Hardware Implementations of the McEliece Cryptosystem for Post Quantum Cryptography

Petter Nyland Røneid

Thesis submitted for the degree of
Master in Informatics: Programming and System Architecture
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2021

Hardware Implementations of the McEliece Cryptosystem for Post Quantum Cryptography

Petter Nyland Røneid

Hardware Implementations of the McEliece Cryptosystem for Post Quantum Cryptography

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

The ever-evolving threat landscape of cybersecurity has shown us that no bit of information is secure once enough time has elapsed. New methods of attack, analysis, and advancements in technology regularly break old cryptographic methods.

Advancements in quantum computing are now becoming a concern for the security of our asymmetric cryptography. Quantum algorithms will drastically reduce the complexity of solving the underlying security primitives and have created the need for quantum-resistant cryptosystems.

The NIST PQC is an ongoing process of selecting a quantum-resistant standard for asymmetric cryptography. One of the candidates to the NIST PQC is Classic McEliece, a system based on the now 40-year-old code-based McEliece. Despite its age, the McEliece cryptosystem with binary Goppa code is still regarded as secure and quantum-resistant with proper parameter choices.

This thesis outlines how linear codes can be used to construct cryptographic schemes. It performs a performance comparison of the software and hardware implementations of the code-based candidates Classic McEliece, BIKE, and HQC.

Finally, the hardware implementation of Classic McEliece was examined in the search for alternative designs and improvements. Some potential improvements were found, and some alternate designs performed poorly, thus reaffirming the choices of the Classic McEliece team.

Acknowledgements

I want to thank my supervisor Thomas Gregersen. Our regular meetings early on helped me select a direction. I am very grateful for the opportunity to explore this topic, and I appreciated the high degree of freedom and independence I was given.

I would also like to thank my co-supervisors, Audun Jøsang and Åvald Åslaugson for being available when I needed help.

The current situation has been challenging for everyone, so I thank my friends and family for attempting to keep me sane.

Lastly, I would like to thank my professor of cryptography Leif Nilsen. Without taking the introductory course TEK4500, I would never have chosen this path. The course was a pivotal period in my life, and I wish Leif a happy retirement.

Contents

Glossary

AES advanced encryption standard. 13

AT anti tampering. 45, 46

BIKE bit flipping key encapsulation. 55

BM Berlekamp-Massey. 31

CM Classic McEliece. 8, 9, 12–14, 35, 49–53, 67, 68, 71, 72, 78, 80

CMHW Classic McEliece Hardware Implementation. 9, 53, 54

coset leader The vector of smallest hamming weight in a coset of vectors. 28

DES data encryption standard. 13

DFR decoding failure rate. 29, 56, 61

DPA differential power analysis. 73

ELP error locator polynomial. 43

FPGA field programmable gate array. 36–39

hamming weight The number of non-zero entries in a binary vector. 33

HDL hardware description language. 14, 37, 38

HLS high level synthesis. 37, 38, 68

HQC Hamming Quasi-cyclic. 9, 35, 60, 61

IND-CCA Indistinguishability under Chosen Ciphertext Attack. 2, 8, 21, 45

IND-CPA Indistinguishability under Chosen Plaintext Attack. 8, 20, 22

ISD Information Set Decoding. 41, 42, 52

KEM Key Encapsulation Mechanism. 22, 45

LUT lookup table. 36, 38

NIST National Institute for Standards and Technology. 13

OW-CPA One-Wayness under Chosen Plaintext Attack. 20, 22

$\mathbf{P_k}$ Public key. 16, 33

perfect code A code in which the distance between all codewords is the same, and decoding spheres around codewords cover the entire vector space.. 28

PKE Public Key Encryption scheme. 2, 8, 12, 16, 18, 19, 21, 22, 33, 49

PKI Public Key Infrastructure. 19

pl programmable logic. 36

PQC Post Quantum Cryptography. 13, 55

QC-MDPC Quasi-cyclic Moderate Density Parity Check. 9, 35, 55

$\mathbf{S_k}$ Secret/Private key. 33

TEMPEST Telecommunications Electronics Material Protected from Emanating Spurious Transmissions. 44

List of Figures

List of Tables

Part I

Introduction and theory

Chapter 1

Introduction

Quantum computers are coming, and with them, they bring potentially catastrophic effects on modern cryptography. If quantum computers become sufficiently powerful, the encryption techniques we use today to ensure secure communication over the internet could be broken entirely overnight, thereby compromising the confidentiality of business processes in the financial, military, commercial, government, and private sectors.

Although the quantum threat is not a certainty, [1, p. 2], the consequences of a quantum computer are too devastating to ignore.

There are two primary categories of cryptography; symmetric (Section 2.2) and asymmetric (Section 2.3). Asymmetric is the category that is used for internet communication, and the security of most asymmetric schemes rely on the difficulty of solving the factoring, and the discrete logarithm problem. With today's technology, these problems are considered to be hard (NP complexity), but if quantum computers become sufficiently powerful, they will be able to solve these problems in polynomial time. This reduction in complexity from non-polynomial to polynomial time, will render the most widely used asymmetric schemes completely broken.

This creates a considerable incentive to find replacements algorithms that are quantum resistant. NIST [2] has an ongoing selection process, the NIST PQC, that aims to select the new standard for asymmetric cryptography. Among the candidates for this selection process we find Classic McEliece (CM) [3].

CM is based on the McEliece PKE introduced by Robert McEliece in 1978 [4]. There are no known attacks that break the security of McEliece in a significant way. Because of this, McEliece is especially interesting in the NIST PQC, as it is regarded as the most tested and mature candidate to PQC.

McEliece has, however, a significant drawback: its key size. To achieve classical 128-bit security in the post-quantum world, the public key is about 1 MB. This is a far cry from our current PKEs, where for instance, an RSA key providing 128-bit security will be 3074 bit.

CM is not, however, the only McEliece based candidate in the NIST PQC. Other proposals offer smaller key sizes, but their security is still under scrutiny because they are based on newer and less tested codes. The proposals all have strengths and weaknesses, and it is not very obvious which is superior.

In addition to the numerous code-based proposals, they all have accompanying hardware implementations that can improve execution time or hardware

cost. To thoroughly compare the proposals, these will also have to be included in the analysis.

1.1 NIST PQC

National Institute for Standards and Technology (NIST) is a US governmental agency with a long-standing history in developing and maintaining standards regarding cybersecurity. Data encryption standard (DES) was announced as the standard for data encryption in 1976. When it many years later was assumed to be insecure, NIST began a selection process to find a new standard; the advanced encryption standard (AES). The winner of this selection process was Rijndael [5], and it is not known as the AES. AES is now the defacto symmetrical encryption algorithm.

It is a similar process we find ourselves in now. The quantum threat to modern asymmetric cryptographic standards has created a need for replacements. This is where the NIST Post Quantum Cryptography (PQC) comes in. NIST PQC is a selection process for post asymmetric cryptography, with separate categories for key exchange and digital signatures.

There are four security primitives that are believed to be quantum resistant, and all the submissions are based on one of these:

1. Lattices
2. **Linear Codes**
3. Isogenies
4. Multivariate-quadratic

It is in this context that this work is done. The key exchange implementations that are based on linear codes will be examined to achieve insight into their construction and performance.

1.2 Research Questions

This work will attempt to answer the following questions that arise when developing code-based cryptographic implementations.

1.2.1 Performance Comparison

Though the structure of the three code based candidates for the NIST PQC are McEliece based, they depend on the security of different linear codes. The binary Goppa codes of Classic McEliece [3] enjoy a maturity that the cyclic codes of BIKE [6] and HQC [7] do not. If the cyclic codes maintain their security in the future, will the performance of these three systems have a clear winner?

1.2.2 Hardware Description Language Generation

The hardware implementation of CM generates some of its modules from SageMath scripts. The scripts are quite straightforward, and in most cases, they will generate the logic in terms of elementary 'xor', 'not', and 'and' gates. This is not an unusual practice, but it poses some interesting questions. It is easy to assume that generating an ideal logic like this will create a minimal circuit, but that is not necessarily true when synthesizing designs on an FPGA. The

synthesized designs are complex interactions between lookup tables and not elementary gates.

It is advantageous to have a solution written in pure HDL, and because we will have a generated module and a written module, we can directly compare the footprints of each. We can then get an idea if ideal logic produces the smallest hardware design.

1.2.3 Optimizations and Alternative Designs

When designing hardware solutions for cryptographic purposes, the goal is often to reduce the runtime. It is, therefore, an obvious goal to look for optimizations in either footprint or runtime in the hardware designs.

Alternative designs are could also lead to performance improvements or trade-offs. If an alternative design is found, it should be compared to the reference design and discussed.

1.3 Research Methods

To answer the questions in Section 1.2 requires a thorough study of code-based cryptography. This included studying the three code-based systems in the NIST PQC, getting familiar with abstract algebra and its application in cryptography, and studying hardware description languages.

To not fall into every pitfall, it was also emphasized to get familiar with the literature around attacking McEliece. Any cryptographic implementation must be created with the knowledge of the threats to its security.

Because Classic McEliece is the only code-based finalist in round 3, it was given special attention.

Running hardware synthesis requires some proprietary tools. A virtual machine running Ubuntu 16.04.07 Xenial was set up, as this was one of the only versions that support the entire toolchain. To conduct hardware simulation, Modelsim was used, and to perform synthesis, Vivado WebPack was used. In addition, to run the CM hardware code generation scripts SageMath 7.4 was installed. Note that the version numbers are important when they are explicitly stated.

The software implementation of classic McEliece was modified to produce keys, messages, and ciphertexts to be used in the hardware experiments. Then modules from the hardware submission were isolated to perform experiments. The code for the experiments can be found at:
https://github.uio.no/petternr/McElieceAPI

Chapter 2

Theory

2.1 Quantum Computing

Quantum computers, should they become a practical, will allow us to design new types of algorithms that cannot run on classical computers that are limited by the classical laws of physics. By leveraging quantum mechanics to do computations we will be able to solve new and old problems with incredible speeds by today's standards.

Unfortunately, two of these problems are the discrete logarithm problem and the factoring problem; the two bases for computational security of modern asymmetric cryptography.

2.1.1 Shor's Algorithm

Shor's Algorithm [8] was designed by Peter Shor in 1994, and is an algorithm for factorization designed to run on a quantum computer. In classical computers, factorization is assumed to be hard and can not be done better than non-polynomial time, but in a quantum computer, Shor's can do this in polynomial time.

Shor later revised his algorithm to also solve discrete logarithms [9].

2.1.2 Grover's Algorithm

Grover's Algorithm [10] was designed by Lov Grover in 1996. It does not reduce the complexity of any one cryptosystem like Shor's, but it reduces the complexity of all search problems.

Grover's will allow finding an element in an unstructured dataset in $\mathcal{O}(\sqrt{n})$ where n is the size of the dataset. This is a huge improvement (and also quite counter-intuitive) as we traditionally would have to search through all elements and scale with $\mathcal{O}(n)$.

This means that every single cryptographic technology, symmetric, asymmetric and hash functions will at least have the cost of a brute force attack cut by about $\mathcal{O}(\sqrt{n})$ (assuming that Grover can be implemented efficiently).

For this reason, it is common to scale the security level in bits by a factor of 2. This indicates a squaring search space, and will therefore negate the worst-case effects of Grover's.

2.1.3 The end for classical PKE's

All PKE's in use today have security based on the hardness of either the factoring problem or the discrete logarithm problem. These problems are believed to be in the classical category NP, which means that the program that solves them will have running time-scaled exponentially with the numbers that are being computed.

Shor's algorithm [8] was invented by Peter Shor in 1994 and is an algorithm for integer factorization in polynomial time, which can be generalized to solve a general discrete logarithm problem. This is terrible news for the crypto world, as it completely breaks the security of all widely used PKE's. Luckily it can not be implemented on classical computers and can only work if larger quantum computers can be built. So if quantum computers of an arbitrary qubit-length become feasible, our current asymmetric cryptography will be completely broken.

That is not to say that RSA, DH, and EC can no longer be used, but they will require absurdly large keys to maintain the current level of security. A post-quantum RSA scheme would require a one terabyte P_k, and the cost of each new encryption and decryption would be on the scale of $1 [11].

2.2 Symmetric Cryptography

Cryptography is the practice of protecting communication between two parties against an adversary which wants to read or tamper with the messages. In other words, cryptography allows us to send a message that only the intended recipient can read.

Consider Figure 2.1. Alice can send Bob a message, but everything sent over the insecure channel is visible to Oscar. To have secure communication, we have to make the information unreadable for Oscar. *Sensitive data* is indicated as red, while black indicates *non-sensitive data*.

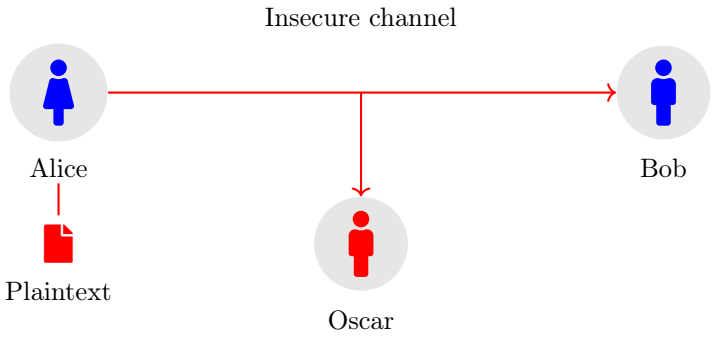

Figure 2.1: Communication over an insecure channel

Now consider Figure 2.2, with the addition of encryption and decryption, the data sent over the insecure channel is no longer sensitive. Granted that Alice and Bob have a shared key they can use for encryption and decryption. Oscar can not obtain their communication without first obtaining the shared key.

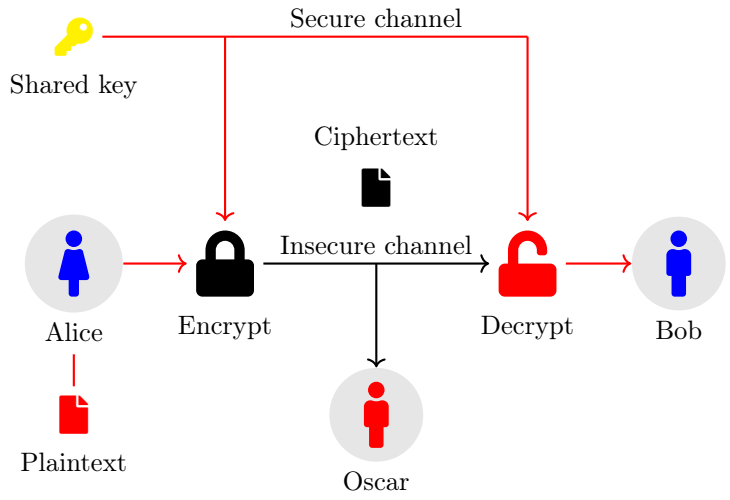

Figure 2.2: Encrypted communication over an insecure channel

Symmetric cryptography is cryptography where the communicating parties share a secret key.

This also creates a new problem, namely, how can Alice and Bob exchange a shared key in the first place? Traditionally a secure channel will mean that they will have to exchange the key physically, but this is not always practical. For n parties to communicate securely, we need $n(n-1)/2$ key pairs. This is the *key distribution problem.*

During the cold war, the United States and the Soviet Union had a communication channel based on the one-time-pad, which requires a keystream of the same length as the message. Due to the high-security risk of these keys, armed couriers had to deliver them. Now, this was difficult enough for two parties, but imagine what it would be like for the entire world. This is the problem that asymmetric cryptography helps us avoid.

So, in summary, let us define what a symmetric scheme is. To do this; we have to create some other definitions first.

Definition 1. *Spaces*
A space is the set of all possible values within the context of a cryptographic scheme, where
$\mathcal{K}$ *is the key space,*
$\mathcal{M}$ *is the message space,*
$\mathcal{C}$ *is the ciphertext space.*

Definition 2. *Encryption*
Let $\mathcal{E} : \mathcal{K} \times \mathcal{M} \to \mathcal{C}$*, encryption of a message* $m \in \mathcal{M}$ *under a key* $k \in \mathcal{K}$ *is done by:*

$$\mathcal{E}_k(m) = c$$

where the subscript k *of* $\mathcal{E}$ *implies the call* $\mathcal{E}(k, m)$ *where* k *is fixed.*

Definition 3. *Decryption*
Let c *be the encrypted ciphertext of* m *under the key* k*, and*

let $\mathcal{D} : \mathcal{K} \times \mathcal{C} \to \mathcal{M}$ *be the function that can perform decryption by:*

$$\mathcal{D}_k(c) = p$$

where the subscript k *of* $\mathcal{D}$ *implies the call* $\mathcal{D}(k, c)$ *where* k *is fixed.*

Definition 4. *Symmetric Cryptographic Scheme*
let $k \in \mathcal{K}$ *be generated by the function,*
$K_{gen} : () \to \mathcal{K}$,
$\mathcal{E} : \mathcal{K} \times \mathcal{M} \to \mathcal{C}$,
$\mathcal{D} : \mathcal{K} \times \mathcal{C} \to \mathcal{M}$
be the function that allows decryption of $\mathcal{E}$ *so that:*

$$\mathcal{D}_k(\mathcal{E}_k(p)) = p$$

A symmetric cryptographic scheme is $\Pi = (K_{gen}, \mathcal{E}, \mathcal{D})$.

Note that these definitions do not define any properties that make a cryptographic scheme secure, and if $\mathcal{E}_k$ and $\mathcal{D}_k$ where the functions that just returned the input these would be a valid cryptographic scheme according to this definition. To have secure cryptographic scheme additional properties are necessary (Section 2.5).

2.3 Asymmetric cryptography

Symmetric cryptography created a problem of key distribution: how do we distribute keys to communicating parties. *Asymmetric cryptography* allows two parties with no prior knowledge to exchange a shared secret key over an insecure channel.

Public Key Encryption scheme (PKE) is often used interchangeably with asymmetric cryptography.

So like with symmetric cryptography, let us define what a PKE is.

Definition 5. *Public Key Encryption scheme (PKE)*
Let the tuples of keys P_k *and* S_k *in the respective keyspaces* $\mathcal{K}_p, \mathcal{K}_s$,
$\mathcal{M}$ *be the message space, and* $\mathcal{C}$ *be the ciphertext space.*
$K_{gen} : () \to \mathcal{K}$,
$\mathcal{E} : \mathcal{K}_p \times \mathcal{M} \to \mathcal{C}$,
$\mathcal{D} : \mathcal{K}_s \times \mathcal{C} \to \mathcal{M}$ *so that:*

$$\mathcal{D}(S_k, \mathcal{E}(P_k, p)) = \mathcal{D}(S_k, c) = p$$

A PKE is the tuple $(K_{gen}, \mathcal{E}_{S_k}, \mathcal{D}_{S_k},)$

So unlike a symmetric scheme, a PKE C will have two keys; one for encryption (public key), and one for decryption (private or secret key). The public key will as the name implies be known to the public, but the private key is the secret part of the decryption algorithm and will be kept secret by the owner of the key. There is a unique relation between k_{priv} and k_{pub} and they can not be used interchangeably; one can not use a private key to decrypt a message that was not encrypted using the corresponding public key.

The flow of a PKE is illustrated in 2.3. Here Alice will use Bob's public key to encrypt the message, and Bob can decrypt it using his private key.

We also see that Oscar has access to Bob's public key, but he can not use this key to discern anything about what the plaintext is. The public key must be distributed using a Public Key Infrastructure (PKI). In short, a PKI is a way of distributing public keys so that their integrity is assured. If Alice requests Bob's public key, Oscar could potentially intercept and replace Bob's public key with his own. This is the problem that a PKI solves. We will not be concerned about how a PKI works, but if the reader is curious an explanation can be found in [12, p. 566].

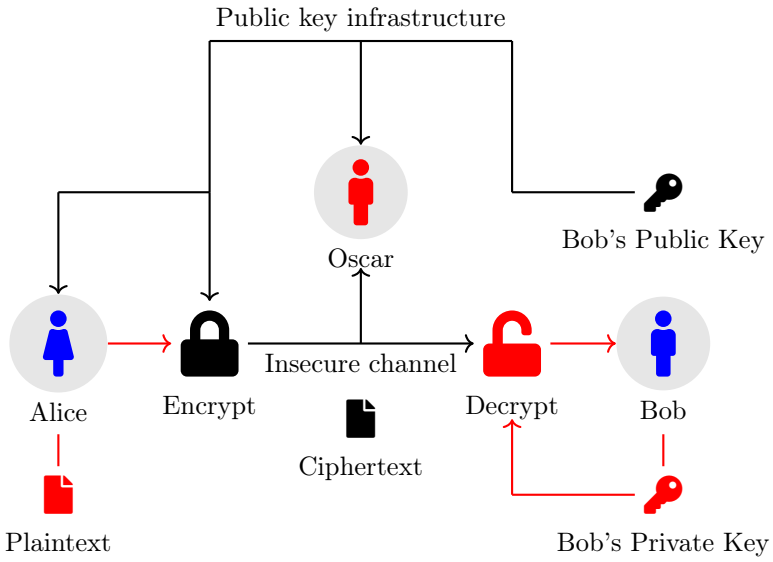

Figure 2.3: Public key encryption

2.4 128-Bit Security

A *security goal* is an abstract concept that allows us to compare the security of two heterogeneous cryptosystems. The gold standard of security is regarded to be 128-bit. This means that to break the security of a scheme, there must exist no attack better than a brute force search of a keyspace of 128-bit.

Breaking RSA, for instance, is equivalent to factoring the public key, which is a large number. Factoring a number has lower complexity than just brute-forcing the keyspace, and therefore an RSA key must be quite a bit larger than 128-bit to achieve 128-bit security (3072 bit).

Due to Grover's, the security goal for post-quantum cryptography is considered to be 256 bit.

2.5 IND-CPA

We saw from Section 2.2 that the definitions of a Cryptographic scheme does not say anything about the security of them. They could potentially completely expose the messages that they intended to encrypt, but still, meet the definition of a cryptographic scheme.

To say something about the security of a cryptographic system, we have some useful properties they can achieve that guarantees that an attacker will not be able to break the system.

One of the primary target properties is Indistinguishability under Chosen Plaintext Attack (IND-CPA), and to achieve this property, a cryptographic system must be demonstrated to win a game. The game is defined in Figure 2.4.

$\text{Exp}_{\Pi}^{\text{IND-CPA}}(A)$ describes an experiment where a challenger will allow any adversary to gain access to an encryption oracle $\mathcal{E}_k$ which will allow them to encrypt any message they want under the secret key k. In step 3, the adversary will select two messages, where one of them will be encrypted based on b. The adversary will then make a guess whether b was 0 or 1. If they are correct they win, and if they can consistently beat the game, then $\Pr[b = b']$ will be greater than $1/2$. Just by guessing an adversary will be expected to win half the time, so $\text{Adv}_{\Pi}^{\text{IND-CPA}}(A)$ should be close to 0 to guarantee that any adversary can not do better than guessing.

There is a weaker security goal called OW-CPA where the adversary will not be allowed to encrypt messages, so this property only guarantees that an adversary can not find a connection between plaintext and ciphertext.

$\text{Exp}_{\Pi}^{\text{IND-CPA}}(A)$

1. $k \xleftarrow{\$} \Pi.\text{KeyGen}$
2. $b \leftarrow \{0, 1\}$
3. $M_0, M_1 \leftarrow A^{\mathcal{E}_k(\cdot)}$
4. $C^* \leftarrow \Pi.\mathcal{E}_k(M_b)$
5. $b' \leftarrow A^{\mathcal{E}_k(\cdot)}(C^*)$
6. **return** $[b = b']$

$$\boxed{\text{Adv}_{\Pi}^{\text{IND-CPA}}(A) \overset{def}{=} |2\Pr[b = b'] - 1|}$$

Figure 2.4: Indistinguishability under Chosen Plaintext Attack described in pseudo-code

If a cryptosystem is proven to be IND-CPA, it will imply several desirable properties of that system that apply to any adversary. Any adversary will **not** be able to:

- Learn what the message is by encrypting the M_1 or M_2 after step 3. This means that every encryption of the same message is different.
- Learn a connection between message and ciphertext.
- learn any bit of the message given a ciphertext even when the plaintext is known.

2.6 IND-CCA for a PKE

In a modern cryptosystem, only the key should be secret. The inner workings of an algorithm is assumed to be known to an attacker, so we need to create cryptosystems in a way that an attacker can not learn anything about the plaintext when observing the ciphertext.

A cryptosystem can achieve formal security if it can be shown to achieve Indistinguishability under Chosen Ciphertext Attack (IND-CCA). IND-CCA is defined as a game and is described in Figure 2.5. $\text{Exp}^{\text{IND-CCA}}_{\text{PKE}}(A)$ describes how an attacker A can win. A secret and public key is fixed from the keyspace of the algorithm. Then a bit is fixed, and the attacker will select two messages with access to the decryption oracle $\tilde{\mathcal{D}}_{S_k}$ and the public key. The decryption algorithm will allow the attacker to decrypt any ciphertext, but after C^* has been selected, they can not decrypt the selected messages' ciphertexts. The attacker will then guess if the bit was 0 or 1, and wins if they guess correctly.

$\text{Exp}^{\text{IND-CCA}}_{\text{PKE}}(A)$

1. $(s_k, p_k) \xleftarrow{\$} \text{PKE.KeyGen}$
2. $b \leftarrow \{0,1\}$
3. $M_0, M_1 \leftarrow A^{\tilde{\mathcal{D}}_{S_k}(\cdot)}(p_k)$
4. $C^* \leftarrow \text{PKE}.\mathcal{E}_{p_k}(M_b)$
5. $b' \leftarrow A^{\tilde{\mathcal{D}}_{S_k}(\cdot)}(C^*)$
6. **return** $[b = b']$

$\tilde{\mathcal{D}}_{s_k}(C)$

1. **if** $C = C^*$
2. **return** $\perp$
3. **else**
4. **return** $\mathcal{D}_{S_k}(C)$

$$\boxed{\text{Adv}^{\text{IND-CCA}}_{\text{PKE}}(A) \stackrel{def}{=} |2\text{Pr}[b = b'] - 1|}$$

Figure 2.5: Indistinguishability under Chosen Ciphertext Attack described in pseudo-code for a PKE

$\text{Adv}^{\text{IND-CCA}}_{\text{PKE}}(A)$ is the advantage an attacker can achieve over the system which fall on the interval $[0, 1]$. If an attacker can correctly guess b they must have broken the system, so we should expect an attacker to gain no advantage better than $0 + \epsilon$, where ϵ is very small. So we expect an adversary to obtain no better advantage than ≈ 0, which implies that the best they can do is guessing at b.

The definition of IND-CCA might look a bit arbitrary, but if we consider what it would take for an adversary to win this game, we can see that the definition implies some very favorable properties of our PKE.

If an adversary can not obtain a meaningful advantage in this game, they can **not**:

1. Obtain the key.
2. Learn a connection between plaintext and ciphertext.
3. Learn a connection between similar-looking ciphertexts.
4. Encrypt the same message again to compare with the given ciphertext.
5. Learn any bit of the plaintext given the ciphertexts.
6. Predict the structure of the ciphertext.

2.7 KEM

A Key Encapsulation Mechanism (KEM) is often very similar to a PKE, but instead of encrypting a message, a KEM will will encrypt a symmetric key directly. This is how most asymmetric cryptography works because symmetric cryptography is much more effective in terms of time and therefore cost. A Key Encapsulation Mechanism (KEM) is usually a PKE that has been modified to encapsulate a key instead of a message, and one can then construct a PKE using the KEM to derive a shared encryption key.

A common way of constructing a KEM is to take an existing PKE with a weak security goal like OW-CPA and add an encapsulation stage to ensure that derived secret is cryptographically secure.

As seen in Definition 7, encapsulation will generate a random element m from the message space and use a function H to derive a shared secret. H is commonly selected to be cryptographically secure hash function, a function that works as a random oracle.

Definition 6. *Hash function*
A hash function $H : \mathcal{M} \to \mathcal{H}$,
where $\mathcal{H}$ *is some fixed space* $\{0,1\}^l$,
and $\mathcal{M}$ *is the space of all messages* $\{0,1\}^*$

Definition 7. *KEM scheme*
let $(K_{gen}, \mathcal{E}, \mathcal{D})$ *be the components of a IND-CPA secure PKE,*
$H : \mathcal{M} \to \mathcal{K}^*$
$\mathcal{K}^*$ *a symmetrical keyspace*
$Encapsulate : \mathcal{K} \to \mathcal{C} \times \mathcal{K}^*$
$Decapsulate : \mathcal{C} \times \mathcal{K} \to \mathcal{K}^*$ *so that*

$$Encapsulate(P_k) = (c, K), Decapsulate(S_k, c) = K$$

The components of a KEM are $(K_{gen}, \mathcal{E}, Encapsulate, , \mathcal{D}, Decapsulate, H)$.

2.8 NIST PQC Categories

The security of the NIST PQC proposals are measured by adhering to 5 categories, where 5 is the highest and 1 is the lowest. The categories are listed in Table 2.1. The type defines that the system must be at least as hard as exhaustive search for AES, or a collision search for SHA corresponding to the given size of either AES or SHA. For key exchange, categories 1,3, and 5 are the most interesting categories. 2 and 4 are more interesting when discussing signature schemes which are also subjects for the NIST PQC.

Category	Minimum security level	type
1	128	AES
2	256	SHA
3	192	AES
4	386	SHA
5	256	AES

Table 2.1: NIST PQC security categories

Chapter 3

Linear Codes

To understand what a linear code is, let us first build an intuition for what a code is. The word code in this context means that we are talking about encoding something. That is to use a symbolic representation to represent an object that is not the same as the object itself.

Mathematically a code C is uniquely identified as a subspace of a space P. That is to say that a code can be the entire space P or any proper subspace of P. This definition is not very useful as it does not tell us how to encode anything.

To encode a message from another space M we create a mapping $G : M \to C$ so that every element $m \in M$ corresponds to an element in C. With this mapping, we have a way to encode elements in M, but it does not guarantee us a way of decoding them. For instance if every element $m \in M$ maps to the same *codeword* $c \in C$ it is impossible to go the other way.

To be able to decode our encoding, the mapping needs to be reversible. In other words, it needs to be injective and therefore invertible.

The injective mapping between our two spaces $G : M \to C$ implies that there exists an inverse function.

Definition 8. *A code C over F^n is is a linear code iff*
1. *$u - v \in C$ for all $u, v \in C$*
2. *$au \in C$ for all $a \in F, i \in C$*

Where F is a field

3.1 Distance Between Codewords

Every element of our code is a codeword and can be represented as a vector. We then observe that every invalid codeword y in the larger body P will have some distance d to a valid codeword c. If we create a linear code with a clever choice of d, we will then be able to uniquely express $y = c + e$ where $e \in P$ with some low weight. So if a low weight vector e is added to c we will be able to detect it and hopefully correct it.

If we consider fig 3.2 we see that an error occurred under transmission. This low weight error changed the message, and Bob cannot determine what the original message was.

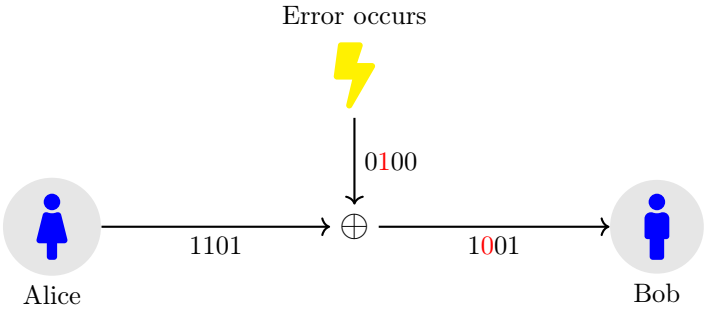

Figure 3.1: Error occurs during transmission.

If now Alice uses a linear code to encode the message as in figure 3.2, Bob will be able to detect that an error occurred during transmission. Because this invalid message has a determinable distance to a valid codeword, it can be corrected.

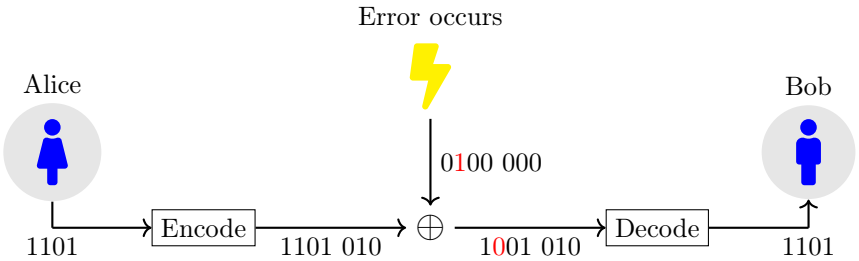

Figure 3.2: Error is detected and corrected.

Definition 9. *Hamming weight* $h_{wt}(v)$
The number of non zero elements in a binary vector $GF(2)^n$. *Ex* $h_{wt}(v) = h_{wt}([1,1,0]) = 2$

Definition 10. *Hamming distance*
$h_d(u,v) = h_{wt}(u-v)$ *for any* $u, v \in GF(2)^n$
Ex $u = [1,1,0], v = [1,0,1] \rightarrow h_d(u,v) = h_{wt}(0,1,1) = 2$

So by using linear codes, we cannot only detect errors in messages but also correct them. To identify the closest codewords uniquely, we need to define our code so that the distance between valid codewords is the same and an odd number. Any invalid codeword can be uniquely decoded into a valid one. This distance between valid codewords is called *minimum Hamming distance*.

To understand the interaction between minimum Hamming distance and correctable errors, consider Figure 3.3. We can see the entire space a code C exists on, where the codewords are evenly spaced out. Each valid codeword has its *hamming bound* drawn around it, which here is a circle of radius $(d-1)/2) = 1$. The bound indicates that every invalid codeword within the bound is uniquely decoded to the valid codeword it belongs to. A code where the bounds cover the entire space (as seen in the Figure) is a *perfect code*.

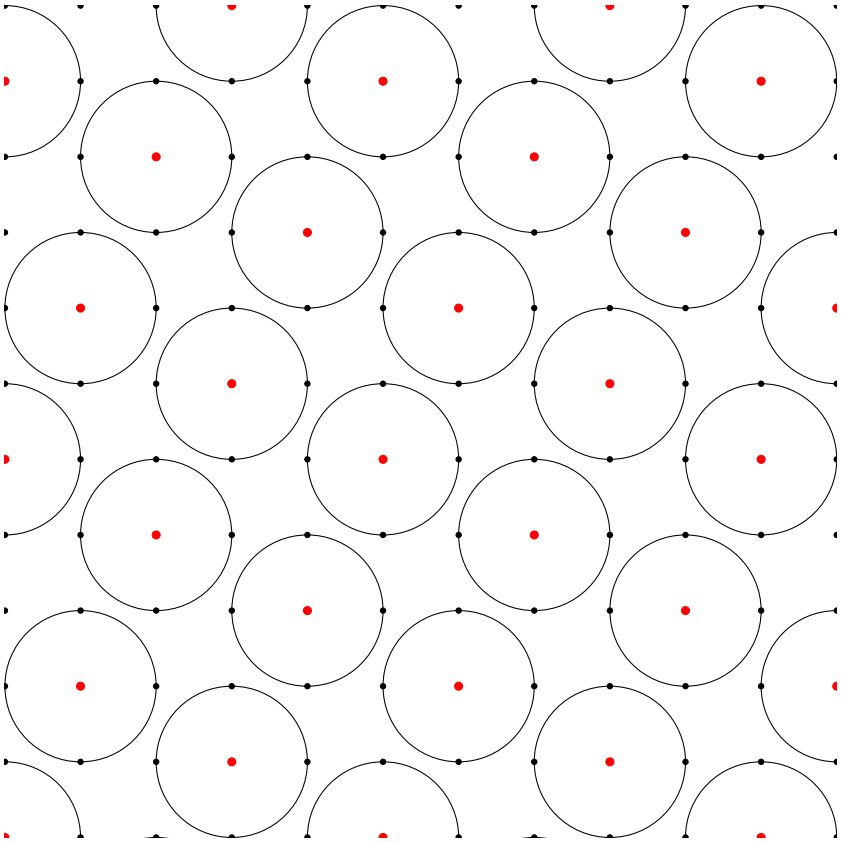

Figure 3.3: Hamming bounds around valid codewords (red) in a perfect code with $d = 3$

Definition 11. *C is a q-ary linear $[n, k, d]$ code if*

1. *C is a subspace of $GF(q)^n$*
2. *C has dimension k*
3. *C has minimum Hamming distance d*

3.2 Generator Matrix

We have talked about a mapping $G : M \to C$ that can be expressed as a matrix G'. This matrix G' is called the *Generator matrix* for C and the structure of G' is so that the rows of G' form a basis for C.

Definition 12. *Let C be a (n, k) linear code over F_q. If a matrix G's rowspace equals C then G is the generator matrix for C.*

3.3 Parity Check Matrix and Syndromes

A parity check matrix H is defined as the matrix that has the relation $H\mathbf{c}^T = \mathbf{0}$ for any $\mathbf{c} \in C$.

Definition 13. *Let C be an (n, k) linear code over F_q. Then a matrix H for which $H\mathbf{c}^T = \mathbf{0}$ iff $\mathbf{c} \in C$*

The parity check matrix is particularly useful because by definition it will essentially negate the any valid codeword when multiplied. If we have a valid codeword $c \in C$, and an error is introduced to the vector $y = c + e$, then the *syndrome* $s = Hy$ will only be a product of e as $s = Hc + He = 0 + He$

The syndrome is, as the word implies, a symptom of an underlying problem and it can tell us where the error is located in $\mathbf{y}$.

3.4 Encoding

Encoding using a Linear code is very fast. If we have a (n, k) linear code C over F_2 with generator matrix G, we select a message from $\mathbf{m} \in F_2^k$ and encode it to $\mathbf{x} \in F_2^n$ by $\mathbf{m}G = \mathbf{x}$.

This is a linear operation and can be implemented very effectively.

3.5 Decoding

Decoding a word $r = c + e$ where $c \in C$ and e is some error, is to find the closest codeword to $r \to c$ or to find the error e.

Definition 14. *The general decoding problem*
let C be a linear code and r be a received word on the form

$$r = c + e$$

Where $c \in C$ and e is a random low weight vector.
The Decoding Problem is to find m or e given r and C, and is split into two problems.

Definition 15. *Decoding problem 1 - Syndrome decoding*
let C be a (n, k) linear code with parity check matrix H.
Given a syndrome $s = He$, determine e of weight t such that

$$He = s$$

Definition 16. *Decoding problem 2 - Codeword finding*
let C be a (n, k) linear code with parity check matrix H.
determine c of a given weight t such that

$$Hc = \mathbf{0}$$

In some cases, the error vector will have a predetermined weight that will allow unique decoding. As long as the code has minimum hamming distance d, unique decoding will be possible with $h_{wt}(e) <= \lceil \frac{d-1}{2} \rceil$. This is called half-distance decoding, and most decoding algorithms target this type of decoding.

If e can also be $\lceil \frac{d-1}{2} \rceil < h_{wt}(e) <= d$, then decoding is still possible, but decoding will not have a unique answer. There might be several codewords with the same minimum distance from r therefore decoding might produce a list of closest codewords and different solutions for $r = c + e$. This is called

full-distance decoding and requires list decoding algorithms that will produce a list of solutions. This also makes consideration of which error weight to be prioritized. There might be different solutions with different error weights, and it is not obvious which one is the correct weight.

Decoding algorithms depend on what code one is using. The simplest way of decoding is to create a syndrome using the parity check matrix of the code. By adding an error to a code C, one will create a new code C' in which all the elements $c' \in C'$ will have the same syndrome. If the code is sufficiently small, one can enumerate all these sets and find which set r belongs. The lowest weight vector, coset leader, in this set will be the error e.

Coset decoding is impossible to perform if the code grows too large to feasibly enumerate all the cosets, and this is why there are decoding algorithms tailored for each class of code.

The general decoding problem does not require a decoding algorithm. It is possible to decode any word $r = c + e$, as an attacker will have access to the generator matrix [1], and because the rows of this matrix form a basis for C they have access to the entire code. It is then possible for the attacker to solve $r = c + e$ by at least enumerating $c \in C$. The general decoding problem has proved to be computationally hard, and is classified as NP-complete [13].

3.6 Hamming Codes

Hamming codes are an easy-to-understand subset of linear codes. They are constructed by adding parity check bits to create a codeword. These parity check bits will be a function of the message bits.

They are parameterized with $[n, k, d]$. Hamming codes are perfect codes as long as $n = 2^r - 1$ and $k = n - r$ for $r \in \mathbb{Z}$ and $r > 2$ and $d = 3$.

To construct a $[7, 4, 3]$ hamming code, one only needs to decide three parity check bits x, y, z to append to the message bits m_1-.

$$\begin{aligned} x &= a + b + d \\ y &= a + c + d \\ z &= b + c + d \end{aligned}$$

Where all arithmetic is done in $\mathbb{Z}_2$. We can then construct the generator matrix for the code:

$$G = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Because the generator matrix is already in the form:

$G = [I_k \mid T_{n-k \times k}]$ we can easily construct the parity check matrix by $H = [-T^t \mid I_{n-k}]$.

[1] An attacker might have access to a scrambled version of the generator matrix, but they will be able to construct an equivalent code through linear algebra.

$$H = \begin{bmatrix} 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{bmatrix}$$

Now every vector in $m \in Z_2^4$ encoded with G by $y = mG + e$ with e being a random error of weight 1, will be uniquely identifiable by the syndrome $s = Hy$. If s is non zero, this means that e is non zero and because e can only be of weight 1, and that mG cancels out by H; then the syndrome is only a function of the column in which the error occurred.

So to decode a hamming code, one can simply determine the syndrome which will be the same as a column in H, and the index of this column will be the location of the error in y. This only works because the syndrome is a function of only one error, and if there were multiple errors; then the syndrome would be some linear combination of multiple columns, which is harder to determine.

3.7 Cyclic Codes

Definition 17. *A cyclic code is a linear code C where any circular shift of a codeword $c \in C$ is also in c.*

Cyclic codes are constructed using polynomial rings $\mathcal{R} = \mathbb{F}_2[X]/(X^r - 1)$. The generator matrix is then uniquely defined by the first row:

$$r_1 = p_0 + p_1 x + \cdots + p_r x^{r-1}$$

The second row:

$$r_2 = x r_1 \bmod X^r - 1$$

The third row:

$$r_2 = x^2 r_1 \bmod X^r - 1$$

and so on.

3.7.1 Parameters

A cyclic code has codelength n, dimension k, and row density w, where typically $w = O(\sqrt{n})$.

3.7.2 Decoding Cyclic Codes

Cyclic codes are often decoded using probabilistic algorithms and must be adjusted to have an acceptable decoding failure rate (DFR). DFR is the rate at which decoding either fails.

Bit Flipping

Gallager's bit flipping [14] is a probabilistic decoding algorithm that allows correction of $\mathcal{O}(\sqrt{n})$ errors with high probability.

Because cyclic codes are sparse, it is possible given a syndrome $s = He$ to check the probability that every position of e is in error.

Bit flipping is sequentially guessing that $e_i = 1$ based on the $P_r[e_i = 1 \mid s]$ which can be calculated.

Input parity check matrix H, ciphertext c Maximal iterations X, Maximal syndrome weight u
Output error vector e

1. $s \rightarrow Hc$
2. for i in $0...X$:
   (a) $th = computeThreshold(s, e)$
   (b) for j in $0...n-1$:
      i. $unsatifiedParityChecks = H_i \cdot s$
      ii. if $unsatifiedParityChecks > th$ $e[j] = e[j] \oplus 1$
   (c) $s = H(c + e)$
3. output e if $h_{wt}(s) = u$ else $\perp$

Figure 3.4: Bit flipping algorithm in [15]

3.8 Binary Goppa Codes

A (n, m, t) Goppa code C is a linear code of length n that correct t errors and m is the size of the finite field used. C is uniquely defined by a irreducible polynomial $g(x)$.

3.8.1 Parameters and Setup

To construct a Goppa code n, m and t are selected for desired error correcting capability t. When the parameters are chosen, the code is constructed by selecting a random list L of n elements in $GF(2^m)$ called the *Support*, and a random monic polynomial $g(x) \in GF(2^m)[X]$ of degree t for which $g(L) = 0$ called the *Goppa Polynomial*.

Support

The support is a set of n elements from the finite field $GF(2^m)$. This field is usually constructed with an irreducible polynomial h of degree m and all field arithmetic is done in the extension field $GF(2)/h$. The support L is then n random elements in this field.

$$L = [\alpha_0, \alpha_1, \cdots, \alpha_{n-1}]$$

Generating the support is done:

Goppa Polynomial

A Goppa polynomial g is a monic polynomial of degree t and coefficients in $GF(2^m)$, with the added requirement that $g(\alpha_i) \neq 0$ for $i \in [0, \cdots, n-1]$. This can be achieved by selecting an irreducible polynomial, or a square-free polynomial.

3.8.2 Decoding Goppa Codes

There are two common algorithms for decoding Goppa Codes; Patterson [16] and Berlekamp-Massey [13].

$$H = \begin{bmatrix} \frac{1}{g(\alpha_0)} & \frac{1}{g(\alpha_1)} & \cdots & \frac{1}{g(\alpha_{n-1})} \\ \frac{\alpha_0^1}{g(\alpha_0)} & \frac{\alpha_1^1}{g(\alpha_1)} & \cdots & \frac{\alpha_{n-1}^1}{g(\alpha_{n-1})} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\alpha_0^{t-1}}{g(\alpha_0)} & \frac{\alpha_1^{t-1}}{g(\alpha_1)} & \cdots & \frac{\alpha_{n-1}^{t-1}}{g(\alpha_{n-1})} \end{bmatrix}$$

Figure 3.5: Parity Check matrix for A Goppa Code

The main idea of decoding a received word encoded with a Goppa code is finding an error locator polynomial σ that will have roots in L. When σ is found L is evaluated such that $\sigma(\alpha_i) = 0$ indicates a 1 in the error vector e.

Patterson's Algorithm

The algorithm is outlined in Figure 3.6. Step a) can be done with Euclid's Algorithm, which will have a variable runtime. It is unknown if Patterson's can be completed in constant time, and there are timing attacks that have been shown to exist in [17]. This makes Patterson unsuitable for cryptographic purposes without mitigation.

1. Compute syndrome $s(z) = \sum_{i=1}^{n} \frac{r_i}{z-\alpha_i}$
2. Determine error locator polynomial $\sigma(z)$ by:
   (a) Determine $h(z)$ such that $s(z)h(z) = 1 \mod g(z)$. if $h(z) = z$ let $\sigma(z) = z$
   (b) Calculate $d(z)$ such that $d^2(z) \equiv h(z) + z \mod g(z)$
   (c) Find $a(z)$ and $b(z)$ with $b(z)$ of least degree, such that $d(z)b(z) \equiv a(z) \mod g(z)$
   (d) Put $\sigma(z) = a^2(z) + b^2(z)z$
3. Determine the set of α_i such that $\sigma(\alpha_i) = 0$
4. The error positions are then found by the corresponding indices i.

Received word is z, $g(z)$ is the Goppa Polynomial, and the set of $\{\alpha_i\}$ is the Goppa support.

Figure 3.6: Parrerson Decoding algorithm with a Goppa Code

Berlekamp-Massey

Berlekamp-Massey (BM) [13] is a decoding algorithm that allows decoding of $t/2$ errors under a Goppa code. In order to find all errors a trick was proposed in [18] where the syndrome is doubled and arithmetic is done under the Goppa polynomial $g(x)^2$.

BM initializes the error locator polynomial $\sigma(x), \beta(x) \in GF(2^m)[x]$ and integers $l, \delta \in GF(2^m)$. It then incrementally updates these values for $2t$ iterations and leaves $\sigma(x)$ as the true error locator polynomial.

Input: t, syndrome $S(x)$
Output: Error locator polynomial $\sigma(x)$

1. Initialize $\sigma(x) = 1, \beta(x) = x, l = 0, \delta = 1$
2. for $k \in [0, 2t-1]$
3. $\quad d = \sum_{i=0}^{t} \sigma_i S_{k-i}$
4. $\quad$ **if** $d = 0$ or $k < 2t$:
5. $\quad\quad [\sigma(x), \beta(x), l, \delta] = [\sigma(x) - d\delta^{-1}\beta(x), x\beta(x), l, \delta]$
6. $\quad$ **else**:
7. $\quad\quad [\sigma(x), \beta(x), l, \delta] = [\sigma(x) - d\delta^{-1}\beta(x), x\sigma(x), k - l + 1, d]$
8. **return** $\sigma(x)$

Figure 3.7: Berlekamp-Massey algorithm in pseudo code

3.8.3 Beyond t

If a linear code C has error correction capability t, it is possible to introduce additional errors to break the uniqueness of decoding. As mentioned in Section 3.5 this requires list decoding and also has the benefit of making attacks more complicated [19].

A list decoding algorithm for binary goppa codes is outlined in [20] which can decode approximately $n - \sqrt{n(n-2t-2)}$ errors.

Chapter 4

McEliece

McEliece is a code based Public Key Encryption scheme (PKE), and was introduced by Robert McEliece in 1978 [4]. In a general McEliec scheme the compents are:

C : A linear code with generator matrix $\hat{G}$, parity-check matrix H and a chosen error-correction capability t.

S : A non-singular matrix to provide diffusion.

P : A permutation matrix to provide confusion.

The public key is constructed $G = S\hat{G}P$, and data is encrypted by $c = m\mathrm{P_k}+e$ where e is a vector of hamming weight t (see Figure 4.2).

The Secret/Private key consists of S, P and a decoding algorithm $\mathcal{D}$, and the c is decoded using S^{-1}, P^{-1} and $\mathcal{D}$. What is important to notice that decryption is more complicated than encryption, and some decoding algorithms cannot be implemented in constant time.

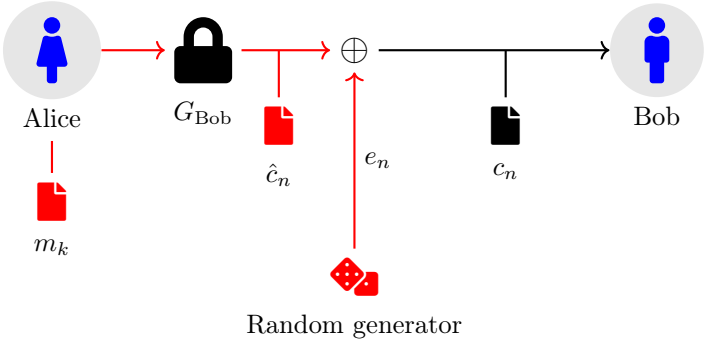

Figure 4.1: Encryption in a standard McEliece Cryptosystem.

4.1 Niederreiter

Now that we have all the building blocks defined, we are ready to look at the Niederreiter variation of McEliece. Niederreiter is a dual variation of McEliece and was proposed by Harald Niederreiter in 1986 [21]. In the system, one uses the parity check matrix for encryption, essentially going backward.

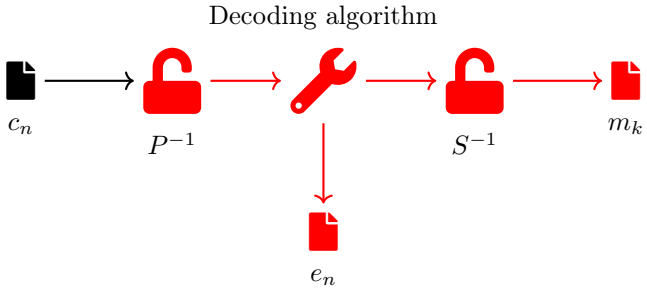

Figure 4.2: Decryption in a standard McEliece Cryptosystem.

Codelength n, Dimension k and t correctable errors $\hat{G}_{k\times n}$ - Generator matrix for random linear code C $S_{k\times k}$ - Non singular matrix $P_{n\times n}$ - Permutation matrix		
Key	S_k $(S, \hat{G}, P)$	P_k $G_{k\times n} = SGP$
Encryption	$h_{wt}(e) = t$ $\mathcal{E}(e) = mG + e = c$	
Decryption	$\hat{c} = cP^{-1} = mSG + eP^{-1}$ $\mathcal{D}(\hat{c}) \to mS$ $m = mSS^{-1}$	

Table 4.1: Standard McEliece Cryptosystem

It is possible to construct a Niederreiter system from any linear code, but many systems using codes that allow smaller key sizes have been found to be insecure.

The most confidence-inspiring code is still the same as the original proposal [4] which uses binary Goppa codes.

Codelength n, Dimension k and t correctable errors $\hat{H}_{(n-k)\times n}$ - Parity check matrix for random linear code C $S_{(n-k)\times(n-k)}$ - Non singular matrix $P_{n\times n}$ - Permutation matrix		
Key	S_k $(S, \hat{H}, P)$	P_k $H_{k\times n} = S\hat{H}P$
Encryption	Encode m as $e, h_{wt}(e) = t$ $c = He$	
Decryption	$\hat{c} = S^{-1}c = \hat{H}Pe$ $\text{Decode}(\hat{c}) \to Pe$ $e = P^{-1}Pe \to m$	

Table 4.2: Niederreiter Cryptosystem

4.1.1 Keysizes

Key sizes using Goppa codes are relatively large in the PQC landscape. From table 4.3 we see that even when limiting the analysis to only code-based systems, Goppa codes are orders of magnitude larger than the others.

Code	Implementation	Security level	Size	%
Goppa	CM [22]	266	1022 kB	100%
HQC	HQC [7]	256	7.245 kB	0.7%
QC-MDPC	BIKE [6]	256	5.121 kB	0.5%

Table 4.3: McEliece Systems with different codes

4.1.2 Reduction in Message Space

Using the Niederreiter scheme, one has to encode the message to a binary vector of length n and weight t. This means that not the entire code can be used.

In the regular setup, the dimension would be k, which would allow the encryption of 2^k distinct messages.

By representing the message as an error, the maximal possible messages are instead the number of vectors of length n and weight t. So for a (n, m, t) code C, if used in a Niederreiter scheme:

$$|C_{\text{Niederreiter}}| = \binom{n}{t}$$

And we can find the message bits of our code by $\log_2(|C_{\text{Niederreiter}}|)$. The parameter choices for [3] are listed and compared in Table 4.4

Parameters n/m/t	McEliece message bits	Niederreiter message bits
3488/12/64	2720	456
4608/13/96	3360	668
6688/13/128	5024	908
6960/13/119	5413	863
8192/13/128	6528	946

Table 4.4: Maximum message bits for a McEliece scheme with CM parameters

Chapter 5

Programmable Logic

We often encounter problems where computational time is a bottleneck. Modern CPUs are highly versatile but often unsuitable for solving specific problems in real-time on low-cost hardware. To address this issue, it is common to design specialized hardware that drastically reduces the time needed for computation.

Examples of problems like these are decoding Reed-Solomon codes in CD players [23], [24], or image recognition in demanding areas [25]. Without specialized hardware to perform computations; it would be impossible to have these technologies working in real-time.

It is assumed that the reader has some knowledge about programming and software implementations, and in this chapter, the basic outlines of hardware development will be outlined.

To test hardware components, it is vital to have platforms that allow us to change the designs without creating the circuits physically. Devices that allow the user to configure the physical layout of the chip are called programmable logic (pl) devices. There are many types of pl devices and some will allow the user to reconfigure the device any number of times.

5.1 FPGA

A field programmable gate array (FPGA) is an integrated circuit that can be programmed to replicate a hardware design. An FPGA is one of many programmable logic devices, but for this text, we will use FPGAs and programmable logic interchangeably.

The basic idea is that a FPGA is a chip with a set number of IO ports. Within the chip is a large number of LUTs that can be configured and routed arbitrarily to produce any desired behavior of the IO ports. What is so remarkable about this is that this can be done within a single clock cycle, and the programmer is only limited to how fast the actual electrical signal can propagate through the inner circuits. This is a common problem in hardware development and more complex logic will result in a longer path for the signal to travel, and thus the clock speed might suffer to make sure the signal has time to propagate.

5.1.1 Hardware Description Language

There are two primary language's that are used for FPGA development; Verilog and VHDL. These languages are called hardware description languages (HDLs) and are similar to programming languages but function quite differently.

Verilog, or SystemVerilog, is standardized as IEEE 1800 [26]. At the time of writing, there are about 8 thousand repositories on GitHub that contains Verilog or SystemVerilog.

VHDL, or Very High-Speed Integrated Circuit Hardware Description Language, was developed in 1983, and the latest version is standardized as IEEE 1076-2008 [27]. At the time of writing, there are about 450 repositories that contain VHDL.

As Norwegian businesses more widely use VHDL, it is the target language of this work.

In order to use a hardware design written in a HDL one can not compile it and run it on a CPU. First, the design is usually verified in simulation software. When it has been shown to behave as expected, it synthesized using proprietary tools to produce an actual hardware design for the target platform. Unfortunately, these tools are often quite expensive, and there are no open-source alternatives for design synthesization.

5.1.2 Manufacturers

The two leading manufacturers of FPGA are Intel/Altera and Xilinx. They both offer a wide range of FPGA, and both have proprietary synthesization software. Intel's software is Quartus®, and Xilinx has Vivado®.

5.1.3 HLS

High level synthesis (HLS) is a new way of synthesizing hardware designs from more conventional programming languages. With HLS the user can take existing functions written in c, and synthesize these as hardware components.

Performance of HLS is not as good as writing components directly in a HDL, but this can be improved by writing/rewriting the function specifically for HLS. Kris Gaj et al. reported in a seminar [28] that by using HLS for their various NewHope [29] and Kyber [30] implementations, the clock frequency was *reduced* by 17%, and LUT usage was increased by $14 - 76\%$. So if this is a general trend, using HLS, we can expect slightly slower clock speeds and higher logic usage.

HLS also dramatically reduces development time by eliminating the need for component simulation. To verify a component written in a HDL one typically will need to write an extensive test bench and run in through simulation software such as Modelsim®. This is a time-consuming and often difficult effort as bugs or unexpected behavior will leave the developer staring at wave plots for hours on end. With HLS the developer can verify the behavior of the components in software, and writing unit tests for a function is often much more straightforward than writing a good test bench.

These are reasons to be optimistic about HLS as a concept, and there is no reason to believe that the technology will not improve with time and additional

iteration. If HLS closes the performance gap to traditional HDL development it will leave little argument to even use HDLs.

5.1.4 Performance Metrics

To compare how good a hardware component is, we often look at its footprint on the FPGA. This footprint is measured in LUTs, as this is the primary building block the FPGA uses to construct circuits. Additionally, different hardware components may allow different maximal clock speeds, which means that two similar components may differ in both utilization and timing, which leads to the performance metric time $\times$ space.

For this work, we will be mainly concerned about the utilization in LUTs and the simulated speed in clock cycles to obtain the time $\times$ space metrics.

Performance metrics to compare FPGAs are also number of LUTs and maximal clock frequency, but modern FPGAs have become increasingly sophisticated devices with many features that, in some cases, makes them hard to compare to each other. Some boards like those in the Xilinx Spartan®series are low cost I/O optimized boards, while those in the more expensive Zynq®series will have built-in ARM processors that greatly increases the functionality of the board.

Further, it is essential to note that specific hardware designs must be synthesized for a specific platform to give an accurate number on utilization, and the LUT usage may vary from board to board.

5.1.5 Xillinx FPGA Board Survey

This section will look at some of the Xilinx FPGA boards on offer to get more familiar with what is on offer. There are numerous boards that are split into some notable families. We will limit the survey to the following cost-optimized families. The goal is to give the reader an idea of what boards are on the market and how board selection can be made.

Numbers are gathered from Xilinx's cost-optimized product selection guide. Each slice contains 4 Luts and eight flip-flops.

Spartan®-6 Models in the Spartan®-6 family are general-purpose I/O FPGAs. Typical use cases for these boards are any projects that require custom any-to-any connection like traffic lights, elevators, or vending machines.

The Spartan®-6 family was hugely popular and received an extended lifecycle support until at least 2027. They have short lead times, and Xilinx recommends this board as the default choice if the customer needs to start development immediately.

Models in this family are built on a 45nm process.

A summary of specifications of the Spartan®-6 family is found in table 5.1.

Spartan®-7 The Spartan®-7 family is an iteration of the spartan®-6 family. The models in this family are built on a 28nm process and provide roughly 2.5 times the performance per watt as Spartan®-6.

The use cases for Spartan®-7 models are the same as the Spartan®-6, but if performance or power consumption is an issue, the Spartan®-7 is to be preferred.

[1]Each slice contains 4 usable LUTs and 8 flip-flops

Spartan®-6 Family	
Models (13)	XC6SLX4, XC6SLX9, XC6SLX16, XC6SLX25, XC6SLX45, XC6SLX75, XC6SLX100, XC6SLX150, XC6SLX25T, XC6SLX45T, XC6SLX75T, XC6SLX100T, XC6SLX150T

	Minimum	Maximum
Logic Cells	3840	147443
Differential I/O Pairs	66	288
Single-Ended I/O Pins	132	576
Slice[1]	600	23,038
Total Block RAM (Kb)	216	4824
LUTs	2400	92,152

Table 5.1: Spartan®-6 board specifications

A summary of specifications of the Spartan®-7 family is found in table 5.2.

Spartan®-7 Family	
Models (6)	XC7S6, XC7S15, XC7S25, XC7S50, XC7S75, XC7S100

	Minimum	Maximum
Logic Cells	6000	10,2400
Differential I/O Pairs	48	192
Single-Ended I/O Pins	100	400
Slices	938	16,000
Distributed RAM (Kb)	70	1100
LUTs	3,752	64,000
DSP F_{max} MHz	464	550
BRAM F_{max} MHz	388	461

Table 5.2: Spartan®-7 board specifications

Artix®-7 Models in the Artix®-7 family will, unlike the Spratan®families, have high-speed transceivers that enable them to perform tasks that require high bandwidth. Applications that involve communication over USB, PCI-E, or driving HDMI signals will be best suited with models in this family.

This family is also suited for cryptographic purposes, as interfacing with the hardware will usually require some connection to a computer from PCI-E or USB interface.

A summary of specifications of the Artix®-7 family is found in table 5.3.

Zynq®-7000 The models in the Zynq®-7000 family are different from the previous families as they will have integrated CPUs. This enables these models to be fully SoC as an application can run on the CPU, which can interface directly with the FPGAs.

A summary of specifications of the Zynq®-7000 family is found in table 5.4.

Artix®-7 Family		
Models (8)	XC7A12T, XC7A15T, XC7A25T, XC7A35T, XC7A50T, XC7A75T, XC7A100T, XC7A200T	
	Minimum	Maximum
Logic Cells	12,800	215,360
Differential I/O Pairs	72	240
Single-Ended I/O pins	150	500
Slices	2000	33,650
Total Block RAM (Kb)	720	13,140
LUTs	8000	134,600
PCIe Gen2	x1	x1
DSP F_{max} MHz	464	628
BRAM F_{max} MHz	388	509

Table 5.3: Artix®-7 board specifications

Zynq®-7000 Family		
Models (10)	XC7Z007S, XC7Z012S, XC7Z014S, XC7Z010, XC7Z015, XC7Z020, XC7Z030, XC7Z035, XC7Z045, XC7Z100	
Processors	Single-core ARM®Cortex®A9 766 MHz Dual-core ARM®Cortex®A9 766 MHz Dual-core ARM®Cortex®A9 1 GHz	
	Minimum	Maximum
Logic Cells	23,000	444,000
Flip-Flops	28,800	554,800
Total Block RAM (Kb)	1800	26,500
LUTs	14,400	277,400
PCIe Gen2	-	x8

Table 5.4: Zynq®-7000 board specifications

Chapter 6

Attacking McEliece

The security of McEliece has stood the test of time rather well. With decades of attack papers, only marginal compromises have been achieved.

6.1 Information Set Decoding

Information Set Decoding (ISD) is an unstructured decoding attack against a linear code. ISD will allow the decoding of a word $r = c + e$ with a given code C.

In short ISD are clever ways of searching a linear code for a codeword of some property. Given a random linear code C ISD will either find e given the syndrome $s = He$ and parity check matrix H, or it will find e given $y = c + e$ where $c \in C$.

ISD was originally proposed in [31], and has since seen a number of improvements.

There are many variation's of attacks, but one prevalent method is to use Stern's attack algorithm [32]. In the case that an attacker has obtained a ciphertext $y = c + e$ and has access to the generator matrix G, the attacker can create a new code $C + 0, y$. With this new code, they can be sure of two things, it definitely contains y, and it definitely contains e as C contains c and $c+y = e$. They can then use Stern's attack algorithm to find e in $C + 0, y$. This increases the complexity of the decoding attack as it increases the dimension of the code by 1.

Stern's attack algorithm [32] is an algorithm for finding a low-weight codeword in a random linear code. It has inputs: an integer $w >= 0$ and a parity check matrix H for a random (n, k) linear code C. With linear algebra, an attacker can construct the modified parity-check matrix from the modified generator matrix as discussed above, which includes the ciphertext.

The attacker then feeds Stern the modified parity-check matrix and $w = t$, and stern can solve to find e.

An improved variant of Stern's attack algorithm was in [19] used to successfully break the original parameters suggested by McEliece in [4]. The same article also suggests new parameters, which are some of the same as the ones used later in Classic McEliece [33].

6.1.1 Speed of ISD Attacks

The dimension k of a code C tells us the size of the code is 2^k. A brute force to look for a specific element in this body, the running time will be on average $2^k/2 = 2^{k-1}$.

The original ISD proposed [31] achieved a worst-case running time of $2^{0.121n}$, and has seen several improvements since it was published. A more recent work [34] achieved a running time of $2^{0.0885n}$. These running times are for full length decoding, and [34] achieves higher speedups on higher error rates.

ISD attacks are the best-known attacks to the general decoding problem. We can see that they hold substantial improvements in running time over a brute force search, and it is, therefore, vital to consider the cost of ISD attacks when choosing parameters for a code-based cryptosystem.

6.2 Oracle Decryption

The structure of the McEliece cryptosystem, in this case, the Niederreiter variation, unfortunately, lends itself to a very simple oracle decryption attack.

If a McEliece scheme explicitly rejects invalid ciphertexts, it is vulnerable to this attack.

If an attacker intercepts a ciphertext encrypted by this scheme, they may use the public key to iteratively modify the ciphertext. The modified ciphertext is then submitted to decryption, and if it is rejected, the attacker learns something about the plaintext.

In the Niederreiter scheme, the plaintext m corresponds to an error vector of weight t. It is encrypted with the public key $c = P_k m$ and the attacker obtains c. The attacker will then add the i-th column of the P_k to c, and submit it to the oracle for decryption. The addition of the i-th column will then correspond to a flipping of the i-th bit in m. If this already was a 1, then an error is canceled out, and the decryption will not fail. If it was not a 1, then a new error was introduced to m. This will result in an error vector of weight $t+1$, and because the code can only correct t errors; the decryption will fail.

If the system informs the attacker that encryption has failed, they will then have a way of determining the entire m in n iterations. They will be able to do this by iteratively selecting $i \in [0, n-1]$, adding the i-th column of P_k. They can then determine i-th position of m by decrypting this modified message, 1 in it is successful and 0 if it fails.

This attack method was improved in [35]. Because of the sparse nature of m it proved more effective to submit changes in batches to determine more of m per decryption. The oracle was constructed by listening to decryption on hardware. This shows that even if the implementation is protected with implicit rejection, the implementation might still be broken if an attacker can find other channels of analysis.

It is challenging to mitigate such an attack fully. The system must not explicitly inform the user of failed decryption, and instead present the user with some value. Further, the interaction with the system must be done in constant time so that the attacker can not determine successful decryption by timing the execution. We also see from [35] that the system must be protected from unauthorized access during execution. Side-channel attacks are outlined

in section 6.3.

6.3 Side-Channel Attacks

A cryptographic scheme might have unintended behavior that leaves it vulnerable through direct interaction plaintext and ciphertext. Additionally, a considerable threat to security is the information that leaks during interaction with the system.

As seen in Figure 6.1, an attacker (which as always is Oscar) can access information by observing the system through a side channel. A *side channel* can in this context be regarded as any way of obtaining information about a cryptographic system without direct interaction. When a cryptographic system is executing on a device, it will leak information in various forms. It will consume more power, components will emit electromagnetic radiation, it will produce more heat, the execution might vary in timing, or the device might produce a sound that can identify operations.

It is not always obvious that a cryptographic scheme leaks information. In [17] it is demonstrated that a straightforward Patterson implementation is vulnerable to two message recovery side-channel attacks in the form of timing.

The first lets the user detect differences in time-based on polynomial evaluation in the error locator polynomial (ELP). The degree of the ELP relies on the weight of the error vector, so if the attacker sequentially flips the bits of an intercepted ciphertext, they can learn by timing the execution of the bit was an error or not. When the entire n-bit ciphertext is analyzed like this the attacker has reconstructed the entire error vector and can easily obtain the message. To mitigate this attack, a technique is proposed of artificially raising the degree of the ELP if it is of a lower degree than t.

The second allows the attacker to time the determination of the ELP. The attack procedure is quite similar, but in this case, when the bits are flipped, the determination of ELP will take longer if the bit was not an error and shorter if not. The effect is also the same, allowing the attacker to fully recover the error vector and more easily determine the original message. The increased time comes from iterations of the extended euclidean algorithm, and the proposed countermeasure is to assure that the algorithm continues. This is done by modifying the degree of the intermediate polynomials a and b.

In [17] a similar attack was implemented by constructing an oracle from differential power analysis. By being able to detect decoding failure in power consumption, the authors were able to construct an oracle. Even though the implementation was protected, it leaked the information it tried to obscure in the form of power consumption.

6.3.1 Operating System

In recent years several side channels have been found in the operating system/hardware itself. Vulnerabilities like Meltdown [36] and Spectre [37] show that an application might be secure but still, be vulnerable because of the platform it runs on.

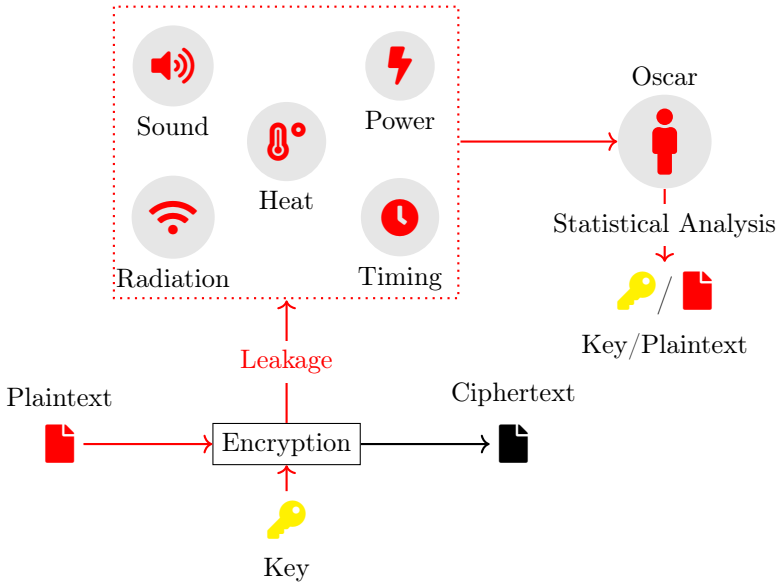

Figure 6.1: Exploitable leakage in an unprotected symmetric cryptographic implementation

6.3.2 TEMPEST

Telecommunications Electronics Material Protected from Emanating Spurious Transmissions (TEMPEST) [38] is a security standard that defends against another form of side-channel; the electromagnetic radiation the device emits. A device will produce radiation when it consumes power, which can be detected and analyzed.

One of the most common ways this can be exploited is through leakage of video signals on computers [39], [40], [41] which can allow an attacker to eavesdrop on the video signal. This is a huge security concern and needs to be mitigated if confidential information is to be shown on a computer screen.

To mitigate against attacks like this, devices and video cables have to be shielded from emitting radiation. An interesting mitigation is discussed in [42], where the authors developed a font that becomes unreadable through minimal noise.

6.4 What is a Secure Cryptographic Implementation

By looking at these attacks and vulnerabilities, it is clear that a secure cryptographic implementation is quite hard to produce, and perhaps impossible to ever guarantee that it truly is secure.

However, we can identify some properties that characterize a secure system. A secure asymmetric cryptographic system

- Encrypts information under a scheme (usually symmetric) that provides 128-bit security

- provides a KEM that has 128 bit security and IND-CCA.
- runs on a trusted platform.
- is shielded from emitting electromagnetic radiation to threat agents.
- executes in constant time for all inputs.
- consumes a constant power during execution or hides the power consumption of cryptographic operations.
- is protected from unauthorized access. This can be done either by having the device in a secure environment or having anti-tampering mechanics that can detect hardware changes or if the device has been opened.

We can summarize from this that the design of the cryptographic scheme is just a tiny part of a secure implementation. Many of these properties are completely impossible to guarantee, and we never know what will happen in the future to compromise them. This is why most cryptographic systems or security measures are only designed to provide security for an estimated period, and there is no such thing as indefinite security.

6.4.1 Anti Tampering

It is pretty common to see self-destructing messages in spy movies. These are often quite impractical as they do not authenticate the listener, and an adversary can potentially obtain the message before James Bond gets access to it. These messages could use anti tampering (AT) in that they should detect when they have been obtained by an untrusted party and destroy the message before they can obtain it.

AT, or tamper-proofing, is any mechanism or device which purpose is to detect or prevent changes and potentially trigger a reaction. It can be things like packing tape that leaves marks on the package when removed, but in this context we are concerned about AT mechanisms that prevent attackers from modifying or obtaining secret information.

A device might have mechanical systems that destroy the device if it is dismantled. It might have electrical trip devices that detect that it is being opened. It might have anti-tamper chips on the device itself to detect strange behavior. The detected malicious behavior might then be set to trigger a number of actions. It might wipe the device's memory, be rigged to destroy a vital component or modify all confidential data to be useless.

Cybersecurity is a well-known and often emphasized issue. AT is equally important and, in many cases, a lot harder. AT is asking for security even when a device falls into enemy hands. In comparison, cybersecurity only protects against adversaries interacting with known endpoints.

Xilinx and Altera both provide guidelines on how to develop tamper-resistant FPGA designs [43], [44]. These documents outline how to make use of built-in AT mechanisms.

Xilinx categorizes its mechanisms as active or passive measures. Passive measures are security measures that protect the user's design by default. These are measures like authenticating and encrypting the bitstream to ensure no leakage occurs while the designs are configured. Active measures are security measures that require the user to implement them in the design. These will, for instance, allow the user to specify that an AES key is to be deleted on power loss.

Xillinx also splits AT into three categories.

1. Prevention
2. Detection
3. Response

These categories help understanding the scope of AT in general. They can either prevent tampering, detect it, or respond to tampering.

Part II

The code-based NIST PQC candidates

Here we will look at the code-based NIST submissions and gather information that will allow us to directly compare them. Some will also have partial hardware implementations that demonstrate the speedups that can be achieved.

Chapter 7

Classic McEliece

Classic McEliece (CM) [3] is a NIST PQC finalist for round 3, and is a Niederreiter based McEliece KEM. It was selected to be a finalist because of the well-known construction of the Goppa code McEliece system [45]. The scope of CM is to be a KEM achieving IND-CCA2 with parameters for category 1,3 and 5 security levels (see NIST). The main focus of the implementation is security. It is implemented in C.

CM builds on multiple McEliece implementations as seen in Figure 7.1.

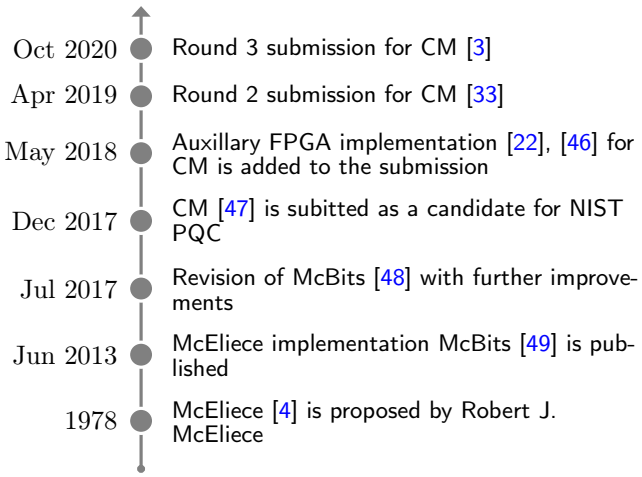

Figure 7.1: CM timeline

7.1 Niederreiter KEM

CM is a full KEM (Section 2.7) implemented with the Niederreiter variant of McEliece (Section 4.1) using binary Goppa codes (Section 3.8).

We have seen from Section 2.3 that a PKE consists of $(K_{\text{gen}}, \mathcal{E}_{P_k}, \mathcal{D}_{S_k})$, and from Section 2.7 a KEM additionally requires $(\text{Encaps}, \text{Decaps}, H)$. Let us go through these and define these for the KEM used in CM [33].

In underlying PKE is defined in [49] and [48].

7.1.1 Parameters

The adjustable parameters for a CM KEM are:

n - Codelength of the goppa code used.
m - Size of the finite field used q is defined by $q = 2^m$.
t - Number of correctable errors for the Goppa Code.

The code dimension $k = n - mt$.

The submission also has parameter choices for category 1,3, and 5 as seen in Table 7.1. The parameter set 6960/13/119 appears to be the best trade-off in most cases for category 5, so this is the parameter set that will be used in the later comparison with BIKE and HQC.

NIST PQC Category	n	m	t
1	3488	12	64
3	4608	13	96
5	6688	13	128
5	6960	13	119
5	8192	13	128

Table 7.1: Parameters for CM

7.1.2 $K_{\mathbf{gen}}$

Key generation generates the public key P_k and the secret key S_k.

Definition 18. *Secret key S_k in CM KEM*

$$S_k = (s, \Gamma)$$

Where $\Gamma = (g, L)$, g is the goppa polynomial and $L = [\alpha_1, \alpha_2, \cdots, \alpha_n]$ is the support for the Goppa Code C, and s is a uniform random n-bit string.

Definition 19. *Public key P_k in CM KEM*

$$P_k = T$$

Where T is the part $H = [I_{n-k} \mid T]$, and H is the semi-systematic form of the parity check matrix H' of the Goppa Code C (see figures 7.2).

Key generation for a given set of CM parameters is done by:

1. Generate uniform random monic irreducible polynomial $g(x) \in \mathbb{F}_q[x]$ of degree t.
2. Select uniform random sequence $L = (\alpha_1, \alpha_2, \cdots, \alpha_n)$ of n distinct elements of $\mathbb{F}_q$.
3. Compute the $t \times n$ parity check matrix H' for the Goppa Code Γ defined by (g, L) (fig 7.2).
4. Form the $mt \times n$ matrix H^* by replacing each entry with a column of t bits where the bits correspond to the coefficient of the polynomial representation of the entry.
5. Reduce H^* to semi systematic form H (fig 7.3). If step fails return to step 1.
6. Generate uniform random n-bit string s.
7. Output $S_k = (s, \Gamma)$, $P_k = T$

$$H' = \begin{bmatrix} \frac{1}{g(\alpha_0)} & \frac{1}{g(\alpha_1)} & \cdots & \frac{1}{g(\alpha_{n-1})} \\ \frac{\alpha_0^1}{g(\alpha_0)} & \frac{\alpha_1^1}{g(\alpha_1)} & \cdots & \frac{\alpha_{n-1}^1}{g(\alpha_{n-1})} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\alpha_0^{t-1}}{g(\alpha_0)} & \frac{\alpha_1^{t-1}}{g(\alpha_1)} & \cdots & \frac{\alpha_{n-1}^{t-1}}{g(\alpha_{n-1})} \end{bmatrix}$$

Figure 7.2: Parity Check matrix for A Goppa Code

$$H = \left[I_{n-k} \mid T \right]$$

Figure 7.3: Parity check matrix reduced to semi-systematic form

7.1.3 $\mathcal{E}$

Given the the public key T and a vector e of length n and weight t, encoding [1] is done by:

1. Define $H = (I_{n-k} \mid T)$
2. Compute and return $C_0 = He \in \mathbb{F}_2^{n-k}$

7.1.4 H

In CM H is chosen to be Keccak 256 bit [50] but is interchangeable with any cryptographic hash function. H is used in encapsulation and decapsulation.

7.1.5 $\mathcal{D}$

Given the private key (Γ, s) and the ciphertext C_0.

1. Decode using $BM(C_0, \Gamma) = e$ Berlekamp-Massey (see Section 3.8.2)
2. output e if $h_{wt}(e) = t$, else $\perp$

7.1.6 Encapsulation

Encapsulation follows the SimpleKem proposed in [51]. The complete key encapsulation is done by:

1. Generate a uniform vector $e \in \mathbb{F}_2^n$ of weight t.
2. Encode $\mathcal{E}(T, e) = C_0$.
3. Compute $C_1 = H(2, e)$. Put $C = (C_0, C_1)$.
4. Compute $K = H(1, e, C)$
5. Output session key K and ciphertext C.

Figure 7.4: SimpleKem Encapsulation used in CM

[1] The term encoding is used instead of encryption because this is only part of the encapsulation process. Further the goal of the KEM is to encapsulate a key, and not to encrypt a message.

K is then used as the session key for the communication, and the sender sends the ciphertext C to the receiver.

7.1.7 Decapsulation

The reciever decapsulates the session key K from C by:

1. Split the ciphertext C as (C_0, C_1).
2. Set $b \leftarrow 1$.
3. Use $\mathcal{D}(\Gamma, C_0) = e$, if the subroutine returns $\perp$ then set $e \leftarrow s$ and $b \leftarrow 0$.
4. Compute $C_1' = H(2, e)$
5. If $C_1' \neq C_1$ set $e \leftarrow s$ and $b \leftarrow 0$.
6. Compute $K = H(b, e, C)$
7. Output session key K.

Figure 7.5: SimpleKem Decapsulation used in CM

CM is a full KEM (see Section 2.7). That is to say that it is not used to encrypt and decrypt messages. Instead, it is used to encapsulate a shared secret between the communicating parties.

CM uses the SimpleKem scheme [51], and a SHA-3 hash function to derive the key.

Simply put in CM the sender generates a random error vector e and calculates the syndrome $C_0 = [I_{n-k} \mid T]e$ using the pubic key T. They then compute the ciphertext C and session key K using the encapsulation scheme outlined in Figure 7.4.

The receiver then uses their private key to perform the decapsulation scheme outlined in Figure 7.5. If the decapsulation succeeds, then they will derive the same session key as the sender. If not, they will get a different key, and communication will not be possible.

Implicit rejection of invalid ciphertexts is essential as it does not allow an attacker to use the receiver as a decapsulation oracle. Explicit rejection can leak information about the encrypted message or key and can be utilized in an ISD attack [52]. Implicit rejection does not inform an attacker that the modified ciphertext they submitted was valid or not.

7.2 Performance

Performance of CM is measured in average clock cycles. During key generation, the public key will only be reducible to systematic form for 29% of random binary Goppa codes. That means that the system will reject 3-4 random Goppa polynomials on average, making the procedure restart.

To make the implementations more directly comparable, latency is calculated to be the time of each operation on a 3.5 GHz CPU.

Overall we see that the hardware implementation offers a substantial latency improvement in key generation.

Latency is worse for encapsulation and decapsulation because the linear algebra lends itself well to optimization with AVX in software, and the lower clock speed in hardware results in worse performance.

CM software performance

Parameters	keypair		Encapsulation		Decapsulation	
$n/m/t$	cycles	latency	cycles	latency	cycles	latency
3488/12/64	$5.8 \cdot 10^7$	14.51 ms	$4.4 \cdot 10^4$	0.01 ms	$1.3 \cdot 10^5$	0.03 ms
4608/13/96	$2.2 \cdot 10^8$	53.95 ms	$1.2 \cdot 10^5$	0.03 ms	$2.7 \cdot 10^5$	0.07 ms
6688/13/128	$5.6 \cdot 10^8$	140.00 ms	$1.5 \cdot 10^5$	0.04 ms	$3.2 \cdot 10^5$	0.08 ms
6960/13/119	$4.4 \cdot 10^8$	109.55 ms	$1.6 \cdot 10^5$	0.04 ms	$3.0 \cdot 10^5$	0.08 ms
8192/13/128	$5.1 \cdot 10^8$	127.50 ms	$1.8 \cdot 10^5$	0.05 ms	$3.2 \cdot 10^5$	0.08 ms

CM software performance - semi systematic form

Parameters	keypair		Encapsulation		Decapsulation	
$n/m/t$	cycles	latency	cycles	latency	cycles	latency
3488/12/64	$3.7 \cdot 10^7$	9.16 ms	$4.4 \cdot 10^4$	0.01 ms	$1.3 \cdot 10^5$	0.03 ms
4608/13/96	$1.2 \cdot 10^8$	29.27 ms	$1.2 \cdot 10^5$	0.03 ms	$2.7 \cdot 10^5$	0.07 ms
6960/13/119	$2.5 \cdot 10^8$	61.63 ms	$1.6 \cdot 10^5$	0.04 ms	$3.0 \cdot 10^5$	0.08 ms

Table 7.2: Perfomance of CM [3]

7.3 Classic McEliece Hardware Implementation

Classic McEliece Hardware Implementation (CMHW) is part of the CM [3] NIST PQC submission. Key generation is explained in [22] and the entire implementation in [46].

The implementation provides adjustable parameters that through a Make-File produces synthesizable Verilog code.

7.3.1 Dependencies

The hardware implementation of CM generates some of its modules from Sage-Math scripts. These scripts require the user to install a large ($\approx$ 5 GB) and specific version of the SageMath runtime environment.

7.3.2 Niederreiter

CMHW is not a full hardware implementation of the Niederreiter KEM, and is only the Niederreiter part of the CM cryptosystem.

The supported operations are:

1. $S_k, P_k \leftarrow K_{gen}$ (Section 7.1.2)
2. $c \leftarrow \mathcal{E}(P_k, e)$ (Section 7.1.3)
3. $e \leftarrow \mathcal{D}(S_k)$ (Section 7.1.7)

The parameters can be seen in Table 7.1.

7.3.3 Performance

The performance numbers from Table 7.3 are collected directly from [3]. The implementation allows the user to adjust parameters to optimize for either size or time, and the numbers shown are the time-optimized numbers.

Performance can be seen in Table 7.3, and footprint in Table 7.4.

BRAM modules are 36 kB each. Note that CM hardware implementation only performs encoding and decoding, not the entire encapsulation and decapsulation operations.

CM hardware performance

Parameters	Fmax	keypair		Encoding		Decoding	
n/m/t	MHz	cycles	latency	cycles	latency	cycles	latency
3488/12/64	104	$4.8 \cdot 10^5$	4.64 ms	$2.7 \cdot 10^3$	0.03 ms	$1.2 \cdot 10^4$	0.12 ms
4608/13/96	108	$1.4 \cdot 10^6$	12.81 ms	$3.4 \cdot 10^3$	0.03 ms	$1.9 \cdot 10^4$	0.17 ms
6688/13/128	143	$3.3 \cdot 10^6$	23.40 ms	$5.0 \cdot 10^3$	0.04 ms	$3.2 \cdot 10^4$	0.22 ms
6960/13/119	136	$3.1 \cdot 10^6$	22.69 ms	$5.4 \cdot 10^3$	0.04 ms	$2.7 \cdot 10^4$	0.20 ms
8192/13/128	131	$4.1 \cdot 10^6$	31.42 ms	$6.5 \cdot 10^3$	0.05 ms	$3.4 \cdot 10^4$	0.26 ms

Table 7.3: Perfomance of CMHW [46]

CM hardware sizes

Parameters		LUTs	FF	BRAM
3488/12/64	Keypair	24227	44016	192
	Encoding	2648	3351	0
	Decoding	12890	23084	20
	Total	39765	70451	212

Parameters		LUTs	FF	BRAM
4608/13/96	Keypair	34809	60637	315
	Encoding	3804	4617	0
	Decoding	18520	31800	33
	Total	57133	97054	348

Parameters		LUTs	FF	BRAM
6960/13/119	Keypair	38766	72210	460
	Encoding	4237	5498	0
	Decoding	20625	37870	48
	Total	63628	115578	508

Parameters		LUTs	FF	BRAM
6688/13/128	Keypair	40585	69536	444
	Encoding	4435	5295	0
	Decoding	21593	36467	47
	Total	66613	111298	491

Parameters		LUTs	FF	BRAM
8192/13/128	Keypair	41098	72360	516
	Encoding	4492	5510	0
	Decoding	21866	37948	55
	Total	67456	115818	571

Table 7.4: Footprint of the CM hardware implementation [3]

Chapter 8

BIKE

Bit flipping key encapsulation (BIKE) [6] is an alternate candidate to the NIST PQC. It was not selected to be a finalist in the NIST PQC because of the performance suffered in encoding to target IND-CCA (due to decoding failure rate) [45].

Block length r, row weight w and t correctable errors $\mathcal{R} = \mathbb{F}_2[X]/(X^r - 1)$,$(h_0, h_1) \in \mathcal{H}_w$ Private key space $\mathcal{H}_w = \{(h_0, h_1) \in \mathcal{R}^2 \mid h_{wt}(h_0) = h_{wt}(h_1) = w/2\}$ Error space $\mathcal{E}_t = \{(e_0, e_1) \in \mathcal{R}^2 \mid h_{wt}(e_0) + h_{wt}(e_1) = t\}$		
Key	S_k	P_k
	(h_0, h_1)	$h = h_1 h_0^{-1} \in \mathcal{R}$
Encode	Encode m as $(e_0, e_1) \in \mathcal{E}_t$ $c = e_0 + e_1 h \in \mathcal{R}$	
Decode	Decode$(sh_0, h_0, h_1) \rightarrow (e_0, e_1)$	

Table 8.1: BIKE QC-MDPC encoding and decoding

BIKE is built as a Niederreiter scheme using QC-MDPC codes. The system uses the Fujisaki-Okamoto transformation with implicit rejection [53], [54] to transform the PKE to a KEM.

8.1 Niederreiter KEM

BIKE is a KEM (Section 2.7) using the Niederreiter (Section 4.1) variant of McEliece. It uses Quasi-cyclic Moderate Density Parity Check (QC-MDPC) codes.

8.1.1 Parameters

BIKE has parameter choices for NIST security categories 1, 3, and 5. These paramter choices are for the QC-MDPC and are listed in Table 8.2.

NIST Category	r	w	t	DFR
1	12323	142	134	2^{-128}
3	24659	206	199	2^{-192}
5	40973	274	264	2^{-256}

Table 8.2: Parameters for BIKE

8.1.2 $K_{\mathbf{gen}}$

The private key consists of the tuple $(h_0, h_1) \in \mathcal{H}_w$ and a randomly chosen $\sigma \in \mathcal{M}$. Note that h_0 needs to be invertible in $\mathcal{R}$, and therefore $w/2$ must be odd. $\mathcal{M}$ is the message space

Keygen
Input parameters n, w, t
Output S_k, P_k

1. $(h_0, h_1) \xleftarrow{\$} \mathcal{H}_w$
2. $h = h_1 h_0 - 1$
3. $\sigma \xleftarrow{\$} \mathcal{M}$
4. output $S_k = (h_0, h_1, \sigma), P_k = h$

Figure 8.1: Key generation for BIKE

8.1.3 Encode

Encode
Input error vectors $(e_0, e_1) \in \mathcal{E}_t$, public key $P_k = h \in \mathcal{R}$
Output ciphertext c

1. $s = e_0 + e_1 h$
2. output c

Figure 8.2: Encoding subroutine for BIKE

8.1.4 Decode

Decode
Input syndrome c, secret key S_k
Output $e \in \mathcal{E}_t$

1. $(e_0, e_1) = \mathit{decoder}(sh_0, h_0, h_1)$
2. output (e_0, e_1) if $\in \mathcal{E}_t$ else $\perp$

Figure 8.3: Encoding subroutine for BIKE

Decoding Algorithm for BIKE

The decoder in the decoding subroutine 8.3 is the Black-Gray Bit flipping (BGF) algorithm described in [15].

The parity check matrix $H = (H_0 \mid H_1) \in \mathbb{Z}_2^{r \times 2r}$ is built from the circulant blocks H_0, H_1 derived from (h_0, h_1) respectively.

A Black iteration is a bit flipping (see Figure 3.4) that occurs only if the number of unsatisfied parity checks are greater or equal to th. In other words we assume $e_i = 1$ if $unsatifiedParityChecks = H_i s \geq th$

A Gray iteration is a bit flipping that occurs if the number of unsatisfied parity check are greater or equal than $th - \delta$, where δ is an integer. In other words we assume $e_i = 1$ if $unsatifiedParityChecks = H_i s \geq th - \delta$. For BIKE $\delta = 4$.

The decoding BGF algorithm is outlined in Figure 8.4, and it's decoding failure rate is listed in Table 8.2.

BGF Decoder
Input parity check matrix H and ciphertext $c = e_0 + e_1 h$
Output (e_0, e_1)

1. Initialize $e = \mathbf{0}$
2. For $i = 0...N$ iterations:
   (a) Calculate threshold for e, c, and H based on i
   (b) Update $e, black, gray$ from a normal bit flip
   (c) If $i = 0$, update e from black bitflip with threshold $(d+1)/2 + 1$
   (d) If $i = 0$, update e from gray bitflip with threshold $(d+1)/2 + 1$
3. output e if $c = He$ else $\perp$

Figure 8.4: Black-Gray-Flip decoding algorithm used in BIKE

8.1.5 Encapsulation

BIKE encapsulation require the functions:

$$\mathbf{H} : \mathcal{M} \to \mathcal{E}_t$$

$$\mathbf{K} : \mathcal{M} \times \mathcal{R} \times \mathcal{M} \to \mathcal{K}$$

Where $\mathcal{K}$ is the shared key space, $\mathcal{M}$ is the message space and $\mathcal{R} = \mathbb{Z}_2[X]/(X^r - 1)$.

The functions are modeled as random oracles. The encapsulation procedure is outlined in Figure 8.6

Note that because $\mathbf{H}$ needs to output a vector of length $n = 2r$ with weight t, we limit the input space $\mathcal{M}$ to

$$|\mathcal{M}| = \binom{n}{t}$$

8.1.6 Decapsulation

Decapsulation also requires $\mathbf{H}$, $\mathbf{K}$, and additionally $\mathbf{L} : \mathcal{R}^2 \to \mathcal{M}$.

Parameters $r/w/t$	$\lvert\mathcal{M}\rvert$
12323/142/134	2^{1068}
24659/206/199	2^{1671}
40973/274/264	2^{2302}

Figure 8.5: Maximal message space size for BIKE

Encapsulate
Input $h \in \mathcal{R}$
Output $K \in \mathcal{K}, c \in \mathcal{R} \times \mathcal{M}$
1. $m \leftarrow \mathcal{M}$
2. $(e_0, e_1) \leftarrow \mathbf{H}(m)$
3. $c \leftarrow (e_0 + e_1 h, m \oplus \mathbf{L}(e_0, e_1))$
4. $K \leftarrow \mathbf{K}(m, c)$
5. Output K, c

Figure 8.6: Encapsulation procedure in BIKE

Decapsulate
Input $S_k = (h_0, h_1, \sigma)$, $c \in \mathcal{R} \times \mathcal{M}$
Output $K \in \mathcal{K}$
1. $e' \leftarrow decoder(c_0 h_0, h_0, h_1)$
2. $m' \leftarrow c_1 \oplus \mathbf{L}(e')$
3. $K \leftarrow \mathbf{K}(m', c)$ if $e' = \mathbf{H}(m')$ else $\mathbf{K}(\sigma, c)$
4. Output K

Figure 8.7: Decapsulation procedure in BIKE

8.2 Performance

The performance of BIKE is listed in Table 8.3. The BIKE submission does not provide performance numbers for category 5 parameters.

Latency is calculated on a CPU with a clock speed of 3.5 GHz.

BIKE software performance

Parameters	keypair		Encapsulation		Decapsulation	
$r/w/t$	cycles	latency	cycles	latency	cycles	latency
12323/142/134	$6.0 \cdot 10^5$	0.15 ms	$2.2 \cdot 10^5$	0.06 ms	$2.2 \cdot 10^6$	0.56 ms
24659/206/199	$1.8 \cdot 10^6$	0.44 ms	$4.7 \cdot 10^5$	0.12 ms	$6.6 \cdot 10^6$	1.65 ms

Table 8.3: Perfomance of BIKE [6]

8.3 Hardware Implementation

BIKE also supplies a hardware implementation for category 1. The performance can be seen in Table 8.4, and the footprint is seen in Table 8.5.

BRAM modules are 36kB each.

The hardware implementation of BIKE offers no latency improvement over the software implementation.

BIKE hardware performance

Parameters	Fmax	keypair		Encapsulation		Decapsulation	
n/m/t	MHz	cycles	latency	cycles	latency	cycles	latency
	96	$2.6 \cdot 10^5$	2.70 ms				
12323/142/134	122			$1.2 \cdot 10^4$	0.10 ms		
	100					$1.9 \cdot 10^5$	1.90 ms

Table 8.4: Perfomance BIKE hardware implementation [6]

BIKE hardware sizes

Parameters		LUTs	FF	BRAM
1	Keypair	12654	1044	10
	Encryption	14894	3477	10
	Decryption	29908	5075	29
	Total	57456	9596	49

Table 8.5: Footprint of the BIKE implementation [6]

Chapter 9

HQC

Hamming Quasi-cyclic (HQC) [7] is an alternate candidate the NIST PQC.

It uses the Fujisaki-Okamoto transform to construct a KEM from an underlying PKE based on a concatenated code.

9.1 HQC KEM

9.1.1 Concatenated Codes

HQC uses a concatenated code, consisting of an internal Reed-Solomon (length n_1) code and an external Reed-Muller code (length n_2).

The underlying PKE can be seen in Table 9.1, and the parameters for the varying security levels in Table 9.2.

Code length n, dimension k, δ correctable errors w weight of x and y, w_r weight of r, and w_e weight of e $\mathcal{R} = \mathbb{F}_2[X]/(X^n - 1)$ $x, y, h, e \in \mathcal{R}$ Generator matrix $G \in \mathbb{F}_2^{n_1 n_2}$		
Key	S_k	P_k
	$(x, y) \xleftarrow{\$} \mathcal{R}^2$	$(h, s = x + h \cdot y)$
Encode	$e \xleftarrow{\$} \mathcal{R}$ $(r_1, r_2) \xleftarrow{\$} \mathcal{R}^2$ $u = r_1 + h \cdot r_2$ $v = mG + s \cdot r_2 + e$ Output $c = (u, v)$	
Decode	$m \leftarrow \text{Decode}(v - u \cdot y)$ Output m	

Table 9.1: HQC PKE

NIST Category	n_1	n_2	n	w	$w_r = w_e$	DFR
1	46	384	17,669	66	75	$< 2^{-128}$
3	56	640	35,851	100	114	$< 2^{-192}$
5	90	640	57,637	131	149	$< 2^{-256}$

Table 9.2: Parameters for the concatenated code used in HQC

NIST Category	S_k	P_k	ciphertext	session key
1	40	2249	4,481	64
3	40	4522	9,026	64
5	40	7245	14,469	64

Table 9.3: Sizees in bytes for HQC

9.1.2 Encoding

Because HQC has a concatenated code, encoding is done in two steps. First the message $m \in \mathbb{F}_2^k$ is encoded into $m_1 \in \mathbb{F}_{2^8}^{n_1}$ by the internal Reed Solomon code, and then each coordinate is encoded into $\tilde{m}_{1,i} \in \mathbb{F}_2^{n_2}$ so we end up with $mG = \tilde{m} = (\tilde{m}_{1,0}, ...\tilde{m}_{1,n_1-1}))$ to match the definition in Figure 9.1.

9.1.3 Decoding

Decoding is performed by decoding both codes individually. Because these codes require extensive preliminary explanations, the decoding details will be omitted.

The detailed decoding process can be found in [7, pp. 24–26].

9.1.4 Encapsulation

To perform encapsulation, HQC additionally needs hash functions $\mathbf{G}, \mathbf{H}$, and $\mathbf{K}$.

Encapsulate
Input P_k
Output $K \in \mathcal{K}, c \in \mathcal{R}^2 \times \mathcal{M}, d \in \mathbb{Z}_2^l$

1. $m \xleftarrow{\$} \mathcal{M}$
2. $\theta \leftarrow \mathbf{G}(m)$
3. $c \leftarrow PQC.PKE.Encode(P_k, m, \theta)$
4. $K \leftarrow \mathbf{K}(m, c)$
5. $d \leftarrow \mathbf{H}(m)$
6. Output $K, (c, d)$

Figure 9.1: Encapsulation procedure in HQC

9.1.5 Decapsulation

Decapsulation procedure can be seen in Figure 9.1.

Decapsulate
Input $S_k, P_k \in \mathcal{R}^2, (c, d) \in \mathcal{R}^2 \times \mathcal{M} \times \mathbb{Z}_2^l$
Output $K \in \mathcal{K}$

1. $m' \leftarrow PQC.PKE.Decode(S_k, c)$
2. $c' \leftarrow PQC.PKE.Encode(S_k, c)$
3. $K \leftarrow \mathbf{K}(m, c)$ if $m' = m$ and $\mathbf{H}(m') = d$ else $\perp$
4. Output K

Figure 9.2: Decapsulation procedure in HQC

9.2 Performance

The performance of the HQC software version can be seen in Table 9.4. Latency is calculated on a CPU with a clock speed of 3.5 GHz.

HQC software performance

NIST	keypair		Encryption		Decryption	
Category	cycles	latency	cycles	latency	cycles	latency
1	$1.4 \cdot 10^5$	0.03 ms	$2.2 \cdot 10^5$	0.06 ms	$3.8 \cdot 10^5$	0.10 ms
3	$3.0 \cdot 10^5$	0.08 ms	$5.0 \cdot 10^5$	0.13 ms	$8.2 \cdot 10^5$	0.21 ms
5	$5.5 \cdot 10^5$	0.14 ms	$9.2 \cdot 10^5$	0.23 ms	$1.5 \cdot 10^6$	0.38 ms

Table 9.4: Perfomance of HQC [7]

9.3 Hardware Implementation

HQC also supplies a hardware implementation for category 1. The performance can be seen in Table 9.5, and the footprint is seen in Table 9.6.

BRAM modules are 36kB each.

HQC hardware performance

NIST	Fmax	keypair		Encapsulation		Decapsulation	
Category	MHz	cycles	latency	cycles	latency	cycles	latency
1	180	$5.9 \cdot 10^4$	0.33 ms	$1.6 \cdot 10^5$	0.88 ms	$2.7 \cdot 10^5$	1.48 ms

Table 9.5: Perfomance HQC hardware implementation [7]

HQC hardware sizes

Category		LUTs	FF	BRAM
1	Keypair	12654	1044	10
	Encryption	14894	3477	10
	Decryption	29908	5075	29
	Total	57456	9596	49

Table 9.6: Footprint of the HQC implementation [7]

Part III

Conclusion and future work

Here we will look at possible improvements to the hardware implementations. It is very often the goal of a hardware implementation to be minimal, and any improvement that can be made can be made in either size or speed can make a difference in the hardware requirements.

Minimizing time $\times$ area is an obvious goal, but alternative designs might also have different use cases. Even if an alternative design might be slower, it might require less logic and therefore be able to run on a less expensive device.

This outlines the rationale to optimize time $\times$ area and investigate alternative designs.

Chapter 10

Comparison

Now that we have defined all the code based NIST candidates, we can accurately compare their performance.

10.1 Software Performance

Some of the implementations have C99, AVX-2, and AVX-512 optimized implementations. Because the newer AVX-512 is only available in very recent server-grade CPUs, the AVX-2 optimizations where preferred in this comparison as it is more widely available.

The schemes are ranked by the cost in cycles. The normalized column is cycles normalized to the smallest value in the category and relative cost is normalized to the largest.

Category 1 keypairs				
rank	scheme	cycles	normalized	relative cost
1	HQC	$1.4 \cdot 10^5$	1	$2.3 \cdot 10^{-3}$
2	BIKE	$6.0 \cdot 10^5$	4.41	0.01
3	CM_f	$3.7 \cdot 10^7$	269.42	0.63
4	CM	$5.8 \cdot 10^7$	426.72	1

Category 3 keypairs				
rank	scheme	cycles	normalized	relative cost
1	HQC	$3.0 \cdot 10^5$	1	$1.4 \cdot 10^{-3}$
2	BIKE	$1.8 \cdot 10^6$	5.84	$8.2 \cdot 10^{-3}$
3	CM_f	$1.2 \cdot 10^8$	383.83	0.54
4	CM	$2.2 \cdot 10^8$	707.49	1

Category 5 keypairs				
rank	scheme	cycles	normalized	relative cost
1	HQC	$5.5 \cdot 10^5$	1	$1.2 \cdot 10^{-3}$
2	CM_f	$2.5 \cdot 10^8$	452.31	0.56
3	CM	$4.4 \cdot 10^8$	804.07	1

Table 10.1: Key pair generation ranked by performance

For key pair generation, the alternative design in CM [3] is included. This design uses a semi systematic form of the public key generation, which gives it better performance in key generation. From Table 10.1 we can see that this method roughly cuts the cost of key generation by a third.

HQC and BIKE have a substantially lower cost for key generation than CM.

Category 1 Encapsulation				
rank	scheme	cycles	normalized	relative cost
1	CM	$4.4 \cdot 10^4$	1	0.20
2	BIKE	$2.2 \cdot 10^5$	4.96	1
2	HQC	$2.2 \cdot 10^5$	4.96	1

Category 3 Encapsulation				
rank	scheme	cycles	normalized	relative cost
1	CM	$1.2 \cdot 10^5$	1	0.24
2	BIKE	$4.7 \cdot 10^5$	3.95	0.93
3	HQC	$5.0 \cdot 10^5$	4.25	1

Category 5 Encapsulation				
rank	scheme	cycles	normalized	relative cost
1	CM	$1.6 \cdot 10^5$	1	0.18
2	HQC	$9.2 \cdot 10^5$	5.69	1

Table 10.2: Encapsulation ranked by performance

Encapsulation performance is listed in Table 10.2. Here we see that CM has lowest cost. BIKE and HQC appear to be quite similar, but BIKE loses some ground to HQC for category 3. Because BIKE did not supply performance numbers for category 5, we can not know if this performance delta grows for category 5.

Note that CM only performs Encoding and not the entire encapsulation procedure.

Finally, decapsulation performance is listed in Table 10.3. Here we see that CM has the lowest cost by a large margin, followed by HQC. BIKE has the highest cost of decapsulation, and we see that the performance delta grows for each category. CM appears to scale better with the higher categories, and the relative cost of CM reduces with an increase in category.

Overall, CM has a high cost in key generation but a very low cost of both encapsulation and decapsulation. HQC has an extremely low cost of key generation but suffers a bit in encapsulation.

10.2 Hardware Performance

The CM [46] hardware implementation supplies customizable parameters for all NIST categories. The CM numbers are collected from [3], but these numbers supply footprint numbers for the entire system. The numbers in Table 10.4 are collected by using this total footprint and multiplying with the ratio's from [46]

Category 1 Decapsulation				
rank	scheme	cycles	normalized	relative cost
1	CM	$1.3 \cdot 10^5$	1	0.06
2	HQC	$3.8 \cdot 10^5$	2.85	0.17
3	BIKE	$2.2 \cdot 10^6$	16.48	1

Category 3 Decapsulation				
rank	scheme	cycles	normalized	relative cost
1	CM	$2.7 \cdot 10^5$	1	0.04
2	HQC	$8.2 \cdot 10^5$	3.02	0.12
3	BIKE	$6.6 \cdot 10^6$	24.33	1

Category 5 Decapsulation				
rank	scheme	cycles	normalized	relative cost
1	CM	$3.0 \cdot 10^5$	1	0.20
2	HQC	$1.5 \cdot 10^6$	5.10	1

Table 10.3: Decapsulation ranked by performance

for the "balanced" configuration. This configuration is optimized for a balance between time and size.

BIKE and HQC only supply performance numbers for category 1 parameters. Because of this, the comparison is limited to category 1. It should be noted that the CM implementation is far more customizable than the other two, and that HQC has used HLS (Section 5.1.3) which greatly increases the maintainability of their code.

Performance rankings for category 1 security can be found in Table 10.4. Time×area measures:

$$\frac{\text{LUTs} \cdot \text{cycles}}{\text{clock frequency (MHz)}}$$

LUTs were chosen to represent logic as it is the building block of the logic in an FPGA design, and often the limiting factor. Both FF and BRAM should also be considered if they are a limiting factor.

10.3 Bandwidth

In this section, we will look at the bandwidth requirements of the submissions. In the general case, a key exchange must transfer the public key and also a ciphertext. Some use cases will benefit from small ciphertexts, and some might benefit from smaller keys.

The key sizes are listed in Table 10.5 where they are ranked by their bandwidth. BIKE has the smallest bandwidth requirements, while HQC requires more than double. CM has the most significant bandwidth requirements, but their very small ciphertext sizes might give it an advantage in other use cases.

Category 1 keypairs

rank	scheme	cycles	MHz	LUTs	FF	BRAM	time×area	normalized
1	HQC	59485	180	5174	2358	3	$1.7 \cdot 10^6$	1
2	BIKE	258750	96	12654	1044	10	$3.4 \cdot 10^7$	19.95
3	CM	482893	104	24227	44016	192	$1.1 \cdot 10^8$	65.79

Category 1 Encapsulation

rank	scheme	cycles	MHz	LUTs	FF	BRAM	time×area	normalized
1	CM	2720	104	2648	3351	0	$6.9 \cdot 10^4$	1
2	BIKE	12240	122	14894	3477	10	$1.5 \cdot 10^6$	21.58
3	HQC	158251	180	7766	4643	11	$6.8 \cdot 10^6$	98.59

Category 1 Decapsulation

rank	scheme	cycles	MHz	LUTs	FF	BRAM	time×area	normalized
1	CM	12036	104	12890	23084	20	$1.5 \cdot 10^6$	1
2	HQC	265836	180	11236	7836	12	$1.7 \cdot 10^7$	11.12
3	BIKE	189615	100	29908	5075	29	$5.7 \cdot 10^7$	38.02

Table 10.4: Hardware footprints and performance of the code based NIST PQC submissions. Ranked by time×area

Category 1 bandwidth

rank	scheme	P_k (B)	S_k (B)	ciphertext (B)	session key (B)	Bandwidth (kB)
1	BIKE	1540	280	1572	32	3.11
2	HQC	2249	40	4481	64	6.73
3	CM	261129	6492	128	32	261.26

Category 3 bandwidth

rank	scheme	P_k (B)	S_k (B)	ciphertext (B)	session key (B)	Bandwidth (kB)
1	BIKE	3082	418	3114	32	6.20
2	HQC	4522	40	9026	64	13.55
3	CM	524160	13608	188	32	524.35

Category 5 bandwidth

rank	scheme	P_k (B)	S_k (B)	ciphertext (B)	session key (B)	Bandwidth (kB)
1	BIKE	5121	580	5153	32	10.27
2	HQC	7245	40	14469	64	21.71
3	CM	1047319	13948	226	32	1047.55

Table 10.5: Ranked Bandwidth (public key + ciphertext) requirements of the code based NIST PQC submissions

10.4 FPGA Requirements

We have compared the implementations in terms of logic requirements, so let us now look at the minimum required FPGA board. We will consider the Artix®as potential boards.

The prices are collected from avnet.com, which is an authorized distributor of Xilinx FPGAs. They are also bulk prices, meaning that the price is per unit

for purchase of 40-60+ units. The lowest cost unit price was gathered for this comparison.

Category 1 requirements - keypair					**Utilization %**		
rank	scheme	Family	Board	price	LUTs	FF	BRAM
1	HQC	Artix-7	XC7A12T	$ 25.95	64.7	14.7	15.0
2	BIKE	Artix-7	XC7A25T	$ 33.60	86.7	3.6	22.2
3	CM	Artix-7	XC7A200T	$ 210.69	18.0	16.4	52.6

Category 1 requirements - Encapsulation					**Utilization %**		
rank	scheme	Family	Board	price	LUTs	FF	BRAM
1	CM	Artix-7	XC7A12T	$ 25.95	33.1	20.9	0.0
1	HQC	Artix-7	XC7A12T	$ 25.95	97.1	29.0	55.0
3	BIKE	Artix-7	XC7A35T	$ 34.68	71.6	8.4	20.0

Category 1 requirements - Decapsulation					**Utilization %**		
rank	scheme	Family	Board	price	LUTs	FF	BRAM
1	HQC	Artix-7	XC7A25T	$ 33.60	77.0	26.8	26.7
1	CM	Artix-7	XC7A25T	$ 33.60	88.3	79.1	44.4
3	BIKE	Artix-7	XC7A50T	$ 61.73	91.7	7.8	38.7

Table 10.6: Minimum FPGA requirements for the code based NIST PQC candidates per operation

From Table 10.6 we can see that CM key generation is very expensive, while BIKE and HQC are relatively cheap.

Encapsulation can be done on the same part for CM and HQC, while a slightly more expensive part is needed for BIKE. HQC has higher utilization than CM.

Decapsulation can also be done on the same part for CM and HQC, while BIKE requires a part of almost twice the cost. CM has higher utilization than HQC. Note that CM does only perform decoding and not the entire decapsulation procedure.

Category 1 requirements					**Utilization %**		
rank	scheme	Family	Board	price	LUTs	FF	BRAM
1	HQC	Artix-7	XC7A50T	$ 61.73	74.2	22.8	34.7
2	BIKE	Artix-7	XC7A100T	$ 126.84	90.6	7.6	36.3
3	CM	Artix-7	XC7A200T	$ 210.69	29.5	26.2	58.1

Category 3 requirements					**Utilization %**		
rank	scheme	Family	Board	price	LUTs	FF	BRAM
1	CM	Artix-7	XC7A200T	$ 210.69	42.4	36.1	95.3

Category 5 requirements					**Utilization %**		
rank	scheme	Family	Board	price	LUTs	FF	BRAM
1	CM	Virtex-7	XC7VX330T	$ 4478	31.2	28.3	67.7

Table 10.7: FPGA requirements for the code based NIST PQC candidates

From Table 10.7 we see that the increased logic cost of CM (primarily key generation) makes it require a substantially more expensive part to run. Due to the high BRAM requirements in category 5, CM can no longer run on an Artix®board but requires one from the larger Virtex®family. If the BRAM usage could be limited, it can quite easily fit on a cheaper board.

Overall, HQC can run the cheapest hardware, followed by BIKE.

10.5 Code-Based NIST PQC

Now that we have looked at all the code-based NIST PQC submissions, how did they compare? They all offer different performance profiles with advantages and disadvantages.

Based on the information presented in this chapter, Table 10.8 shows some of the highlights of each submission.

HQC

Advantage	Disadvantage
++ Low cost key generation	- High cost encapsulation
++ Runs on low cost hardware	- Low security maturity
+ Low bandwidth requirements	
+ Small public key	

BIKE

Advantage	Disadvantage
++ Low bandwidth requirements	- High cost decapsulation
++ Small public key	- Low security maturity
+ Low bandwidth requirements	
+ Runs on low cost hardware	

CM

Advantage	Disadvantage
++ Small ciphertext	-- Large public key
++ Low cost encoding	-- Large bandwidth requirements
++ Low cost decoding	-- Requires more expensive hardware
++ High security maturity	- not full hardware KEM

Table 10.8: Advantages/disadvantages of HQC

Chapter 11

Niderreiter with binary Goppa codes - Hardware

To find improvements to the Niederreiter scheme some components were isolated and rewritten. To produce known answer tests the CM software implementation was modified to produce public and secret keys, as well as error vector e and ciphertext c.

The goal of these improvements is to reduce time×space by either reducing run time or design footprint in the FPGA. FPGA utilization is counted in LUTs, and runtime is counted in clock cycles from simulation. In some cases, it is necessary to use optional metrics to compare two different designs accurately.

The produced components were simulated in ModelSim and synthesized in Vivado. They were not tested on hardware.

The reference implementation discussed in this chapter is the FPGA implementation described in [46].

11.1 Encryption

Encryption under a (n, m, t) Niederreiter scheme Π is the linear operation of $c = P_k e$ where e has weight t and length n, and where P_k (Section 4.1) is the public key. You can then split e into parts e_0 as the first mt bits of e, and e_1 as the last k bits.

$$P_k e = I_{mt} e_0 + T e_1 = e_0 + T e_1$$

Further e is a low density vector with t non-zero elements. This means that c will be a sum of t columns of P_k, and if this can be exploited this can save operations.

A possible way to do this is to represent the error vector as a list of indices $e' = [e_1, e_2, \cdots, e_t]$ where e_i represents a 1 in the ith position of the length n vector e. Having e on this form has two effects:

- The indices require $log_2(n) \cdot t$ bits to store, and for all possible parameter choices this means that $log_2(n) \cdot t < n$.
- Encrypting can be done by looking up t columns of the public key.

An implementation of this method can be found in the file

FPGA/src/encoder.vhd

The performance of this module can be seen in Table 11.1. The encoder has a register for columns of T and drives the address from another register. The memory module is used in testing is the same as the reference and can look up columns within a clock cycle. Because the encoder drives the address signal, there is a delay from setting the address and reading the column of 2 clock cycle cycles.

e is stored as a bit vector of $log(n) \cdot m$, which the encoder iterates over. The encoder sets the address to the value of the $log(n)$ chunks of e. If the value is less than $l = n - k$, then the bit in the given index is flipped, as this corresponds to the multiplication of the I portion of the public key $P_k = [I \mid T]$. This is completed in a single clock cycle.

This means that encoding is not guaranteed to be in constant time, as the delay incurred by a memory lookup in the last index of e would mean that encoding is completed in $121 - 122$ cycles based on the previous value. The encoder is pipelined, so it will always at most take 123 cycles, so it would be very easy to modify to always take this long.

Because of this invariability in encoding time breaks IND-CPA, and it can not be used as-is. A mitigation to this will be to account for the possible 2 or 1 cycle delay incurred from encoding.

Another side effect of this variability in encoding time is that it might be easier to determine the structure of e from DPA. If two error vectors e with differing amounts of bits in the mt- and k portions are encoded; the one with the most bits in the k portion would incur more memory lookups and more additions. This would likely be very visible in power consumption, and an adversary would be able to distinguish the two. This would also break IND-CPA.

	cycles	logic	time×logic
Reference [46]	5413	4276	$2.31 \cdot 10^7$
Encoder	123	4584	$5.6 \cdot 10^5$

Table 11.1: Performance of encoder

11.1.1 Generating e

In [3] the error vector e needs to be a uniform randomly generated vector of weight t. A possible way to do this is to randomly generate the t indices that are used in the alternate encoder.

Generating the alternate error vector $e' = e_1, e_2, \cdots e_t$ where $e_i \in [0, n-1]$, will then fail if $e_i = e_j$ for $i \neq j$.

The probability of generating a unique random list where $e_i \in [0, n-1]$ is enumerated in Table 11.2

Parameters n/m/t	Unique list probability %
3488/12/64	56
4608/13/96	37
6688/13/128	29
6960/13/119	36
8192/13/128	37

Table 11.2: Parameter choices probability of generating t unique values on the range $[0, n-1]$ by uniform random generation

11.2 Finite field operations

11.2.1 Multiplication GF(2^m)

The addition in $GF(2^m)$ is very simple. It maps to the binary xor operation for the representation of elements in $GF(2^m)$, and is very cheap and fast to implement in hardware. Multiplication, however, is not quite so simple and requires additional steps. The finite field is constructed with the polynomials $\mathbb{Z}_2[x]/h$ where h is an irreducible polynomial of degree m. Multiplying two elements g and h then becomes.

$$y = \sum_{i=0}^{m-1} f_i h$$

And because every entry is either 1 or 0, this maps to a sum of bit-shifted entries of h, which is straightforward to implement in hardware. To save on clock cycles, the reference implementation has implemented this multiplication as a combinatorial module. This means that multiplication of these base elements are completed within 1 clock cycle (as long as the total delay of the component allows it).

The way the authors achieved this is to generate the ideal logic from SageMath This poses an interesting question, as the produced HDL script is very minimal. It can, however, be implemented directly in a HDL, but how will this affect the synthesized module?

To find some insights into this question, the following experiment was constructed. The reference module was synthesized and compared to a HDL solution, and a fully generic HDL solution.

Having more generic modules is also useful as the reference implementation is written in an older version of SageMath, and getting it to run was not so straightforward. Further, it is easier to modify a solution written in a single language rather than multiple, and having a fully generic solution would be beneficial.

The results of these experiments can be seen in Table 11.3. The expectation was that the ideal logic produced by the SageMath script was to be the lowest cost in logic. We see, however, that the gf_mul.vhd has comparable but lower logic cost. To make sure this was not a function of Vivado preferring VHDL code, the Verilog reference (gf_mul_ref.v) was translated to VHDL (gf_mul_ref.vhd) and also synthesized.

	LUTs	SageMath dependant	h changeable
gf_mul_ref.v	90	yes	no
gf_mul_ref.vhd	90	yes	no
gf_mul.vhd	80	no	no
gf_mul_g.vhd	250	no	yes

Table 11.3: Estimated LUTs after synthesis for multiplication modules of elements in $GF(2^{13})$

The worst total delay of the implementation was 10.8 ns (4 ns logic delay, 6.8 net delay), which translates to roughly 92.6 MHz clock speed.

Further, we see that the generic module which can be instantiated with any number of m where the field polynomials h can be hot-swapped, the logic cost is quite a lot higher.

From this experiment, we get the sense that the synthesis tool is robust enough to be trusted when it comes to writing source code. It is not very obvious that creating ideal logic in the source file will lead to a smaller synthesized module.

11.2.2 Squaring in GF(2^m)

Squaring is a finite field is possible with a multiplication module, but due to the structure of a finite field $\mathbb{Z}_2[x]$, squaring has a very simple structure as all multiples of 2 are equivalent to 0. This leads to the general solution for squaring polynomials in $GF(2^m)$:

$$(x^{m-1} + x^{m-2} + \cdots + x + 1)^2 = (x^{2(m-1)} + x^{2(m-2)} + \cdots + x^2 + 1^2)$$

This general solution makes it possible to implement squaring in substantially lower logic. This is implemented in the reference as code generated by SageMath. To continue comparing with ideal generated logic and manually written components, the vhd equivalents were written (see Table 11.4).

	LUTs	SageMath dependant	generic
gf_sq_ref.v	7	yes	no
gf_sq.vhd	3	no	no
gf_sq_alt.vhd	7	no	no
gf_sq_hin.vhd	39	no	yes

Table 11.4: Estimated LUTs after synthesis for squaring modules of elements in $GF(2^{13})$

Some variations were tested to see their impact on space usage. First, gf_sq.vhd is the straightforward implementation where the field polynomial needs to be specified in the file, and all the powers of the polynomial is calculated.

The second is gf_sq_alt.vhd, where only the needed powers of the polynomial are calculated, as the odd powers are not needed when squaring. We can see that it made no difference in space usage.

The third is gf_sq_hin.vhd, where the powers of the field polynomial are fed to the component in a flat array. This makes the component generic and would allow swapping and computing powers of a new field polynomial from the outside.

The total delay of the gf_sq.vhd was 6.2 ns (3.4 logic and 2.8 net) which approximately translates to a clock speed of 161.3 MHz.

11.2.3 Inverting elements in GF(2^m)

In [22], inversion is done with a pre-computed Lookup table that is stored in BRAM (read-only). They do not compare their solution to an inversion module, so this is what we will be looking at here. Different implementations might have an excess of either BRAM or LUTs, so it would be advantageous to have a choice as to what resource to use. In order to compare a combinatorial circuit to the lookup Table, BRAM was disabled in synthesized, and the reference design was forced to use LUTs instead.

		LUTs	
Field Size	$GF(2^{12})$	$GF(2^{13})$	$GF(2^{14})$
Modulus	$(x^{12}+x^3+1)$	$(x^{13}+x^4+x^3+x+1)$	$(x^{14}+x^5+1)$
gf_inv_ref.v	1758	3776	8126
gf_inv.vhd	1118	1950	2314

Table 11.5: Estimated LUTs after synthesis for inversion modules of elements in $GF(2^{12-14})$

From the results of synthesis seen in Table 11.5 we can see that the implementation in pure combinatorial logic is smaller than the lookup method. Further, for generic fields, they will expect the lookup method to at least double in size for every increment in m.

We also observe that the combinatorial method's size relies on the complexity of the modulus, that is to say, that a smaller field polynomial will lead to less required logic. We see that there is a substantial increase when going from $GF(2^{12})$ to $GF(2^{13})$ as the modulus has more coefficients. Then we see a comparatively lesser increase in logic when going from $GF(2^{13})$ to $GF(2^{14})$, as this modulus has fewer coefficients.

Even though the combinatorial method is smaller in all cases, it is not obvious which to chose for $m = 12, 13$. Because the lookup method is implemented using dual port ram modules; they can potentially double the throughput of the modules.

The worst total delay of the gf_inv.vhd module is 43 ns (7.8 ns logic and 35.2 ns net) for $GF(2^{13})$. This translates to a clock speed of approximately 23.2 MHz. This means that with the use of BRAM in the lookup method, which can have an $F_{max} = 500MHz$, the maximal throughput of the lookup method is $2 \cdot 500 \cdot 10^6/s$, while the combinatorial method is $23.2 \cdot 10^6/s$.

Because the two components are so different in how they operate, we must devise some comparable metrics to gauge their relative performance. Table 11.6 we can see the estimated throughput of the modules (Inverses/s) and the latency, which will be the time from when you request the inverse to when you can read it. Because the reference uses synchronized memory lookup, the

memory address is set at one clock cycle and then read the next. This means that the value is not available for two clock cycles after the request.

Module	Area	Inverses/s	%	latency (ns)	%
reference (BRAM)	3.5	10^9	100	4.0	100
reference (LUTs)	3776	10^8	10	37.6	940
gf_inv.vhd (LUTs)	1950	$23.2 \cdot 10^6$	0.02	43.0	1075

Table 11.6: Performance comparisons for squaring modules in $GF(2^{13})$

So we have seen that a combinatorial inverse is possible and has a smaller footprint than the lookup method. The combinatorial method does, however, suffer in terms of performance. It has about 0.2‰ of the throughput and roughly ten times the latency.

This should give the observer an intuition that highly sequential operations cannot always be made to go faster in hardware. Hardware implementations can substantially speed up parallelizable problems, but when more logic is used to complete a longer sequence of operations it occurs substantial logic and net delays (the time the signal needs to propagate through the circuit).

The only case where the combinatorial method could have a use case is if BRAMs are not available, and the throughput is not an issue. In this case, the relative latency $\times$ area of the combinatorial method is better at $\frac{3776 \cdot 37.6}{1950 \cdot 43} = 1.7$. It is also improbable that a design will fully utilize the maximal throughput of the lookup method.

So to conclude, the lookup method with the use of BRAM outperforms the other solutions by a great margin in all cases. But the the theoretical max speed will not be achievable in most cases, and in the performance numbers from Table 7.3 we see that the max clock speed is about 100 MHz. This means that the throughput in a real-world scenario would be divided by 5. It is also likely that the net delay will be shorter should the combinatorial method is used in an actual design. When simulating singular components, the net delay is artificially high as it will count the path from a physical input pin, through the component, and then to a physical output pin.

On a side note, the combinatorial method is essentially a square and multiply for elements in $GF(2^{13})$. By modifying the module to have a target exponent, it was made a general solution in gf_sq_mul.vhd, and the footprint can be seen in Table 11.4.

		LUTs	
Field Size	$GF(2^{12})$	$GF(2^{13})$	$GF(2^{14})$
Modulus	$(x^{12} + x^3 + 1)$	$(x^{13} + x^4 + x^3 + x + 1)$	$(x^{14} + x^5 + 1)$
gf_inv.vhd	1118	1950	2314
gf_sq_mul.vhd	1178	2593	2386

Table 11.7: Estimated LUTs after synthesis for inversion modules of elements in $GF(2^{12-14})$

Chapter 12

Conclusion

12.1 Future work

12.1.1 Alternate Support Generation

CM generates the Goppa support by sorting a randomized list. They utilize a merge sorting algorithm. The basic idea is that a list of n sorted elements in the underlying $GF(2^m)$ are appended with a random 32-bit integer. The list is then sorted by comparing this 32-bit integer.

The merge sorting algorithm used has a complexity of $\mathcal{O}(n \log(n))$. By using some type of radix sort, this could potentially be improved, as radix has a complexity of $\mathcal{O}(n)$. Radix does, however, introduce substantial overhead, and this will most likely suffer a performance loss.

If, for instance, we use a radix sort with lists, where we select the number of lists to be 4, we would end up having 5 total registers with the size $(m+32) \cdot n$. This is up from the 2 registers used by the merge sort. We could then sort the list in 8 iterations, which is lower than $\log(n) = 13$. Further, Each iteration would require us to move the value into a list and back into the sorted list, so the number of operations is at least a factor of 2.

From this, we could expect a radix sort module to be $(8 \cdot 2)/13 = 1.2$ the speed of a merge sort, and $5/2 = 2.5$ of logic required.

12.1.2 Hardware Optimization for BIKE and HQC

We have looked at the hardware implementation of CM in detail, and found a highly adjustable and comprehensive solution. BIKE and HQC has, however, not been examined, and they give the impression that they are not as developed as the CM.

More substantial improvements can likely be found when examining HQC in particular, as they have made a hardware implementation using HLS (Section 5.1.3). This makes the implementation far more easily maintained, but in many cases it does not produce an optimal solution in terms of performance.

12.1.3 AVX Optimizations

The AVX instruction set can offer substantial speed-ups when performing linear operations. All the submissions already target AVX for speed improvements, but the design room for AVX improvements have not been looked at in this work.

Newer CPUs will have AVX512, which double the size of the registers. This leaves a massive potential to bit-slice operations, and it is very likely to find improvements if looked into.

12.1.4 BRAM Tuning of CM

In Table 10.7 we see that for category 5 security, CM needed a Virtex®board due to its high BRAM usage. By tuning the implementation for these parameters with the intent to bring BRAM usage down, we could potentially drastically reduce the part cost for category 5 security.

This could be achieved by looking at alternative ways of inversion as in Section 11.2.3, or by using an alternative ' polynomial evaluation method. The high BRAM usage comes primarily from the additive FFT used for polynomial evaluation, which is used in both key generation and decryption.

12.1.5 ARM Compatibility

The performance of the systems looked at in this work (BIKE, CM, and HQC) all have AVX optimized implementations that offer substantial performance improvements.

However, the AVX instruction set is limited to newer CPUs with a relatively high operating cost in terms of power. Lower powered CPUs like ARM would be interesting to target to see if how feasible code-based PQC is on a more limited platform. These will have more limited resources and a more limited instruction set.

Progress in this field has already been achieved in [55], where the authors managed to get key generation for category 1 security to run on the ARM Cortex-M4 platform.

For context, the numbers in Table 12.1 show the performance. The latency for key generation is about 570 times the software implementation, encapsulation 350 times, and decapsulation 530 times.

Category 1 CM on the ARM Cortex M4

Clock	Keygen		Encapsulation		Decapsulation	
Mhz	cycles	latency	cycles	latency	cycles	latency
168	$1.4 \cdot 10^9$	8.3 s	$5.8 \cdot 10^5$	3.5 ms	$2.7 \cdot 10^6$	16.1 ms

Table 12.1: CM (semi systematic) performance achieved by [55] for category 1 security on the ARM Cortex M4

The work also offers many space optimization algorithms that allow it to run on a limited platform. Some of these might also offer insights into reducing resource consumption in other areas.

12.2 Closing

The work done by the NIST PQC is very important. We have now seen that we have multiple viable systems for post-quantum cryptography, with NIST commenting that CM might be ready to standardize at the end of round 3.

Although quantum computers are not quite a threat yet, we already have robust cryptographic schemes that are ready to be adopted. The systems do, however, widely differ in their performance and bandwidth profiles. Within the code-based systems, we have seen the extremely small ciphertexts and huge keys of CM, and the very small keys and large ciphertexts of HQC. Depending on requirements this leaves any implementor with a fair deal of choice.

The current status of the NIST PQC suggests that Classic McEliece will be standardized. The bandwidth requirements of CM might not be an issue when communicating over the internet with large bandwidths and high power computers, but there are many situations where low-cost hardware needs to communicate with limited bandwidth with high security. In situations like this, CM is not suitable.

We saw that CM needed 8.3 seconds to generate a key on an ARM Cortex M4, and because of the logic cost, it could not fit the platform used in [55] for category 2 parameters. If we cannot generate keys on the devices and manually load the keys into the devices. This means that it probably would be a a lot easier to use symmetrical keys and have some central key distribution service which keeps track of them, which would bring us back to square one.

For these reasons, it appears very unlikely that we will see CM used in situations with limited bandwidth or low-cost hardware.

Appendices

Appendix A

Mathematical background

A.1 Finite Field

A *finite field* is a commutative division ring with a finite number of elements. A ring is a set of values with addition and multiplication. Commutative means that the order of operations is arbitrary specifically for multiplication, i.e., that $ab = ba$. And finally, a *division ring* means that division is possible for all elements in the set.

Performing division in a finite field requires some special attention. Any element a in a finite field will have an inverse a^{-1}. The *inverse* is another element in the finite field where $aa^{-1} = e$, where e is the identity element of the multiplicative group (usually 1). In a finite field, all elements have inverses, and the inverse is unique.

The easiest way of constructing a finite field is to look at the natural numbers modulo a prime number. These groups are represented as:

$$\mathbb{Z}_p, \ p \text{ is a prime number.}$$

If we look at the elements of $\mathbb{Z}_5$, we can see that a finite field can be constructed, as addition, multiplication and division are defined for all elements in the set as seen in table A.1. Further, we can see from the multiplication table that every element has an inverse, and dividing by a number is the same as multiplying with the inverse of that number. Therefore the elements of $\mathbb{Z}_5$ define a finite field.

$+_5$	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	0
2	2	3	4	0	1
3	3	4	0	1	2
4	4	0	1	2	3

$\times_5$	1	2	3	4
1	1	2	3	4
2	2	4	1	3
3	3	1	4	2
4	4	3	2	1

Table A.1: Addition and multiplication of elements in $\mathbb{Z}_3$

A.2 Galois Field

All finite fields will have a prime- or prime power number of elements. A *Galois field* is a finite field defined by the number of elements in the field. The notation for a Galois field is:

$$GF(p^n),\ p \text{ is prime and } n \text{ is a natural number.}$$

A.3 Binary Arithmetic

A very important finite field is the finite field with only two elements: $GF(2)$ (see table A.2). This field is used to represent the behavior of single bits.

$+_2$	0	1
0	0	1
1	1	0

$\times_2$	1
1	1

Table A.2: Addition and multiplication in $GF(2)$

A.4 Polynomials

Polynomials are expressions consisting of variables and coefficients. We will look at polynomials with a single variable, and these polynomials will have the form:

$$\sum_{i=0}^{n} c_i x^i = c_i x^n + c_{i-1} x^{n-1} + \cdots + c_1 x^1 + c_0$$

Where c_i are coefficients and x^i is the variable.

A.5 Irreducible Polynomials

An irreducible polynomial is a polynomial that can not be expressed as the product of lesser polynomials. So if we have an irreducible polynomial $f(x)$, there are no. Polynomial's $g(x)$ and $h(x)$ such that:

$$g(x)h(x) = f(x)$$

It can be helpful to think of an irreducible polynomial as the equivalent to a prime number in the set of polynomials.

A.6 Polynomial Reduction

Like taking the modulo of a natural number, one can reduce a polynomial $g(x)$ by a given polynomial $f(x)$. If we have $g(x) = x^6 + x^2 + x + 1$, and $f(x) = x^4 + x + 1$, then:

$$\begin{aligned} g(x) &= x^2(x^4) + x^2 + x + 1 \\ &\equiv_{f(x)} x^2(-x-1) + x^2 + 1 \\ &\equiv_{f(x)} -x^3 + 1 \end{aligned}$$

Notice that because $x^4 + x + 1 \equiv_{f(x)} 0$, we substitute $x^4 \equiv_{f(x)} = -x - 1$.

A.7 Polynomial Fields

Just like constructing fields with prime numbers, Polynomial fields can be constructed by using irreducible polynomials. To create a polynomial field, we can take a field F and extend it to the polynomial space by:

$$F[x]/f(x)$$

Here $F[x]$ are the polynomials with coefficients in F, and the field $F[x]/f(x)$ define a *ring* where are the elements are reduced by $f(x)$. Iff $f(x)$ is irreducible than this polynomial ring is a field. Notice that coefficients of the polynomial must also be a field.

A.8 Polynomial Addition

To add two polynomials, the coefficients are added together.

$$f(x) + g(x) = \sum_{i=0}^{n} (f_i + g_i)x_i$$

Where f_i are the coefficients of $f(x)$ and g_i the coefficients of $g(x)$. n is the highest degree of $g(x)$ or $f(x)$.

If a polynomial has coefficients in $GF(2)$ it is very simple to do addition in hardware. The polynomials can be represented as binary strings, and addition is the same as the xor operation, which is extremely cheap when implementing this in hardware.

A.9 Polynomial Multiplication

Multiplication between polynomials is done by multiplying the coefficients and variables.

$$f(x)g(x) = \sum_{i=0}^{n} f_i x_i g(x)$$

Where f_i are the coefficients of $f(x)$. This can also be simplified when dealing with coefficients in $GF(2)$, as the summation is very cheap. There are other ways of implementing a multiplication algorithm (like Karatsuba) that trade multiplications for additional additions. These methods perform favorably as addition is cheaper in hardware implementations.

A.10 Polynomial Inversion

A very important operation is the ability to find the inverse of a polynomial in a polynomial field, as multiplying with the inverse of a polynomial is the same as dividing.

The inverse is defined as:

$$f(x)f(x)^{-1} = 1$$

References

[1] R. De Wolf, "Quantum computing: Lecture notes," *arXiv preprint arXiv:1907.09415*, 2019.

[2] NIST. (). "Nist," [Online]. Available: https://www.nist.gov/about-nist. (accessed: 2020.06.16).

[3] D. J. Bernstein, T. Chou, T. Lange, I. von Maurich, R. Misoczki, R. Niederhagen, E. Persichetti, C. Peters, P. Schwabe, N. Sendrier, *et al.*, "Classic mceliece: Conservative code-based cryptography," *NIST submissions*, 2020.

[4] R. J. McEliece, "A public-key cryptosystem based on algebraic," *Coding Thv*, vol. 4244, pp. 114–116, 1978.

[5] J. Daemen and V. Rijmen, "Aes proposal: Rijndael," 1999.

[6] N. Aragon, P. S. Barreto, S. Bettaieb, F. Worldline, L. Bidoux, O. Blazy, P. Gaborit, T. Güneysu, C. A. Melchor, R. Misoczki, *et al.*, "Bike: Bit flipping key encapsulation,"

[7] C. A. Melchor, N. Aragon, S. Bettaieb, L. Bidoux, O. Blazy, J.-C. Deneuville, P. Gaborit, E. Persichetti, G. Zémor, and I.-C. Bourges, "Hamming quasi-cyclic (hqc)," *NIST PQC Round*, vol. 2, pp. 4–13, 2018.

[8] P. W. Shor, "Algorithms for quantum computation: Discrete logarithms and factoring," in *Proceedings 35th Annual Symposium on Foundations of Computer Science*, 1994, pp. 124–134.

[9] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM review*, vol. 41, no. 2, pp. 303–332, 1999.

[10] L. K. Grover, "A fast quantum mechanical algorithm for database search," in *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, 1996, pp. 212–219.

[11] D. J. Bernstein, N. Heninger, P. Lou, and L. Valenta, "Post-quantum rsa," in *International Workshop on Post-Quantum Cryptography*, Springer, 2017, pp. 311–329.

[12] D. Boneh and V. Shoup, "A graduate course in applied cryptography," *Draft 0.5*, 2020.

[13] E. Berlekamp, R. McEliece, and H. Van Tilborg, "On the inherent intractability of certain coding problems (corresp.)," *IEEE Transactions on Information Theory*, vol. 24, no. 3, pp. 384–386, 1978.

[14] R. Gallager, "Low-density parity-check codes," *IRE Transactions on information theory*, vol. 8, no. 1, pp. 21–28, 1962.

[15] N. Drucker, S. Gueron, and D. Kostic, "Qc-mdpc decoders with several shades of gray," in *International Conference on Post-Quantum Cryptography*, Springer, 2020, pp. 35–50.

[16] N. Patterson, "The algebraic decoding of goppa codes," *IEEE Transactions on Information Theory*, vol. 21, no. 2, pp. 203–207, 1975.

[17] A. Shoufan, F. Strenzke, H. G. Molter, and M. Stöttinger, "A timing attack against patterson algorithm in the mceliece pkc," in *International Conference on Information Security and Cryptology*, Springer, 2009, pp. 161–175.

[18] S. Heyse and T. Güneysu, "Code-based cryptography on reconfigurable hardware: Tweaking niederreiter encryption for performance," *Journal of Cryptographic Engineering*, vol. 3, no. 1, pp. 29–43, 2013.

[19] D. J. Bernstein, T. Lange, and C. Peters, "Attacking and defending the mceliece cryptosystem," in *International Workshop on Post-Quantum Cryptography*, Springer, 2008, pp. 31–46.

[20] D. J. Bernstein, "List decoding for binary goppa codes," in *International Conference on Coding and Cryptology*, Springer, 2011, pp. 62–80.

[21] H. Niederreiter, "Knapsack-type cryptosystems and algebraic coding theory," in *Problems of Control and Information Theory 15*, 1986.

[22] W. Wang, J. Szefer, and R. Niederhagen, "Fpga-based key generator for the niederreiter cryptosystem using binary goppa codes," in *International Conference on Cryptographic Hardware and Embedded Systems*, Springer, 2017, pp. 253–274.

[23] J. VOGEL, A. KRAVTCHENKO, and F. ROMINGER, *Error correction with a cross-interleaved reed-solomon code, particularly for cd-rom*, eng ; fre ; ger, 2001.

[24] K. Kato and S. Choomchuay, "An analysis of time domain reed solomon decoder with fpga implementation," *IEICE TRANSACTIONS on Information and Systems*, vol. 100, no. 12, pp. 2953–2961, 2017.

[25] M. Zhao, C. Hu, F. Wei, K. Wang, C. Wang, and Y. Jiang, "Real-time underwater image recognition with fpga embedded system for convolutional neural network," *Sensors*, vol. 19, no. 2, p. 350, 2019.

[26] IEEE Computer Society and the IEEE Standards Association Corporate Advisory Group, *Ieee standard for systemverilog— unified hardware design, specification, and verification language*, 2017.

[27] IEEE Computer Society, *Ieee standard vhdl language reference manual*, 2009.

[28] K. Gaj, *Implementation and benchmarking of round 2 candidates in the nist post-quantum cryptography standardization process using fpgas*, Lecture, 2020.

[29] Thomas Pöppelmann et al., *Newhope*, 2019.

[30] R. Avanzi, J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé, "Crystals-kyber," *NIST, Tech. Rep*, 2017.

[31] E. Prange, "The use of information sets in decoding cyclic codes," *IRE Transactions on Information Theory*, vol. 8, no. 5, pp. 5–9, 1962.

[32] J. Stern, "A method for finding codewords of small weight," in *International Colloquium on Coding Theory and Applications*, Springer, 1988, pp. 106–113.

[33] D. J. Bernstein, T. Chou, T. Lange, I. von Maurich, R. Misoczki, R. Niederhagen, E. Persichetti, C. Peters, P. Schwabe, N. Sendrier, *et al.*, "Classic mceliece: Conservative code-based cryptography," *NIST submissions*, 2019.

[34] L. Both and A. May, "Decoding linear codes with high error rate and its impact for lpn security," in *International Conference on Post-Quantum Cryptography*, Springer, 2018, pp. 25–46.

[35] N. Lahr, R. Niederhagen, R. Petri, and S. Samardjiska, "Side channel information set decoding using iterative chunking," in *International Conference on the Theory and Application of Cryptology and Information Security*, Springer, 2020, pp. 881–910.

[36] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, *et al.*, "Meltdown: Reading kernel memory from user space," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 973–990.

[37] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, *et al.*, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2019, pp. 1–19.

[38] C. Goodman, "An introduction to tempest," 2021.

[39] P. De Meulemeester, B. Scheers, and G. A. Vandenbosch, "Differential signaling compromises video information security through am and fm leakage emissions," *IEEE Transactions on Electromagnetic Compatibility*, vol. 62, no. 6, pp. 2376–2385, 2020.

[40] ——, "A quantitative approach to eavesdrop video display systems exploiting multiple electromagnetic leakage channels," *IEEE Transactions on Electromagnetic Compatibility*, vol. 62, no. 3, pp. 663–672, 2019.

[41] N. Zhang, Y. Lu, Q. Cui, and Y. Wang, "Investigation of unintentional video emanations from a vga connector in the desktop computers," *IEEE Transactions on Electromagnetic Compatibility*, vol. 59, no. 6, pp. 1826–1834, 2017.

[42] I. Kubiak, "Tempest font protects text data against rf electromagnetic attack," *Tehnički vjesnik*, vol. 27, no. 4, pp. 1058–1065, 2020.

[43] Altera, "Anti-tamper capabilities in fpga designs," 2008.

[44] Xillinx, "Developing tamper resistant designs with xilinx virtex-6 and 7 series fpgas," 2018.

[45] G. Alagic, J. Alperin-Sheriff, D. Apon, D. Cooper, Q. Dang, J. Kelsey, Y.-K. Liu, C. Miller, D. Moody, R. Peralta, *et al.*, "Status report on the second round of the nist post-quantum cryptography standardization process," *US Department of Commerce, NIST*, 2020.

[46] W. Wang, J. Szefer, and R. Niederhagen, "Fpga-based niederreiter cryptosystem using binary goppa codes," in *International Conference on Post-Quantum Cryptography*, Springer, 2018, pp. 77–98.

[47] D. J. Bernstein, T. Chou, T. Lange, I. von Maurich, R. Misoczki, R. Niederhagen, E. Persichetti, C. Peters, P. Schwabe, N. Sendrier, *et al.*, "Classic mceliece: Conservative code-based cryptography," *NIST submissions*, 2017.

[48] T. Chou, "Mcbits revisited," in *International Conference on Cryptographic Hardware and Embedded Systems*, Springer, 2017, pp. 213–231.

[49] D. J. Bernstein, T. Chou, and P. Schwabe, "Mcbits: Fast constant-time code-based cryptography," in *International Conference on Cryptographic Hardware and Embedded Systems*, Springer, 2013, pp. 250–272.

[50] Q. H. Dang, "Secure hash standard," 2015.

[51] D. J. Bernstein and E. Persichetti, "Towards kem unification.," *IACR Cryptol. ePrint Arch.*, vol. 2018, p. 526, 2018.

[52] N. Lahr, R. Niederhagen, R. Petri, and S. Samardjiska, *Side channel information set decoding using iterative chunking*, Cryptology ePrint Archive, Report 2019/1459, https://eprint.iacr.org/2019/1459, 2019.

[53] N. Drucker, S. Gueron, D. Kostic, and E. Persichetti, "On the applicability of the fujisaki-okamoto transformation to the bike kem.," *IACR Cryptol. ePrint Arch.*, vol. 2020, p. 510, 2020.

[54] D. Hofheinz, K. Hövelmanns, and E. Kiltz, "A modular analysis of the fujisaki-okamoto transformation," in *Theory of Cryptography Conference*, Springer, 2017, pp. 341–371.

[55] M.-S. Chen and T. Chou, "Classic mceliece on the arm cortex-m4,"