimizations can increase the performance substantially. The binary operations can be implemented in an FPGA software/hardware solution to increase performance. The percentage of time consumption of `rbc_elt_reduce` reduces when the security level is increased.

`rbc_elt_add` is identical to the corresponding function you find in the new implementation of ROLLO-II. The function is called almost 16,000 times per keygeneration, 30,000 times per encryption and 64,000 times per decryption. The function is small and therefore easily inlined. There are however 44 different places where this function is called from, resulting in an increase in compiled code size and may not fit in cache or memory. The function uses SSE and AVX instructions for doing the binary operations. The percentage of the time consumption by this function reduces when security level is increased.

`rbc_elt_mul` is also identical to the corresponding function in new implementation of ROLLO-II. The function consists of a variable initialization and three function calls. The reason why this function is a major time consumer is not clear, but a reason can be overhead of calling functions, combined with the fact that the function is called almost 30,000 times per decryption. One of the function called by `rbc_elt_mul` is `rbc_elt_ur_mul`. This function increases in percentage of time consumption when the security level is increased, while `rbc_elt_mul` reduces. The `rbc_elt_ur_mul` function consists of SSE and AVX instructions for binary operations to optimize the function.

5.6.3 Comparison

Because the old implementation uses broken parameters it is not recommended to be used, even though the parameters claiming security level 5 is still strong enough to claim security level 1. The new implementation comes with many of the same functions with little to no new optimizations. However the move to make RBC its own library (function with ffi prefix in old implementation) means it is easier for a community to optimize the library for different devices and still work for all security levels and other cryptosystems even.

	Keygen	Encryption	Decryption
RQC-128	62%	54%	133%
RQC-192	58%	62%	113%
RQC-256	42%	44%	102%

As you can see the performance of RQC has decreased after the updated submission. This performance drop is most likely due to parameter changes. It is hard to say if any optimization done by the updated submission have any effect on the results because of the parameter changes.

5.7 Bottlenecks

The most obvious bottleneck of the system is the CPU clock speed. The performance had almost a 1 to 1 correspondence with the CPU clock speed. This happened for all cryptosystems with some erroneous results. If this bottleneck is to be reduced to optimize the solutions, it is possible to use external hardware devices to offload the CPU of the most intense calculations and operations. Such a device can be an FPGA in a software/hardware solution. Another solution is to create new instruction sets in the CPU specially made for a standardized post-quantum cryptosystem.

The reliance on memory seem to not affect the system in any noticeable way, or the results from the different memory speeds are not correct. I did not find a clear reason for this interesting result.

The cache pressure affect on the system seem to be in close link to the C/C++ optimization flag. The value did not have much impact on different cryptosystem when `-O1` or `-O2` flags were used. Somehow the performance with `-O3` flag were affected negatively when the cache pressure was low. This could be interesting to look further into. The idea of using cache pressure first came from trying to limit CPU cache or other types of fast memory that the code could be optimized by. When I did not find a direct way of limiting CPU cache, I found changing the cache pressure to be the next best solution.

The core count does not seem to have much affect on the system, which means most likely all cryptosystem tested are bottlenecked by a single core and do not take advantage of parallelization. It may be that the systems is not so easily parallelized. As modern CPU seem to come with higher and higher core count, it would be a good idea to optimize for this in the future. Libraries available for use like the NTL do have the possibility to parallelize some of its functions.

The C/C++ optimization flags did have a variable affect on the performance from cryptosystem to cryptosystem. I don't think it is correct to say a system is bottlenecked by a optimization flag, but it does say something about how the system has been implemented. If there is not much performance increase by increasing the optimization flag, the likelihood of finding manual optimization of the code is higher. While a significant performance increase by the optimization flags could indicate there has not been done much manual optimization of the code. This could however just mean the developers know and depend on the optimizations by the compiler. A manually optimized code is usually not a good idea unless you know what you do. However I do believe most of the developers of the candidate cryptosystems know what they are doing. A problem with the optimizations of the candidate implementations is that they are designed to run as a standalone solution, not in a program with multiple other cryptosystem and maybe other processes running at the same time.

5.8 Is it possible to find a ”one fits all” cryptosystem?

The idea of finding a ”one fits all” cryptosystem is that a single candidate in the NIST PQC project can replace the standardized PKEs and KEMs found today. Devices used today may have many different constrained resources, e.g. compute power, storage and internet bandwidth. Some use cases may prefer efficient encryption and small public key size over other aspects of a cryptosystem. While intending of using a PKE as a KEM it might find that most aspects of the cryptosystem is important. It is for this reason NIST has no plan of choosing a single winner, but rather multiple. The choice is also based on having diversity in the different mathematical problems the systems are based on. If one is broken, there are still fallback solutions without the need to standardize a new system all over again.

5.9 Performance in real world applications

The cryptosystems are made for having the best efficiency and performance when they are running alone with no other programs interrupting them. In the real world public key cryptography is usually only used once per communication partner. This means the submitted implementations might not be fully optimized for a real world usage. Unfortunately I did not have time to compare the result in OQS library with the results I found from the submissions. The OQS library is meant for real world usage and would be interesting to see if there are any performance differences. This may be something that can be looked into further. When candidates are standardized in Public Key Infastructure, there might be a need for optimizing the solution differently. Some of the candidates have many different implementation of the scheme optimized for different hardware. I decided to only look at the solution that is called the optimized implementation.

LAC seems like a good solution for IoT devices, as the solution is relatively light weight with efficient keygeneration, encryption and decryption. However when trying to use the implementation with a memory-limited device, the optimization flag often used is `-O2`. The results with this flag shows a performance decrease of up to 250%, which is probably unacceptable on small devices with limited processing power.

HQC, RQC and ROLLO-II seems to be good solutions for general Public Key Infrastructure tasks. With small benefits over one another, some have faster keygeneration speed, other have deterministic decryption algorithm.

The LEDAcrypt is a special candidate. It has the fastest encryption and decryption based on the number of bytes it can encrypt, including the fact that it can take a variable sized message resulting in a variable sized ciphertext. This gives the cryptosystem only a small niche of use cases, as a solution of combining one of the other candidate with AES quickly becomes faster. The keygeneration is also significantly slower than the other candidates and the compiled code size is also significantly larger. A single keygeneration for security level 5 took up to five seconds.

Chapter 6

Conclusion and afterthoughts

With the results presented and discussed, it is time to try and answer the research questions:

- *"What is considered bottlenecks in a computer when looking at performance measurements of post-quantum cryptographic algorithms?"*
  It seems the main component bottlenecking the performance of PQC algorithms is as expected the CPU of a computer. The bottleneck of the CPU can be best measured by the correlation of the CPU clock speed with performance of the algorithms. There are also other components that limit the performance of the CPU. These are CPU cache and compiler optimization that we see in the results are affecting the performance. It is logical to think that the memory speed is bottlenecking perfomance to a degree, but I did not find anything in the result to back this up. The candidates are also bottlenecked to a single core, which means CPUs with higher core count can not take advantage of this.

- *"What is the performance of candidates in the NIST Post-Quantum Cryptography project and how do they compare to each other when resources are constrained?"*
  The different public key cryptosystems compared have different advantages and disadvantages. This can be seen in the chapter 5 with different result giving information about what differences the system has to each other.

- *"What kind of performance increase or decrease does updated versions of the cryptographic algorithm candidates have compared to the original versions in round 2 of the NIST Post-Quantum Cryptography project?"* The comparison of the old and updated submission of each candidate that had an updated submission while writing this thesis is compared in chapter 5. When the parameters to candidates are broken this usually leads to increased key sizes and longer runtime for keygeneration, encryption and decryption. When new and better parameters are found it is expected that the runtime would decrease for all three procedure, but for HQC the performance increased for decryption but decreased for keygeneration and encryption even though the parameters and key sizes are reduced. I did not have time to look at the updated implementation of LAC and LEDAcrypt.

- *"What does the resources and components in a computer that affect the performance of post-quantum cryptographic algorithms say about the optimization of these algorithms?"*
  In the results I found, I did not see any direct relation between the components bottlenecking the system with the optimization of it. There are however different correlation between performance of candidates and components. This might be an indication of either optimizations or different operation and functions used in each candidate.

It's hard to find a conclusion that fit for all candidate algorithms I have looked at. I therefore recommend to look in chapter 5 if there is a specific candidate you are looking for information about.

It is definitely possible to do a deeper analysis of the performances for the candidates. However I did not have time to do any more analysis. This might have been fixed by focusing on fewer candidates. This would however make me loose the overall result, which means it would be harded to know when a candidate stands out with some interesting results.

An afterthought is that I maybe should have focused on the KEM versions of the schemes, as the PKE implementation was often missing and had to find the correct functions to use for PKE to work. Doing it this way did benefit my understanding of each candidate, as I had to make sure I understood how each PKE worked. Another afterthought is that the merger of all candidates into a single performance measurement program was very time consuming. This is because of all the prefixing of function for each security level and candidate. It was a lot of repetitive work and I would have saved a lot of time running each candidate separate, however I believe the result gives a more real world performance measurements of the implementations over running them separate.

6.1 Future works

Future works may be:

- Looking closer at how hybrid solution can be used safely in the time until a standardized solution is found.
- A performance and/or security analysis of each NIST PQC candidate.
- A comparison with the rest of the candidates found in the NIST PQC project.
- A deeper analysis of the correlation of performance between the different parameters, like the `-O3` flag with cache pressure.

Bibliography

[1] Dong Pyo Chi et al. 'Lattice Based Cryptography for Beginners'. In: *IACR Cryptol. ePrint Arch.* 2015 (2015), p. 938. URL: https://eprint.iacr.org/2015/938.pdf (visited on 01/01/2020).

[2] Nicolas Sendrier. 'Code-based Cryptography - PQCRYPTO Summer School on Post-Quantum Cryptography 2017'. In: (2017). URL: https://2017.pqcrypto.org/school/slides/cbctutorial.pdf (visited on 01/02/2020).

[3] E. Berlekamp, R. McEliece and H. van Tilborg. 'On the inherent intractability of certain coding problems (Corresp.)' In: *IEEE Transactions on Information Theory* 24.3 (1978), pp. 384–386.

[4] 'Post-Quantum Cryptography PQC - Security (Evaluation Criteria)'. In: (2017). URL: https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/evaluation-criteria/security-(evaluation-criteria) (visited on 01/02/2020).

[5] Elena Andreeva, Andrey Bogdanov and Bart Mennink. *Towards Understanding the Known-Key Security of Block Ciphers.* Cryptology ePrint Archive, Report 2015/222. 2015. URL: https://eprint.iacr.org/2015/222 (visited on 01/04/2020).

[6] Kevin Kimball. *Announcing Request for Nominations for Public-Key Post-Quantum Cryptographic Algorithms.* 2016. URL: https://federalregister.gov/a/2016-30615 (visited on 01/02/2020).

[7] Gorjan Alagic et al. 'Status Report on the First Round of the NIST Post-Quantum Cryptography Standardization Process'. In: (2019). URL: https://doi.org/10.6028/NIST.IR.8240 (visited on 01/02/2020).

[8] Gorjan Alagic et al. *Status Report on the First Round of the NIST Post-Quantum Cryptography Standardization Process.* Jan. 2019. URL: https://doi.org/10.6028/NIST.IR.8240 (visited on 01/02/2020).

[9] Dennis Hofheinz, Kathrin Hövelmanns and Eike Kiltz. *A Modular Analysis of the Fujisaki-Okamoto Transformation.* Sept. 2017. URL: https://eprint.iacr.org/2017/604.pdf (visited on 01/03/2020).

[10] John Martinis. *Quantum Supremacy Using a Programmable Superconducting Processor.* 2019. URL: https://ai.googleblog.com/2019/10/quantum-supremacy-using-programmable.html (visited on 01/03/2020).

[11] Thomas Eisenbarth et al. 'MicroEliece: McEliece for Embedded Devices'. In: *Cryptographic Hardware and Embedded Systems - CHES 2009*. Ed. by Christophe Clavier and Kris Gaj. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 49–64. ISBN: 978-3-642-04138-9.

[12] Qasem Abu Al-Haija et al. 'Efficient FPGA Implementation of RSA Coprocessor Using Scalable Modules'. In: *Procedia Computer Science* 34 (2014). The 9th International Conference on Future Networks and Communications (FNC'14)/The 11th International Conference on Mobile Systems and Pervasive Computing (MobiSPC'14)/Affiliated Workshops, pp. 647–654. ISSN: 1877-0509. DOI: `https://doi.org/10.1016/j.procs.2014.07.092`. URL: `http://www.sciencedirect.com/science/article/pii/S1877050914009478` (visited on 31/05/2020).

[13] Farnoud Farahmand et al. 'Implementing and Benchmarking Seven Round 2 Lattice-Based Key Encapsulation Mechanisms Using a Software/Hardware Codesign Approach'. In: (2019). URL: `https://csrc.nist.gov/CSRC/media/Events/Second-PQC-Standardization-Conference/documents/accepted-papers/gaj-implementing-benchmarking.pdf` (visited on 29/02/2020).

[14] Micheal E. Lee. *Optimization of Computer Programs in C*. 1999. URL: `http://icps.u-strasbg.fr/~bastoul/local_copies/lee.html` (visited on 01/03/2020).

[15] Jacco Bikker. 'Practical SIMD Programming'. In: (2017). URL: `http://www.cs.uu.nl/docs/vakken/magr/2017-2018/files/SIMD%5C%20Tutorial.pdf` (visited on 31/05/2020).

[16] Jiafeng Xie, P.K. Meher and Zhi-Hong Mao. 'High-Throughput Finite Field Multipliers Using Redundant Basis for FPGA and ASIC Implementations'. In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 62 (Jan. 2015), pp. 110–119. DOI: `10.1109/TCSI.2014.2349577`. (Visited on 01/03/2020).

[17] P. W. Shor and J. Preskill. *Simple Proof of Security of the BB84 Quantum Key Distribution Protocol*. May 2000. URL: `https://arxiv.org/abs/quant-ph/0003004` (visited on 29/02/2020).

[18] Carlos Aguilar Melchor et al. 'ROLLO - Rank-Ouroboros, LAKE & LOCKER'. In: 1.2 (Aug. 2019), pp. 9–11. URL: `https://pqc-rollo.org/doc/rollo-specification_2019-08-24.pdf` (visited on 01/02/2020).

[19] Philippe Gaborit and Gilles Zémor. 'On the hardness of the decoding and the minimum distance problems for rank codes'. In: 62.12 (Dec. 2016), pp. 7245–7252.

[20] Magali Bardet et al. 'Algebraic attacks for solving the Rank Decoding and MinRank problems without Gröbner basis'. In: *arXiv e-prints*, arXiv:2002.08322 (Feb. 2020), arXiv:2002.08322. arXiv: `2002.08322 [cs.CR]`. (Visited on 30/04/2020).

[21] Magali Bardet et al. 'An Algebraic Attack on Rank Metric Code-Based Cryptosystems'. In: *arXiv e-prints*, arXiv:1910.00810 (Oct. 2019), arXiv:1910.00810. arXiv: `1910.00810 [cs.CR]`. (Visited on 30/04/2020).

[22] *Magma Software*. Apr. 2020. URL: http://magma.maths.usyd.edu.au/magma/handbook/text/193 (visited on 30/04/2020).

[23] Carlos Aguilar Melchor et al. 'ROLLO - Rank-Ouroboros, LAKE & LOCKER'. In: (Apr. 2020). URL: https://pqc-rollo.org/doc/rollo-specification_2020-04-21.pdf (visited on 30/04/2020).

[24] Carlos Aguilar Melchor et al. 'Intellectual property'. In: (Aug. 2019). URL: https://csrc.nist.gov/CSRC/media/Projects/post-quantum-cryptography/documents/round-2/updated-ip-statements/Rollo-Statements-Round2.pdf (visited on 01/03/2020).

[25] Carlos Aguilar Melchor et al. 'Rank Quasi-Cyclic (RQC)'. In: 1.1 (Aug. 2019), pp. 10–11. URL: https://pqc-hqc.org/doc/hqc-specification_2019-08-24.pdf (visited on 01/02/2020).

[26] Carlos Aguilar Melchor et al. 'Rank Quasi-Cyclic (RQC)'. In: 1.1 (Apr. 2020), pp. 10–11. URL: https://pqc-rqc.org/doc/rqc-specification_2020-04-21.pdf (visited on 30/04/2020).

[27] Carlos Aguilar Melchor et al. 'Intellectual property'. In: (Nov. 2017). URL: https://csrc.nist.gov/CSRC/media/Projects/post-quantum-cryptography/documents/round-2/updated-ip-statements/RQC-Statements-Round2.pdf (visited on 01/03/2020).

[28] Carlos Aguilar Melchor et al. 'Hamming Quasi-Cyclic (HQC)'. In: 1.1 (Aug. 2019), pp. 10–11. URL: https://pqc-rqc.org/doc/rqc-specification_2019-08-24.pdf (visited on 01/02/2020).

[29] Carlos Aguilar Melchor et al. 'Intellectual property'. In: (Nov. 2017). URL: https://csrc.nist.gov/CSRC/media/Projects/post-quantum-cryptography/documents/round-2/updated-ip-statements/HQC-Statements-Round2.pdf (visited on 01/03/2020).

[30] Xianhui Lu. *Update of LAC*. Apr. 2020. URL: https://groups.google.com/a/list.nist.gov/forum/%5C#!topic/pqc-forum/QIrRyOG3-Pk (visited on 01/05/2020).

[31] Yamin Liu et al. 'LAC'. In: 1 (2019), p. 1. URL: https://cs.rit.edu/~ats9095/csci762/pdfs/LAC.pdf (visited on 01/02/2020).

[32] Qian Guo, Thomas Johansson and Jing Yang. *A Novel CCA Attack using Decryption Errors against LAC*. Cryptology ePrint Archive, Report 2019/1308. 2019. URL: https://eprint.iacr.org/2019/1308 (visited on 01/04/2020).

[33] Yamin Liu et al. 'LAC'. In: 1 (Apr. 2020), p. 1. URL: https://eprint.iacr.org/2018/1009.pdf (visited on 01/05/2020).

[34] Marco Baldi et al. 'LEDAcrypt: Low-dEnsity parity-check coDe-bAsed cryptographic systems'. In: 1 (Mar. 2019), p. 10. (Visited on 01/02/2020).

[35] Marco Baldi et al. 'LEDAcrypt: Low-dEnsity parity-check coDe-bAsed cryptographic systems'. In: (Mar. 2020). URL: https://www.ledacrypt.org/documents/LEDAcrypt_spec_2_5.pdf (visited on 01/05/2020).

[36] Daniel Apon et al. 'Cryptanalysis of LEDAcrypt'. In: (Apr. 2020). URL: https://eprint.iacr.org/2020/455.pdf (visited on 01/05/2020).

[37] *PQC - API notes*. Mar. 2017. URL: https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/archive/api-march2017.pdf (visited on 01/02/2020).

Appendix

Cryptosystem	Parameters	Claimed complexity	[20] complexity
ROLLO-I-128	(79, 94, 47, 5)	128	70.2
ROLLO-I-192	(89, 106, 53, 6)	192	86.2
ROLLO-I-256	(113, 134, 67, 7)	256	158.1
ROLLO-II-128	(83, 298, 149, 5)	128	93.0
ROLLO-II-192	(107, 302, 151, 6)	192	110.5
ROLLO-II-256	(127, 314, 157, 7)	256	169.8
ROLLO-III-128	(101, 94, 47, 5)	128	69.5
ROLLO-III-192	(107, 118, 59, 6)	192	88.0
ROLLO-III-256	(131, 134, 67, 7)	256	137.7
RQC-I	(97, 134, 67, 5)	128	76.6
RQC-II	(107, 202, 101, 6)	192	100.9
RQC-III	(137, 262, 131, 7)	256	143.8

Figure 6.1: Claimed security versus the complexity found in [20]