The Side-Channel Resistance of Error Correcting Codes for Post Quantum Cryptography

KARL SOUTHERN

The Side-Channel Resistance of Error Correcting Codes for Post Quantum Cryptography

Karl Southern

A Thesis presented for the degree of
Doctor of Philosophy

Department of Computer Science
Durham University
United Kingdom
April 2023

Declaration

The work in this thesis is based on research carried out at the Department of Computer Science, Durham University, United Kingdom. No part of this thesis has been submitted elsewhere for any other degree or qualification and it is all my own work unless referenced to the contrary in the text.

Acknowledgements

Whilst the writing of a PhD thesis is often an isolating and solitary experience, it is the culmination of the support and effort of many people.

Firstly I would like to thank my PhD advisor, Dr Maximilien Gadouleau. I am incredibly grateful for his support throughout the PhD and my wider academic journey. His guidance and feedback has been invaluable throughout. I will now never again try to write a paper without first setting up LaTeX macros for every piece of notation I use to force consistency as I inevitably change my mind on what symbol to use halfway through. I would also like to thank all of the academic and admin staff in the Computer Science department at Durham, for their help, advice, encouragement and support during both my PhD and my undergraduate study here in Durham.

Large portions of the time spent on the PhD took place during various stages of lockdowns and restrictions during Covid, where it felt like the only thing keeping me sane was virtual meetups with friends. With that it mind I'd like to thank my school friends, Tolkien Society friends and Durham friends, all of whom helped keep me entertained and motivated throughout, and who have acted as a reminder that there is more to life than just academia. A special thank you to Siani, my 'someone-who-lives-here', who has supported me throughout, acted as a rubber duck, ensured I actually take breaks, reminded me to start writing the thesis and, more importantly, reminded me that I have to stop writing it!

Finally a huge thank you to my family for their support and encouragement from day 0 (and day 731). Who would have thought that a treasure hunt formed of encoded messages when I was little would have ended here!

Contents

List of Figures

List of Tables

Acronyms

t-NI non-interference. 24–26, 28

t-SNI strong non-interference. 24–26

AES the Advanced Encryption Standard. 2, 4, 15

AWGN Additive White Gaussian Noise. 114–116

BSC Binary Symmetric Channel. 114

CCA chosen ciphertext attacks. 13, 14, 82

CCA2 adaptive chosen ciphertext attacks. 13

CPA chosen plaintext attacks. 12, 13, 82

ETSI European Telecommunications Standards Institute. 2

FO Fujusaki-Okamoto. 15

ISO the International Organization for Standardization. 2

KEM Key Encapsulation Mechanism. vi, 11, 15, 16, 29, 113, 114, 116

LWE Learning with Errors. v, 16, 19–21

LWR Learning with Rounding. v, 20

NIST US National Institute of Standards and Technology. 2, 4, 14–16, 44

PKC public key cryptography. iv, 1–4, 6, 10–15

PKE public key encryption scheme. 10, 15, 16

PQC post quantum cryptography. 3, 4, 7, 16, 44, 127, 128

R-LWE Ring Learning with Errors. v, 18, 36

R-LWR Ring Learning with Rounding. 20

SC side-channel. iv, v, 4, 6, 23–28, 94, 113

SKC secret key cryptography. 1–3, 11

CHAPTER 1

Introduction

1.1 What is cryptography?

For as long as humans have been writing, they have been trying to hide their message. From messages about troop movements between generals, recipes for making a pottery glaze, even the 1700 year old Kama Sutra has a guide on writing secret messages.

Over the course of millennia the methods for obscuring these messages, and for breaking them, have become more sophisticated and their usage has become more widespread. In the present day people use cryptography constantly, often without knowing. Every time someone connects to the internet or sends a WhatsApp message, they use a wide range of cryptographic protocols.

The majority of algorithms for encryption can be split into two broad categories: public key cryptography (PKC) and secret key cryptography (SKC). These are sometimes referred to as asymmetric cryptography and symmetric cryptography respectively. Secret key cryptography is the oldest form of cryptography and is any form of encryption method where the same key is used for both encryption and decryption. This form of encryption is the one most people will be familiar with and

some famous examples include the Caesar cipher and the Engima cipher. Public key cryptography is where different keys are used for encryption and decryption; typically the public key is used to encrypt the message and the secret key is used to decrypt the message. Comparatively public key cryptography is much more modern, only coming into existence in the latter half of the 20th century, with the most famous examples including RSA and Diffie-Hellman. A lot of public key cryptosystems are based around mathematical problems that are computationally difficult, such as the integer factorisation problem or the discrete logarithm problem.

Historically cryptographic techniques were kept secret and it is only with the importance of interconnectivity that these techniques have become more public. To ensure that devices are using the same cryptographic techniques, and to ensure that the wider cryptographic community is confident in their security, there are a number of ongoing processes to standardise proposed post quantum cryptographic protocols. US National Institute of Standards and Technology (NIST), European Telecommunications Standards Institute (ETSI) , and the International Organization for Standardization (ISO) are currently leading individual standardisation processes.

There are a number of standards for public key cryptography, including RSA and ECDH (Elliptic Curve Diffie-Hellman), and secret key cryptography, including the Advanced Encryption Standard (AES) and ChaCha. For modern applications public key cryptography is used to exchange a secret key between parties who will then both use that key to encrypt a large amount of text using secret key cryptography. All of these schemes have faced a large amount of scrutiny by the cryptographic community to ensure that they are secure against attacks by adversaries with large amounts of computational power.

1.2 Post quantum cryptography

The first ideas behind quantum computers were proposed in the 1980's, suggesting that it might be possible to use quantum interactions as a form of computation. It was believed that these quantum computers would be able to solve problems quicker than classical computers and even solve problems that classical computers are unable

to solve. It has since been shown that quantum computers can solve some problems asymptotically faster than classical computers.

In 1994 Peter Shor developed an algorithm that uses a quantum computer to find the prime factors of a given integer much faster than any existing classical algorithm. This was followed by Grover's algorithm in 1996, which can be considered as speeding up the time required to manually search for an input to a function that gives a specific value. The first quantum computer wasn't built until 1998, and even then it was only 2 qubits big.

Shor's algorithm has a very large impact on, and effectively breaks, the majority of modern public key cryptography, since most major public key cryptosystems rely on the difficulty of finding the prime factors of an integer, or the discrete logarithm problem. However it has no impact on secret key cryptography. Grover's algorithm can speedup breaking secret key cryptosystems, reducing the time required from 2^n to $2^{n/2}$, where n is normally 128 or 256.

Whilst Shor's algorithm does break several modern cryptosystems, at the moment there does not exist a quantum computer large enough to run the algorithm. The current largest quantum computer is IBM's Osprey which has just 433 qubits (as of 9th November 2022). Whilst 433 qubits seems small, it is a huge increase on the previous largest quantum computers. IBM's Eagle, launched in November 2021, which has 127 qubits and Google's Sycamore which has 53 qubits. IBM also aim to launch a qauntum computer with over 1000 qubits in November 2023.

Experts have estimated that it could take 10-20 years before any newly standardised algorithms are fully rolled out and used widely enough in practice. Typically there is also a need for sensitive data to be secure for a number of years after it was encrypted. Combining these two requirements, it's clear to see that the risk is not the state of quantum computing now, but rather the state that quantum computing will be in 20 to 30 years time.

Post quantum cryptography (PQC) is a term used to describe new cryptosystems that have been created to be secure against adversaries who have access to large quantum computers, but is still run on classical computers. Most secret key cryptosystems, such as AES, only require the size of the secret key to be doubled in

order to be secure against quantum computers, they are considered to be quantum secure. However AES is often not considered to be a post quantum cryptosystem since it does not need to change to become quantum secure.

There have been a number of new ideas for problems that are hard for both classical and quantum computers. These can be broadly split into five main of groups: lattice-based, code-based, isogeny-based, hash-based and multivariate. Lattice-based cryptosystems are based on the difficulty of finding a short vector in a lattice, code-based cryptosystems are based on the difficulty of decoding a linear error correcting code, isogeny-based cryptosystems are based on the difficulty of finding walks on an isogeny graph , hash-based cryptosystems are based on the difficulty of finding preimages of a hash function and multivariate cryptosystems are based on the hardness of solving a system of multivariate polynomial equations.

At the time of writing there is a movement to standardise post quantum cryptosystems for public key cryptography. One of the main attempts is NIST's post quantum cryptography standardisation process, which is currently standardising the lattice-based scheme Kyber, along with a number of signature schemes, and has a view to standardise a code-based cryptosystem in the near future.

1.3 Side-channel attacks

Side-channel attacks are a series of attacks on cryptosystems that don't try to break the underlying problem, but instead try to gain extra information about the encryption and decryption processes by monitoring the machines that are performing these processes.

Whilst almost anything that can be monitored on the target machine can be used as a side-channel, some of the most common attacks include timing, power and cache. In a timing based side-channel attack, the adversary monitors how long it takes to perform the encryption or decryption process and can use this to deduce information about the key being used. For some intuition of how this works, consider the case of raising a value to a secret power using the square and multiply method as given in Fig. 1.1. The algorithm iterates over each bit of the secret value and

Figure 1.1: The square and multiply algorithm as given in [Koc96]

```
Let s_0 = 1.
For k = 0 upto w − 1:
  If (bit k of x) is 1 then
    Let R_k = (s_k · y) mod n.
  Else
    Let R_k = s_k.
  Let s_{k+1} = R_k^2 mod n.
EndFor.
Return (R_{w−1}).
```

in the case where the bit (k) is 1 it performs an extra multiplication (line 4) then when the bit is 0 (line 6) It becomes easy to see how this creates a discrepancy in the timing that can be measured.

Power analysis is even more powerful than timing attacks, as it can distinguish between cases that are constant time but require a different amount of power. Power analysis attacks are seen as being less applicable, since they require having physical or virtual access to the machine running the computation, whereas a timing attack can be launched against a remote server. Cache attacks make use of the ability to monitor the algorithm's access to caches, and can be launched against any device where the adversary can monitor cache access, e.g. using the same physical hardware but in different virtual environments.

Three main techniques have been proposed to secure cryptosystems against these attacks. The first method is just restructuring the algorithm. Some examples of this include removing branches and removing variable length loops. The second method is blinding, where the idea is to add random noise to the input which can then be easily removed from the output. The third method is masking, where the input is split into different shares and the algorithm is run on each share separately.

1.4 Error correcting codes and cryptography

When messages are sent on a network, there is the possibility that an error can occur. In the same way it is possible that during the encryption of a message that

a small error is made. In order to ensure that we can still read the messages we make use of error correcting codes. These codes encode a message into a codeword with the aim of making it possible to correct any errors that might occur during transmission. For a simple example consider a code that takes a message and repeats it 4 times, so the message 0 would be turned into the codeword 00000. If one of these bits was changed during transmission (e.g. giving us 01000) then it is easy to see what the original message was probably meant to be (in our case 0). Whilst this code has high redundancy, others are very space efficient albeit with an increase in the amount of time required to decode the message.

There are two main places that error correcting codes are used within cryptography, firstly for their intended use - to correct errors - and secondly they can be used for the cryptosystem itself. We first discuss the use of error correcting codes for correction. For some cases where public key cryptography is used, especially when lattice-based cryptosystems are used, there is a chance that some parts of the message are encrypted in such a way that they can't be decrypted correctly. These cases are referred to as decryption failures. Applying an error correcting code to the message before it is encrypted can help reduce the probability of a decryption failure occurring, however it can't eliminate it completely. This also requires the codeword to be decoded after the decryption has occurred, increasing the amount of time required to completely perform the decryption.

The second use of error correcting codes within cryptography is as part of the cryptosystem itself, as is the case with code-based cryptography. Whilst some of the work of this thesis could be applied to code-based cryptosystems, we leave this as future work, and so we will not explain this area of cryptography in detail.

1.5 Outline of the thesis

This thesis examines: how can we secure error correcting codes against side-channel attacks so that they can be securely used in cryptography?, as well as how can they be used to improve certain lattice-based cryptosystems? An overview of how these ideas are presented in the thesis is given below:

Chapter 2 - Preliminaries. This chapter gives a more detailed overview of the mathematical background and notation used within this thesis and then explains the cryptographic background in more detail including the methods of proving security.

Chapter 3 - Using error correction for post quantum cryptography. In this chapter we discuss how error correcting codes can be used to reduce the size of ciphertexts produced by LWE based schemes. The key contributions of this chapter are the use of Gray codes to reduce the number of bit errors when multi-bit encryption techniques are used, the full analysis of how various techniques could be applied to current KEMs (rather than to just a general scheme) with scripts to enable researchers to find improved parameter sets from a given starting point, and to provide specific parameter sets for these KEMs.

Chapter 4 - Securing linear algebra. We move on to show how various linear algebra algorithms, including LUP Decomposition, can be made to be secure against side-channel attacks. We prove the security of these algorithms in the probing mode as well as giving experimental proofs.

Chapter 5 - Securing BCH codes. In this chapter we show how the algorithms we secured in the previous chapter can be used to create a secure version of the BCH code decoding algorithm. We also prove the security of these algorithms in the probing mode as well as giving experimental proofs.

Chapter 6 - Securing Polar codes. Having shown how to secure the BCH code decoding algorithm, we now show how to secure the decoding algorithm for Polar codes. As with the BCH code decoding algorithm, we also prove the security of these algorithms in the probing mode as well as giving experimental proofs.

Chapter 7 - Cost of using error correcting codes. In this chapter we take the masked versions of the error correcting codes from the previous chapters, and the suggested changes from Chapter 3, and show the cost of using these error correcting codes.

CHAPTER 2

Preliminaries

2.1 Mathematical background

Here we will outline the mathematics that is required for understanding the cryptosystems, and will assume little to no prior knowledge.

Definition 2.1.1. *(Group, $(S, \circ)$)*
A non-empty set S with a binary operation $\circ$ forms a group if:

1. *$\circ$ is associative i.e. $(a \circ b) \circ c = a \circ (b \circ c), \forall a, b, c \in S$,*
2. *There is an identity element e i.e. $\exists e$ s.t. $\forall a \in S, e \circ a = a \circ e = a$,*
3. *Each element has an inverse i.e. $\forall a \in S, \exists b \in S$ s.t. $a \circ b = e$.*

Moreover this group is said to be an abelian group if $\circ$ is commutative i.e. $a \circ b = b \circ a, \forall a, b \in S$

Definition 2.1.2. *(Ring, $(R, +, \cdot)$)*
A non-empty set R with two binary operations $+$ ('addition') and $\cdot$ ('multiplication') such that:

1. *(R,+) is an abelian group,*

2. *Multiplication is associative i.e.* $(a \cdot b) \cdot c = a \cdot (b \cdot c), \forall a, b, c \in R$,

3. *Multiplication is distributive over addition i.e.* $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$ $\forall a, b, c \in R$.

For example the set of integers modulo q with addition and multiplication forms a ring.

Definition 2.1.3. *(Field)*
A non-empty set F with two binary operations $+$ and $\cdot$ where:

1. $(F, +, \cdot)$ *is a ring.*

2. $(F \backslash \{0\}, \cdot)$ *is an abelian group.*

For example, the set of complex numbers equipped with addition and multiplication forms a field. The set of integers modulo p, where p is prime, form a finite field.

Definition 2.1.4. *(Vector space)*
A vector space over the field F is a non-empty set V with two binary operations $+$ and $\cdot$ where:

1. $(V, +)$ *is an abelian group,*

2. *Field multiplication and scalar multiplication are compatible, i.e.* $a(bv) = (ab)v \forall a, b \in S, \forall v \in V$,

3. *Scalar multiplication is distributive over both field and vector addition, i.e.* $a(u + v) = au + av, \forall a \in F, \forall u, v \in V, (a + b)v = av + bv \forall a, b \in F, \forall v \in V$.

For example the complex numbers form a vector space over the reals, with the set $\{i, 1\}$ as a basis.

Definition 2.1.5. *(Subfield, Extension)*
If the field F is contained in the field K, denoted as K/F, then F is said to be a subfield of K and K an extension of F.

For example, the field of complex numbers is an extension of the field of real numbers. Conversely, the field of real numbers is a subfield of the field of complex numbers.

Definition 2.1.6. *(Degree)*
The degree of a field extension K/F, denoted as $[K : F]$, is the dimension of K as a vector space over F.

For example, the degree of the field extension $[\mathbb{C} : \mathbb{R}]$ is 2, with $\{i, 1\}$ being the basis.

2.2 Introduction to PKC

2.2.1 Public Key Cryptography

Public Key Encryption schemes The aim of public key cryptography (PKC) is to have a public key that anyone can use to encrypt messages with, whilst only the person with the secret key is able to decrypt. A public key encryption scheme (PKE) is a 3-tuple of algorithms (KeyGen, Encrypt, Decrypt). KeyGen is a probabilistic algorithm that takes as input a security parameter λ and returns a public key pk and a secret key sk. Encrypt is a probabilistic algorithm that takes as input a message m and a public key pk, and returns a ciphertext c. Decrypt is a deterministic algorithm that takes as input a ciphertext c and a secret key sk and decrypts it to give the message m. With high probability $\text{Decrypt}(\text{Encrypt}(m, pk), sk) = m$.

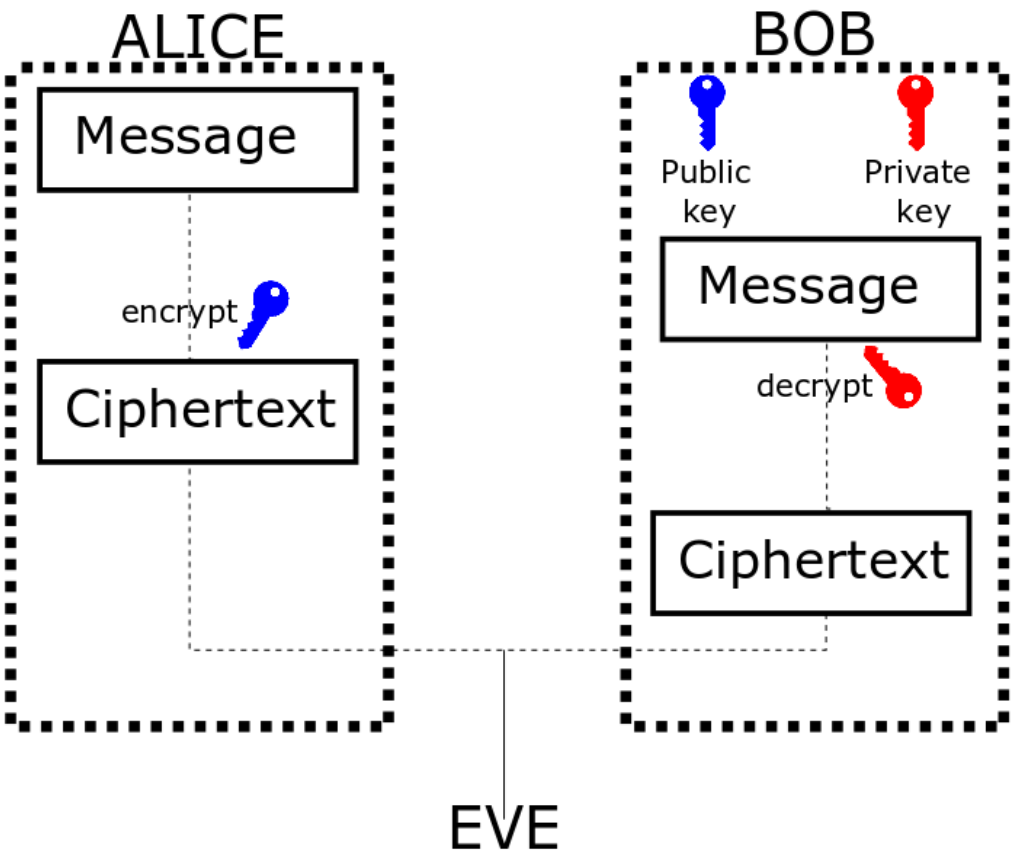

Figure 2.1: The pipeline for public key encryption.

Key Encapsulation Mechanism Public key cryptography (PKC) is quite slow for long messages when compared to secret key cryptography. To get around this, one of the common use cases for PKC is to send a key that will then be used to encrypt larger messages using secret key cryptography. However most keys that are used by secret key cryptography are much shorter than the message size used for public key cryptography. Padding schemes [PKC91] are used to increase the size of the key, however theses are often not secure [Ble98] and have been exploited [CJNP00]. In order to avoid using padding, Key Encapsulation Mechanisms were introduced. A Key Encapsulation Mechanism is a 3-tuple of algorithms (KeyGen, Encaps, Decaps). KeyGen is a probabilistic algorithm that takes as input a security parameter λ and returns a public key pk and a secret key sk. Encaps is a probabilistic algorithm that takes as input a public key pk, and returns a key k and an encapsulation c. Decaps is a deterministic algorithm that takes as input an encapsulation

c and a secret key sk and decapsulates it to give either the key k or $\perp$. With high probability $\textsc{Decaps}(\textsc{Encaps}(k, pk), sk) = k$, when the algorithms are run correctly.

2.2.2 Security notions

In modern cryptography we have a large number of different types of attack models, each of which tends to relate to different assumptions about what an adversary does or does not have access to when they are trying to break the cryptosystem. The main assumption that they all have in common is Kerckhoffs's principle: that the adversary has exact knowledge of how the cryptosystem works.

Attack models

There are various types of attack that can be launched against a cryptosystem. The attacks are classified by the amount of information that the adversary has, we have listed them in increasing order based on the amount of information the adversary has. This order is also approximately the same as how easy it is to secure a cryptosystem against an attack (i.e. the first one is the simplest to secure against).

CPA

Chosen plaintext attacks (CPA) form the foundation of modern cryptanalysis. An adversary using a CPA is allowed to choose an arbitrary amount of plaintexts and request the corresponding ciphertexts. Adaptive chosen plaintext attacks, or CPA2, is a stronger generalisation of CPA where the adversary is able to choose plaintexts after seeing some of the ciphertexts. This model of attack can be seen as the case where the adversary has access to a blackbox that will encrypt plaintexts. In the case of public key cryptography the public key is known by the adversary, and as such it is easy to see how this style of attack could be launched. Modern cryptosystems are expected to be secure against chosen plaintext attacks as a minimum.

CCA

Chosen ciphertext attacks (CCA) are the form that most modern attacks take, such as the successful attacks against DES [BS91, Mat94]. In this attack the adversary first gets to choose a set of plaintexts to be encrypted, and can perform some computation. Once they've finished they choose a set of ciphertexts and are given access to their corresponding plaintexts, after this point they can ask for further plaintexts to be encrypted but not for any more ciphertexts to be decrypted. This attack is sometimes called a lunchtime attack, as it represents the case where an adversary has managed to gain one off access to the decryption device for a short period e.g. a lunchtime. This is often referred to as non-adaptive CCA or CCA1.

CCA2

Adaptive chosen ciphertext attacks (CCA2), first defined in 1991 by Rackoff and Simon [RS92], is the attack style that cryptosystems aim to defend against. Under this model, the adversary has access to a black box that will both encrypt and decrypt. There are few examples of practical CCA2 attacks, but despite this security against CCA2 (IND-CCA2) is required of all but a small subset of modern cryptosystems - this is often called Active Security or security against an Active Adversary. Whilst adaptive and non-adaptive chosen plaintext attacks are equivalent for asymmetric schemes , there is a gap between CCA and CCA2.

Standards for security against attack models

Indistinguishability

The notion of indistinguishability in terms of security was defined by Goldwasser and Micali [GM84]. A cryptosystem is indistinguishably secure if (under a particular attack model) given two messages and a ciphertext which corresponds to one of the messages, it is impossible to work out which of the two the ciphertext corresponds to in polynomial time.

This is formalised as a game with an advantage, in this case you 'win' the game if you can distinguish between which of the two plaintexts was encrypted. The advantage is a metric of how much better an attacker can perform than random, where advantage 0 means the attacker has no advantage over a random guess, and advantage 1 means the attacker can always win the game. For indistiguishability we define the advantage as $Adv = 2|P(A \text{ wins}) - \frac{1}{2}|$.

Within the cryptographic community, the standard is that security under IND-CPA2 suffices for ephemeral key usage and security under IND-CCA/CCA2 is required for any situations where a key is being reused. The justification for this being that if a secret key is only used once then only one decryption is being performed, and so an attacker is unable to choose multiple ciphertexts to be decrypted.

NIST Security

When NIST launched the call for submissions [PQC16a, PQC16b], they gave detailed security requirements. Firstly the call indicated what models of attack it considered as being valid, and secondly it gave concrete values as to how secure the cryptosystems should be against each of these attacks. Finally they detail the limit of what they consider to be valid quantum attacks.

Security Levels

NIST has 5 levels of security, based on how long it takes to break a particular cipher or hash function. A cryptosystem is considered secure at Level I (or III or V) if "any attack that breaks the relevant security definition must require computational resources comparable to or greater than those required for key search on a block cipher with a 128-bit (192-bit, 256-bit) key" [PQC16b]. A cryptosystem is considered secure at Level II (or IV) if "any attack that breaks the relevant security definition must require computational resources comparable to or greater than those required for collision search on a 256-bit (384-bit) hash function" [PQC16b]. For CCA security, the definition is relaxed slightly by capping the adversary to only

having 2^{64} ciphertexts.

In terms of real time and computational power, if every core of the current largest supercomputer (all 2,414,592) was running Intel's optimised AES hardware instructions, then it would take approximately 9.64×10^{23} seconds (3×10^{16} years) to break AES-128 (Level I), assuming ideal AES.

Quantum security

NIST also proposed security levels against quantum adversaries, by giving bounds on the amount of logical qubits and quantum gates. The relevant extract from the call for proposals is given below: "As preliminary guidance to submitters, NIST suggests an approach where quantum attacks are restricted to a fixed running time, or circuit depth. Call this parameter MAXDEPTH. This restriction is motivated by the difficulty of running extremely long serial computations. Plausible values for MAXDEPTH range from 240 logical gates (the approximate number of gates that presently envisioned quantum computing architectures are expected to serially perform in a year) through 2^{64} logical gates (the approximate number of gates that current classical computing architectures can perform serially in a decade), to no more than 2^{96} logical gates (the approximate number of gates that atomic scale qubits with speed of light propagation times could perform in a millennium)." [PQC16b]

Fujusaki-Okamoto Transform

The Fujusaki-Okamoto (FO) transform is used to help take PKE schemes, which are easy to create, and to convert them into secure KEMs. There exists two versions of the Fujisaki-Okamoto transform: the former converts an IND-CPA secure PKE and an IND-CPA symmetric key cryptosystem (e.g. AES) into a hybrid system that is IND-CCA2 secure [FO99b], while the latter converts an IND-CPA secure PKE and a secure hash function into a IND-CCA2 secure KEM [FO99a]. The transform we will focus on is the latter of the two. Several PKE schemes are non-deterministic, in order to make the encryption algorithm deterministic it's common to tweak the encryption algorithm so that it takes a seed as input. The seed is then used as the seed for generating random numbers, meaning that the encryption of a message

with a seed will always be the same. The IND-CCA2 hybrid cryptosystem given by applying the FO-Transform has only one minor change to the standard IND-CPA version. Instead of encrypting a message using a random seed, we encrypt the message appended with a random value, and use the hash of this as the seed. More formally, let $\mathcal{E}(x;s)$ denote the original scheme encrypting the message x using the seed s, $\bar{\mathcal{E}}(x;s)$ denote the new scheme encrypting the message x using the seed s, $\mathcal{D}(y)$ denote the original scheme decrypting the message y, $\bar{\mathcal{D}}(y)$ denote the new scheme decrypting the message y and $H(\cdot)$ denote a hash function. Then $\bar{\mathcal{E}}(x;s) = \mathcal{E}((x||s); H(x||s))$, and $\bar{\mathcal{D}}(y) = (\mathcal{D}(y)$ if $(\mathcal{E}(\mathcal{D}(y); H(\mathcal{D}(y))) = y)$ else reject). There are two methods for rejection, explicit and implicit. For explicit rejection an error is thrown, or $\perp$ is returned, for implicit rejection a new value is generated, normally the hash of some secret value (e.g. secret key). The FO-transform is secure under classical assumptions. However it has been shown to be insecure under quantum assumptions [TU16]; in the same paper a quantum version of the asymmetric/symmetric FO transform is provided. A quantum version of the PKE to KEM FO-transform has since been provided [HHK17], and is used by schemes such as Kyber [SAB+22].

2.3 Lattice based cryptography

2.3.1 LWE

The Learning with Errors (LWE) [Reg05] problem was first proposed by Regev in 2005. Several generalisations and variants have been proposed since and will be discussed later. LWE is also the basis of Frodo, which was one of the candidates in the NIST PQC standardisation process.

LWE Problem

Simply put, the problem is given a pair $(\mathbf{A}, \mathbf{b})$ determine if they are uniformly random or if there exists an $\mathbf{s}$ such that $\mathbf{b} \approx \mathbf{s} \cdot \mathbf{A}$. The decision version of the problem consists of a set of pairs $(\mathbf{a}_i, b_i)$, with the aim being to determine if they correspond to a set of equations in the below format or if they're just uniformly random values.

$$b_1 = \langle \mathbf{s}, \mathbf{a}_1 \rangle + \epsilon_1 (\text{mod } q)$$

$$\vdots$$

$$b_i = \langle \mathbf{s}, \mathbf{a}_i \rangle + \epsilon_i (\text{mod } q)$$

$$\vdots$$

$$b_n = \langle \mathbf{s}, \mathbf{a}_n \rangle + \epsilon_n (\text{mod } q)$$

With $\langle \mathbf{s}, \mathbf{a}_i \rangle$ is the inner product of $\mathbf{s}$ and $\mathbf{a}_i$.

Where: $\mathbf{s} \in \mathbb{Z}_q^n$, $\mathbf{a}_i$ are chosen independently from $\mathbb{Z}_q^n$, $b_i \in \mathbb{Z}_q$ and $\epsilon_i \in \mathbb{Z}_q$ is chosen independently from the probability distribution $\chi : \mathbb{Z}_q \to \mathbb{R}^+$ on $\mathbb{Z}_q$.

There is also a corresponding search version of the problem, where the aim is to find $\mathbf{s}$.

$\text{LWE}_{q,\chi}$	
Instance:	m pairs of $(\mathbf{a}_i, b_i)$.
Question:	Determine if the pairs are of the form $b_i \langle \mathbf{s}, \mathbf{a}_i \rangle + \epsilon_i (\text{mod } q)$ or if they are random.

$\text{LWE-SEARCH}_{q,\chi}$	
Instance:	m pairs of $(\mathbf{a}_i, b_i)$, where $b_i = \langle \mathbf{s}, \mathbf{a}_i \rangle + \epsilon_i (\text{mod } q)$.
Question:	Find the corresponding $\mathbf{s}$ value.

The search version of the problem is sometimes denoted as SLWE, and has been shown to be equivalent to the decision version when q is prime, and $q = O(n^2)$.

In Regev's cryptosystem, explained below, the specific problem used is $\mathbf{LWE}_{q,\Psi_\alpha}$, where $q = O(n^2)$ and Ψ_α is the distribution that the errors are drawn from. In the original cryptosystem this was a discrete Gaussian distribution, with a standard deviation of αq, where $\alpha = 1/(\sqrt{n} \log^2 n)$, however in more recent cryptosystems, such as Kyber [SAB+19] a centered binomial distribution is used.

Cryptosystem

Regev also proposed a basic cryptosystem based on this problem [Reg05]. The cryptosystem has a security parameter n and is parameterised by two integers m, q and a probability distribution χ on $\mathbb{Z}_q$. The parameters should be chosen as follows:

- $q > 2$ is a prime number
- $n^2 < q < 2n^2$
- $m = O(n \log q)$.
- χ is taken to be $\Psi_{\alpha(n)}$
- $\alpha(n) = o(1/\sqrt{n} \log n)$.

All additions are done in $\mathbb{Z}_q$.

Key generation:
The private key is generated by choosing $\mathbf{s} \in \mathbb{Z}_q^n$ uniformly at random.
The public key is generated by choosing m vectors $\mathbf{a}_1, \ldots, \mathbf{a}_m \in \mathbb{Z}_q^n$, each of which is chosen uniformly at random, and also choosing m elements $e_i, \ldots, e_m \in \mathbb{Z}_q$ chosen uniformly at random according to χ.
The public key is then $(\mathbf{a}_i, b_i)_{i=1}^m$ where $b_i = \langle \mathbf{s}, \mathbf{a}_i \rangle + e_i$.

Encryption:
To encrypt a single bit(z) a subset S of $[m]$ is chosen.
The actual encryption is done as $(\sum_{i \in S} \mathbf{a}_i, \sum_{i \in S} b_i + \lfloor \frac{q}{2} \rfloor \cdot z)$.

Decryption:
Decryption is done by checking if $(b - \langle \mathbf{a}, \mathbf{s} \rangle)$ is closer to 0 mod q (in which case z was 0) or to $\lfloor \frac{q}{2} \rfloor$ mod q (in which case z was 1).

2.3.2 R-LWE

Ring Learning with Errors (R-LWE) was first presented in a paper at Eurocrypt 2010 [LPR10] and was expanded on further in 2012 [LPR12]. In these works Lyuba-

shevsky, Peikert and Regev (LPR) showed how the LWE cryptosystem could be made more efficient by exploiting the properties of rings. R-LWE also acts as the basis for several cryptosystems such as NewHope [PAA+19] and LAC [LLJ+19].

Problem

The ring used is described by the polynomial

$$f(x) = x^n + 1 \in \mathbb{Z}[x]$$

where n is a power of 2. This gives us the ring

$$R = \mathbb{Z}[x]/f(x).$$

Which is the ring of integer polynomials mod $f(x)$ i.e. all polynomials of degree less than n. The other ring we use is the ring

$$R_q = R/q = \mathbb{Z}_q[x]/f(x).$$

Which is the ring of integer polynomials mod $f(x)$ and mod q . The R-LWE problem is very similar to the standard LWE problem, consisting of the following set of equations and trying to distinguish them from noise, where $a_i, s, b_i \in R_q$.

$$b_1 = a_1 \cdot s + \epsilon_1$$

$$\vdots$$

$$b_i = a_i \cdot s + \epsilon_i$$

$$\vdots$$

$$b_m = a_m \cdot s + \epsilon_m$$

Cryptosystem

The cryptosystem is parameterised by two integers, n and q, and a distribution Ψ.

Key Generation:
The keys are generated by taking $m = O(\log q)$ elements $a_i \in R_q$ and m small elements $r_i \in R$ (where small means all coefficients are 0 or 1).
a_{m+1} is calculated as $\sum_{i\in[m]} r_i.a_i$, $r_{m+1} = -1$.
The public key is $(a_1, \ldots, a_{m+1}) \in R_q^{m+1}$.
The private key is $(r_1, \ldots, r_{m+1}) \in R^{m+1}$.

Encryption:
To encrypt a n bit message $z \in \{0,1\}^n$, a secret $S \in R_q$ is chosen and m sets of equations are generated.
For $i \in [0, m+1]$ let $b_i \approx_{\epsilon_i} a_i.S \in R_q$.
Where $\epsilon \in \Psi_\alpha$ and $\alpha = \sqrt{\frac{\log n}{n}}$.
$b_{m+1} = b_{m+1} - z \cdot \lfloor \frac{q}{2} \rceil$.
The cipher text returned is $(b_1 \ldots b_{m+1}) \in R_q^{m+1}$.

Decryption:
Decryption is done by calculating $\sum_{i\in[0,m+1]} r_i \cdot b_i$ which is approximately $z \cdot \lfloor \frac{q}{2} \rceil + (\sum_{i\in[0,m+1]} r_i \cdot a_i) \cdot S = z \cdot \lfloor \frac{q}{2} \rceil$.
So by discretizing $\sum_{i\in[0,m+1]} r_i.b_i$ to 0 or $\frac{q}{2}$, we get a value of z.

2.3.3 LWR

Learning with Rounding (LWR) was proposed by Banerjee, Peikert and Rosen [BPR12] in 2011 and they similarly extended it to R-LWR.

Problem

Similar to LWE:
The problem consists of a set of pairs $(\mathbf{a}_i, b)$, with the aim being to determine if

they correspond to a set of equations in the below format or if they're just random values (i.e. determine $\mathbf{s}$).

$$b_i = \lfloor \langle \mathbf{a}_i, \mathbf{s} \rangle \rceil_p (\text{mod } q)$$

With $\lfloor \cdot \rceil_p$ meaning discretize to one of p values, normally considered as $\mathbb{Z}_p$, where the mapping from $\mathbb{Z}_q$ to $\mathbb{Z}_p$ is done by multiplying the element from $\mathbb{Z}_q$ by $\frac{p}{q}$.
Where: $\mathbf{s}$ and $\mathbf{a}_i$ are chosen independently from $\mathbb{Z}_q^n$,$b_i \in \mathbb{Z}_p$.

2.3.4 General LWE

The general decision problem for all LWE variants is, given a pair $(\mathbf{A}, \mathbf{B})$ can it be determined if they are random or if they are of the form

$$\mathbf{B} = \lfloor \mathbf{A} \cdot \mathbf{S} + \mathbf{E} \rceil_{q \to p}.$$

Where $\lfloor \cdot \rceil_{q \to p}$ represents rounding from q to p with $p \ll q$, and $\mathbf{A}, \mathbf{S}, \mathbf{B}$ and $\mathbf{E}$ are drawn from the relevant algebraic structure (e.g. $\mathbb{Z}, R_q, R_q^d$). The cryptosystems are generalised as below, general KeyGen in Algorithm 2.1, Encryption in Algorithm 2.2 and Decryption in Algorithm 2.3.

Algorithm 2.1: KeyGeneration

Input:

Result: $\text{pk} \in \{0,1\}^{256} \times \mathbb{Z}_p^{n \times \tilde{n}}$ or $\{0,1\}^{256} \times R_p$ or $\{0,1\}^{256} \times R_p^{k \times 1}$, $\text{sk} \in \mathbb{Z}_q^{n \times \tilde{n}}$ or R_q or $R_q^{k \times 1}$

1 $\mathtt{seed}_{\mathbf{A}} \xleftarrow{\$} U(0,1)^{256}$

2 $\mathbf{A} \leftarrow \mathtt{gen}(\mathtt{seed}_{\mathbf{A}})$

3 $\mathbf{S} \xleftarrow{\$} \chi_s$

4 $\mathbf{E}_A \xleftarrow{\$} \chi_e$

5 $\mathbf{B} \leftarrow \lfloor \mathbf{A} \cdot \mathbf{S} + \mathbf{E}_A \rceil_{q \to p'}$

Output: $\text{pk} = (\mathbf{B}, \mathtt{seed}_{\mathbf{A}})$, $\text{sk} = \mathbf{S}$

Algorithm 2.2: Encrypt

Input: $\text{pk} = (\mathbf{B}, \mathtt{seed}_{\mathbf{A}})$, $\text{message} = \mathbf{m}$

Result: $\text{ct} \in \mathbb{Z}_p^{\tilde{m} \times n} \times \mathbb{Z}_t^{\tilde{m} \times \tilde{n}}$ or $R_p \times R_t$ or $R_p^{k \times 1} \times R_q$

1 $\mathbf{A} \leftarrow \text{gen}(\mathtt{seed}_{\mathbf{A}})$

2 $\mathbf{R} \xleftarrow{\$} \chi_r$

3 $\mathbf{E}'_B \xleftarrow{\$} \chi'_e$

4 $\mathbf{E}''_B \xleftarrow{\$} \chi'_e$

5 $\mathbf{B}' \leftarrow \lfloor \mathbf{A}^\top \cdot \mathbf{R} + \mathbf{E}'_B \rceil_{q \to p}$

6 $\mathbf{V}' \leftarrow \lfloor \mathbf{B}^\top \cdot \mathbf{R} + \mathbf{E}''_B + \frac{q}{2}\text{enc}(\mathbf{m}) \rceil_{q \to t}$

Output: $\text{ct} = (\mathbf{B}', \mathbf{V}')$

Algorithm 2.3: Decrypt

Input: $\text{sk} = \mathbf{S}$, $\text{ct} = (\mathbf{B}', \mathbf{V}')$

Result: $\mathbf{m}$

1 $\mathbf{B}' \leftarrow \lfloor \mathbf{B}' \rceil_{p \to q}$

2 $\mathbf{V}' \leftarrow \lfloor \mathbf{V}' \rceil_{t \to q}$

3 $\mathbf{V} \leftarrow \mathbf{B}'^\top \cdot \mathbf{S}$

4 $\mathbf{m}' \leftarrow \lfloor \mathbf{V}' - \mathbf{V}^\top \rceil_{q \to 2}$

5 $\mathbf{m} \leftarrow \text{dec}(\mathbf{m}')$

Output: $\text{pt} = \mathbf{m}$

2.4 Introduction to side-channel attacks

Side-channel attacks are a set of attacks that do not aim to break the underlying problem of the cryptosystem, but to instead measure and use information about the secret information that is leaked by the hardware.

The first side-channel attacks - timing attacks - were introduced in 1996 by Kocher [Koc96] and make use of statistical differences in the time taken to perform encryption using different keys. Timing attacks can be used against any system which the adversary can accurately time, this could be a machine that they have access to or even a remote server. Due to the effectiveness and ease with which timing attacks can break unprotected cryptosystems, it is now considered standard to ensure that all aspects of a cryptosystem that use secret data run in constant time. It is often possible to rewrite code to ensure that it runs in constant time, primarily by applying techniques such as removing branching on key variables.

Power analysis attacks were first introduced by Kocher et al. [KJJ99] in 1999, and make use of statistical differences in the power usage between encryption using different keys. Power analysis attacks are seen as being harder to perform as it requires the attacker to be able to measure the power consumption of the target device with fine detail. However recent work [WPH+22] has shown that these attacks can be launched remotely as some severs increase the clock speed based on power usages, allowing power usage to be measured by time.

There are many other flavours of side-channel attacks that exploit different measurements in a broadly similar way. In order to make it easier to discuss security, a generalised model of side-channel attacks has been introduced, known as the probing model [ISW03]. In the probing model we consider a Boolean (or arithmetic) circuit, and a number of probes. Probes can be placed on the wires between gates and are able to perfectly read the value on the wire. A circuit is considered to be t secure in the probing model if an adversary gains no advantage from placing t probes on the circuit.

In the probing model each input is split into shares, typically $t+1$ shares, where the sum of all shares gives the input and the distribution of the sum of any t shares should appear uniformly random. Each gate then needs to be converted to a gadget

that takes $t+1$ times the number of gate inputs whilst ensuring that knowledge of any set of t wires do not give an adversary any advantage.

As well as circuits, it is possible to view probing security in terms of probability distributions. An algorithm can be considered to be t-probing secure if the cross distribution of every set of t intermediate variables depends on at most t input values. Using this view point, two formal notions of security were proposed [BBD+16]. The first, non-interference (t-NI), ensures that the joint distribution of any set of at most t shares is dependent on at most t input shares, however having multiple gadgets that are t-NI secure does not mean that the composition of these gadgets is also t-NI secure. In Fig. 2.2 we give a slightly contrived example of 4 t-NI secure gadgets that perform addition being composed to calculate $3a+3b$ in such a way that the overall sum is not t-NI , since given the first output share $(3a_1+b)$, and one share of a, a_1, gives the full input value b. This led to the second notion, strong non-interference (t-SNI), which strengthens t-NI in such a way that it ensures secure composition. We breifly discuss the notation used before giving the formal definitions of t-NI and t-SNI.

In the rest of this section, and in our definitions of t-NI and t-SNI, we will make use of the following notation. We use a bold capital, $\mathbf{M}$, to refer to a vector or matrix where each variable is shared. We use $\mathbf{M}[i][j]$ to reference the value in the ith row and jth column of $\mathbf{M}$, we extend this to sets and use $\mathbf{M}[\mathcal{I}][\mathcal{J}]$ to reference to all values $\{\mathbf{M}[i][j] : \forall i \in \mathcal{I}\ \forall j \in \mathcal{J}\}$. We use $\mathbf{a}^{(s):(b)}$ to refer to the bth bit of the sth share of the shared variable $\mathbf{a}$. For simplification we use $\mathbf{a}^{(s)}$ to refer to the sth share of the shared variable a, and $\mathbf{a}^{(.):(b)}$ to refer to the shares of the bth bit of the shared variable. s and b can also be extended to sets in a similar form to the indices of the matrix. We will be writing each algorithm in single static assignment (SSA) form, so we will use subscripts to refer to the different assignments of the variable. This will often give us variables such as $\mathbf{M}_x^{(s):(b)}[i][j]$, which refers to using the bth bit of the sth share of the value in the ith row and $j$the column of the xth assignment of the shared variable $\mathbf{M}$.

For a gadget G, we define the *input* shares as the shares of the input variables, the *output* shares as the shares of the output variables, the *internal* shares as the shares

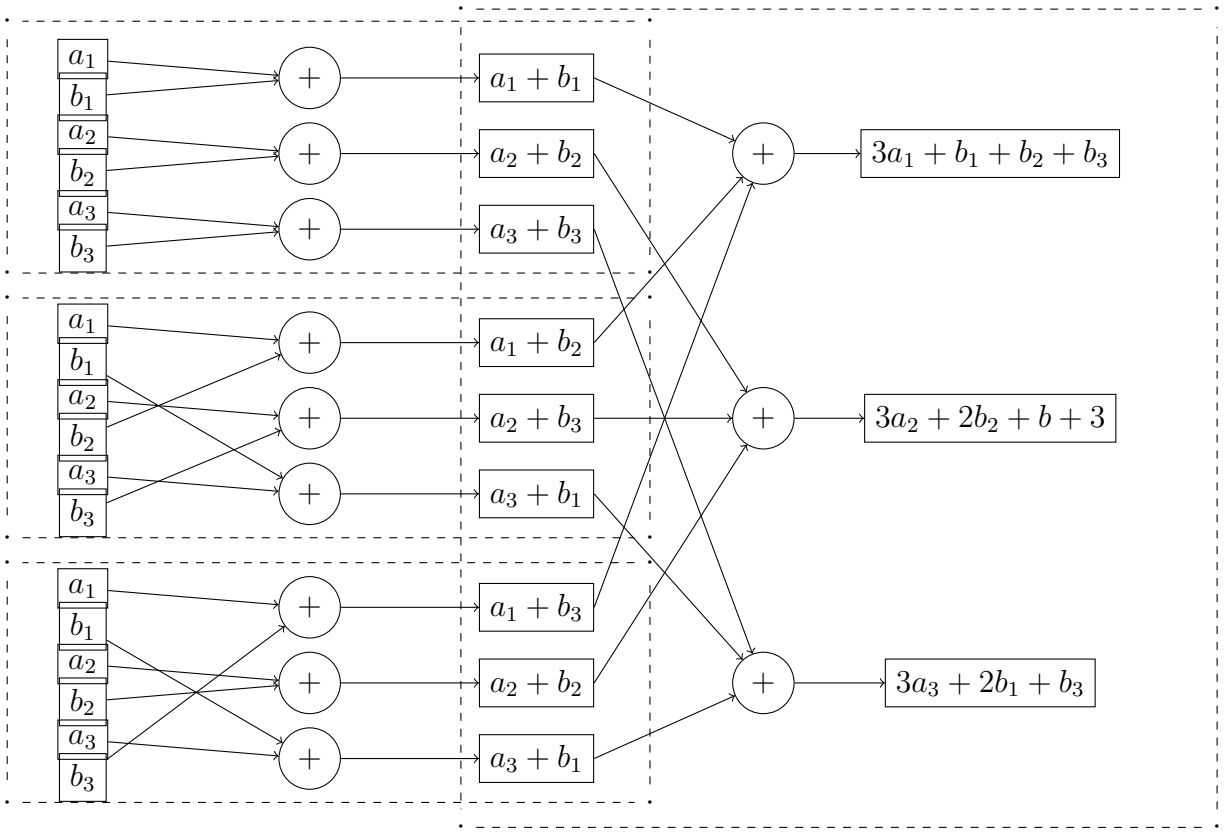

Figure 2.2: An example showing that the composition of t-NI gadgets does not give a t-NI gadget.

of all variables that are not part of the input or output, and finally the *intermediate* shares as shares of internal and output variables. We denote the number of shares per variable as G_s.

In diagrams representing the sequences of gadgets, we use a circle to represent a t-NI gadget, a double circle to represent a t-SNI gadget, a rectangle to represent an input, a double rectangle to represent an output, a directed arrow to indicate that an output of the source is the input of the sink. We use dots to represent the continuation of gadgets in parallel, and dashed arrows to represent the continuation of gadgets in series. In algorithms we take GADGET as being the normal version of the gadget, GADGETS(n) as running the gadget n times in series and GADGETP(n) as running the gadget n times in parallel. Unless otherwise stated, we set $\mathrm{G}_s = t+1$.

Definition 1 (t-NI security [BBD+16] [BGR+21])**.** *Let* G *be a gadget taking as input* $\boldsymbol{x}^{(\cdot)}$ *and outputting* $\boldsymbol{y}^{(\cdot)}$*. The gadget* G *is t-NI secure if for any set of* $t_G \leq t$

intermediate variables, there exists a subset $I \subset [1, t+1]$ *of input indices with* $|I| \leq t_G$*, such that the* t_G *intermediate variables can be perfectly simulated from* $\boldsymbol{x}^{(I)}$*.*

Definition 2 (t-SNI security [BBD+16] [BGR+21])**.** *Let* G *be a gadget taking as input* $\boldsymbol{x}^{(\cdot)}$ *and outputting* $\boldsymbol{y}^{(\cdot)}$*. The gadget* G *is* t*-SNI secure if for any set of* $t_G \leq t$ *intermediate variables and any subset* $O \subset [1, t+1]$ *of output indices, such that* $t_G + |O| \leq t$*, there exists a subset* $I \subset [1, t+1]$ *of input indices with* $|I| \leq t_G$*, such that the* t_G *intermediate variables and the output variables* $\boldsymbol{y}^{(O)}$ *can be perfectly simulated from* $\boldsymbol{x}^{(I)}$*.*

Proposition 1 (Composition of gadgets [BBD+16, Proposition 4])**.** *An algorithm* P *is* t*-NI provided all its gadgets are* t*-NI, and all masked intermediate variables are used at most once as argument of a gadget call other than a* t*-SNI gadget. Moreover* P *is* t*-SNI if it is* t*-NI and one of the following holds, where* G *is some* t*-SNI gadget:*

- *its return expression is* $\boldsymbol{b}$ *and its last instruction is of the form* $\boldsymbol{b} \leftarrow \mathrm{G}(\boldsymbol{a})$*;*
- *its sequence of input parameters is* $(\boldsymbol{a}_1, ..., \boldsymbol{a}_n)$*, its* i*th instruction is* $\boldsymbol{b} \leftarrow_i \mathrm{G}(\boldsymbol{a}_i)$ *for* $1 \leq i \leq n$*, and* $\boldsymbol{a}_i$ *is not used anywhere else in the algorithm.*

The simplest building block is the gadget SNIRef [BBD+16]. This is often used to take a t-NI gadget and turn it into a t-SNI gadget. One of the key gadgets that we will use is SNIMul, proposed in [Cor14] and shown to be t-SNI secure in [BBD+16]. SNIMul, detailed in Algorithm 2.6, takes two shared variables as input and returns a shared variable as output that is the shared version of the multiplication of the two shared inputs. SecMul can be seen as being sharewise multiplication interleaved with SNIRef, for cases where only a t-NI version of the gadget is needed (denoted NIMul) both a general algorithm and optimal algorithms for $t = 2, 3, 4$ are given in [BBP+16].

For $t + 1 = 3$, the t-SNI multiplication algorithm given in Algorithm 2.6 has the

Algorithm 2.4: NISRef

Input: $\mathbf{a}_0 \in \mathbb{K}^{t+1}$

for $i \leftarrow 1$ **to** $t+1$ **do**
  $\mathbf{c}_0^{(i)} \leftarrow \mathbf{a}_0^{(i)}$
end
for $i \leftarrow 1$ **to** $(t+1)$ **do**
  $r_i \xleftarrow{\$} \mathbb{K}$
  $\mathbf{c}_2^{(i)} \leftarrow \mathbf{c}_1^{(i)} + r_i$
  $\mathbf{c}_1^{(i+1)} \leftarrow \mathbf{c}_0^{(i+1)} - r_i$
end

Output: $\mathbf{c}_2$

Algorithm 2.5: SNIRef

Input: $\mathbf{a}_0 \in \mathbb{K}^{t+1}$

for $i \leftarrow 1$ **to** $t+1$ **do**
  $\mathbf{c}_0^{(i)} \leftarrow \mathbf{a}_0^{(i)}$
end
for $i \leftarrow 1$ **to** $(t+1)$ **do**
  for $j \leftarrow 1+1$ **to** $(t+1)$ **do**
    $r_{i(t+1)+j} \xleftarrow{\$} \mathbb{K}$
    $\mathbf{c}_{i(t+1)+j}^{(i)} \leftarrow \mathbf{c}_{i(t+1)+j-1}^{(i)} + r_{i(t+1)+j}$
    $\mathbf{c}_{i(t+1)+j}^{(j)} \leftarrow \mathbf{c}_{i(t+1)+j-1}^{(j)} - r_{i(t+1)+j}$
  end
end

Output: $\mathbf{c}_{t(t+1)}$

Algorithm 2.6: SNIMul

Input: $\mathbf{a}_0, \mathbf{b}_0 \in \mathbb{K}^{t+1}$

for $i \leftarrow 1$ **to** $t+1$ **do**
  $\mathbf{c}_0^{(i)} \leftarrow \mathbf{a}_0^{(i)} \cdot \mathbf{b}_0^{(i)}$
end
for $i \leftarrow 1$ **to** $(t+1)$ **do**
  for $j \leftarrow i+1$ **to** $(t+1)$ **do**
    $s_{i(t+1)+j} \xleftarrow{\$} \mathbb{K}$
    $s'_{i(t+1)+j} \leftarrow (-s_{in+j} + \mathbf{a}_0^{(i)}\mathbf{b}_0^{(j)}) + \mathbf{a}_0^{(j)}\mathbf{b}_0^{(i)}$
    $\mathbf{c}_{i(t+1)+j}^{(i)} \leftarrow \mathbf{c}_{i(t+1)+j-1}^{(i)} + s_{i(t+1)+j}$
    $\mathbf{c}_{i(t+1)+j}^{(j)} \leftarrow \mathbf{c}_{i(t+1)+j-1}^{(j)} + s'_{i(t+1)+j}$
  end
end

Output: $\mathbf{c}_{t(t+1)}$

outputs:

$$c_1 = a_1 \cdot b_1 + t_{1,2} + t_{1,3} = a_1 b_1 + r_{1,2} + r_{1,3}.$$
$$c_2 = a_2 \cdot b_2 + t_{2,1} + t_{2,3} = a_1 b_2 + a_2 b_2 + a_2 b_1 - r_{1,2} + r_{2,3}.$$
$$c_3 = a_3 \cdot b_3 + t_{3,1} + t_{3,2} = a_1 b_3 + a_2 b_3 + a_3 b_3 + a_3 b_2 + a_3 b_1 - r_{1,3} - r_{2,3}.$$

From this it's easy to see that every pair of outputs have some random value in common, and so the distribution of any pair of output shares can be given in terms of just one share of a and some randomness.

Further variants on t-NI exist, such as Probe Isolating Non-Interference [CS18]. The PINI security notion is a composable notion that relies on the locations of probes, rather than the number of probes that the adversary places. This notion is designed such that for any circuit that can be split into $t+1$ circuit shares, where the only interconnections are non-linear gadgets, then the non-interconnecting gadgets are t-probing secure.

As well as the threshold implementations of masking schemes given above, there is also another masking scheme - domain-oriented masking [GMK16]. Whilst we will not be using domain-oriented masking, we briefly outline it. With domain-oriented masking each share of a variable belongs to a different domain, each of which is independent of the other. For most linear function, as with the threshold implementations above, the functions are applied within each domain. However for non-linear functions, where cross domain calculations need to happen, fresh random shares are added.

As well as providing theoretical security proofs in the probing model, it is often common to also provide experimental proofs. This is done by generating traces, where a trace is a measure of a particular property (e.g. power consumption, radiation) over time. Two sets of traces are generated, often one set of traces for a set key and another set where the key is random. For this thesis the traces are typically for a message with 0 errors and the message with the maximum number of errors. The motivation behind this is that the attacks, such as [RRCB20], make use of the knowledge of errors being corrected rather then locations or the specific

number. We therefore use the maximum number of errors, to magnify the noise caused by having errors to correct. Once the traces have been collected, the implementation of the algorithm is tested using the TVLA [GGJR+11] framework. The TVLA framework provides instructions on how the traces should be generated, and how Welch's t-test should be applied. To generate the traces for these experiment results it is common to use emulators, such as ELMO [MOW17], that emulate the power usage of the algorithm on specific hardware, reducing the amount of noise in the traces. Unfortunately we are often only able to produce a small number of traces, in the thousands, due to the size of each trace and limitations of using emulated traces. Whilst the emulated traces are less noisy than standard traces, there is still a gap between this and the standard of producing millions of traces. All experiments in this thesis are performed on the first order masked implementation of the corresponding components.

We also evaluate gadgets using two efficiency metrics: time complexity and randomness complexity. Time complexity is a simple count of the number of atomic non-masked operations, while randomness complexity counts the number of random values that are generated during the masking process. The time complexity of the masked gadget is important, as run time is one of the most common metrics used, after security, when evaluating a KEM [PQC16b]. The randomness complexity of a gadget is also important, as it is difficult to generate secure random numbers for embedded systems [BBP+16]. We will use $\mathcal{T}_{\mathrm{G}}$ and $\mathcal{R}_{\mathrm{G}}$ to represent the time complexity and randomness complexity of gadget G respectively.

2.5 Error correcting codes

We use error correcting codes to encode an input message into a codeword in such a way that if up to a certain number of errors are added during transmission, then the recipient can still decode it to give the original message. More formally the encoding algorithm takes an input message $\mathbf{m}$ of length k and encodes it as a codeword $\mathbf{c}$ of length n, where any two unique codewords differ in at least $d_{\min}$ bits. The codeword is then transmitted across a noisy channel where an error vector $\mathbf{e}$ is added to it,

giving a received vector $\mathbf{r} = \mathbf{c} + \mathbf{e}$. The decoding algorithm will decode the received message $\mathbf{r}$ as $\mathbf{m}'$, if the number of errors added is at most $\lfloor \frac{d_{\min}-1}{2} \rfloor$ then $\mathbf{m} = \mathbf{m}'$. In general an error correcting code can utilise any alphabet, however we consider only binary codes. The redundancy of a code is the number of parity bits added, $n - k$, and the rate of the code is the fraction of the codeword that is made up by message bits, $\frac{k}{n}$. The capacity of a channel is the maximum rate at which information can be accurately transmitted across a noisy channel, we say an error correcting code is capacity achieving if the rate of the code is that same as the channel capacity for a given channel. Most error correcting codes are considered to be 'hard', in the sense that each coefficient of the received vector is considered as being either a 0 or a 1. It is also possible to have 'soft' error correcting codes, here each coefficient is considered as being the probability that it is a 1.

2.5.1 BCH codes

Our brief introduction is based on the excellent lecture notes of Han [Han13], with further information from Wicker [Wic94].

The (binary) BCH code BCH(n, k, d) is a cyclic code of length n, dimension k and minimum distance $\geq d$. For any positive integers m and e there exists a (binary) BCH code with $n = 2^m - 1$, $k \geq n - me$ and $d \geq 2e + 1$. When BCH codes are used practically in cryptosystems, the value of k is often larger than the size of the message being encrypted (normally 128, 192 or 256 bits), and so any unused bits are omitted and assumed to be 0.

Let α be a primitive element in GF(2^m), then the generator polynomial, $\mathbf{g}(x)$, of the code BCH(n, k, d) is defined as the lowest degree monic polynomial over GF(2) with $\alpha, \alpha^2, \alpha^3, \ldots, \alpha^{2e-1}, \alpha^{2e}$ as roots.

We treat the message $\mathbf{m} = (m_0, m_1, \ldots, m_{k-1})$ as a polynomial,

$$\mathbf{m}(x) = m_0 x^0 + m_1 x^1 + \cdots + m_{k-1} x^{k-1}.$$

After encoding the message, the codeword is

$$\mathbf{c}(x) = \mathbf{m}(x)x^{n-k} - (\mathbf{m}(x)x^{n-k} \mod \mathbf{g}(x)),$$

where $\mathbf{c}(x)$ is a polynomial of degree at most $n-1$. The codeword $\mathbf{c}(x)$ is transmitted on the channel, in our case this corresponds to encrypting and the decrypting the codeword, where at most e errors occur. These errors are represented by an error vector $\mathbf{e}(x)$ and the output of the channel is $\mathbf{r}(x) = \mathbf{c}(x) + \mathbf{e}(x)$.

The decoding process, which has an input of $\mathbf{r}(x)$, and returns $\mathbf{c}(x)$, has four components detailed below.

1. Calculating the syndrome:

Let $\mathbf{r}(x) = r_0 + \cdots + r_{n-1}x^{n-1}$ be the received vector, then the syndrome $\mathbf{S} = (S_1, S_2, \ldots, S_{2e})$ where $S_i \in \mathrm{GF}(2^m)$ is given by $S_i = \mathbf{r}(\alpha^i)$.

2. Calculating the error location polynomial:

Given the syndrome $\mathbf{S}$, we create a polynomial, known as the error location polynomial, who's roots correspond to the locations of the errors. The error location polynomial, $\mathbf{\Lambda}(x) = \sum_{i=0}^{e} \Lambda_i x^i$, can be calculated by solving the following equations:

$$\begin{aligned}
\Lambda_0 &= 1 \\
S_1 + \Lambda_1 &= 0 \\
S_3 + \Lambda_1 S_2 + \Lambda_2 S_1 + \Lambda_3 &= 0 \\
S_5 + \Lambda_1 S_4 + \Lambda_2 S_3 + \Lambda_3 S_2 + \Lambda_4 S_1 + \Lambda_5 &= 0 \\
&\vdots \\
S_{2e-1} + \Lambda_1 S_{2e-2} + \Lambda_2 S_{2e-3} + \cdots + \Lambda_e S_{e-1} &= 0
\end{aligned} \tag{2.1}$$

We focus on the Peterson algorithm to calculate $\mathbf{\Lambda}(x)$, which expresses the equations

Algorithm 2.7: ChienSearch

```
   Input: Λ(x)
 1 b = [ ]
 2 for i ← 0 to n do
 3 |   a ← 0
 4 |   for j ← 0 to 2t do
 5 |   |   a ← a + Λ_j
 6 |   |   Λ_j ← Λ_j · α
 7 |   end
 8 |   if a = 1 then
 9 |   |   b = b + [i]
10 end
   Output: b
```

in matrix form as:

$$\mathbf{A\Lambda} = \begin{pmatrix} 1 & 0 & 0 & 0 & \dots & 0 & 0 \\ S_2 & S_1 & 1 & 0 & \dots & 0 & 0 \\ S_4 & S_3 & S_2 & S_1 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ S_{2e-4} & S_{2e-5} & S_{2e-6} & S_{2e-7} & \dots & S_{e-2} & S_{e-3} \\ S_{2e-2} & S_{2e-3} & S_{2e-4} & S_{2e-5} & \dots & S_e & S_{e-1} \end{pmatrix} \begin{pmatrix} \Lambda_1 \\ \Lambda_2 \\ \Lambda_3 \\ \vdots \\ \Lambda_{e-1} \\ \Lambda_e \end{pmatrix} = \begin{pmatrix} -S_1 \\ -S_3 \\ -S_5 \\ \vdots \\ -S_{2e-3} \\ -S_{2e-1} \end{pmatrix}. \tag{2.2}$$

We refer to the matrix $\mathbf{A}$ as the Peterson matrix; it has a non-zero determinant if there are e or $e-1$ errors. The Peterson algorithm [Pet60] removes the bottom two rows and the rightmost two columns of $\mathbf{A}$ until its determinant is non-zero. Once $\mathbf{A}$ has been made nonsingular it is inverted and *Eq.* (2.2) is solved to give $\mathbf{\Lambda}(x)$.

3. Finding the roots of the error location polynomial:

Once the error location polynomial has been calculated, its roots can be found using the Chien search [Chi64] to find the error locations. The Chien search brute forces the roots of $\mathbf{\Lambda}(x)$ by iterating over each possible value of x until the roots are found. Each root represents the location of an error. We outline the pseudocode for the Chien search in Algorithm 2.7.

4. Correcting the errors:

Once the locations of the errors have been found, it is simply a matter of flipping the relevant bits to correct the errors.

The pseudocode of the decoding algorithm is in Algorithm 2.8.

Algorithm 2.8: BCHDecoding

Input: Received vector $r(x)$

Result: Codeword $r(x)$

```
1  S ← Syndromes(r(x))
2  M ← GeneratePetersonMatrix(S)
3  det ← 0
4  while det == 0 do
5  |   det ← LUPDeterminant(M)
6  |   if det == 0 then
7  |   |   M ← M[0 : −2][0 : −2]
8  |   end
9  end
10 Λ(x) ← LUPSolve(M)
11 r(x) ← ChienSearch(Λ(x), r(x))
```

Output: $\mathbf{r}(x)$

2.5.2 Polar codes

Polar codes [Ari09] are a family of soft error correcting codes that are capacity-achieving for a number of binary channels. For clarity we use slightly different notation for polar codes than used previously. The encoding algorithm takes a binary message $\mathbf{m}$ of length k as input and produce a $N = 2^n$ bit codeword c, where $k \leq N = 2^n$. The decoding algorithm takes a received vector $\mathbf{r}$ of length $N = 2^n$, that it then decodes to give $\hat{\mathbf{c}}$ of length $N = 2^n$ and finally the data bits are removed to give the decoded message $\hat{\mathbf{m}}$. The polar code construction polarises the channels such that some approach a capacity of 1 and others approach a capacity of 0. The $2^n - k$ lowest capacity channels are frozen. The polarization process is a recursive algorithm that is laid out in Algorithm 2.9 and is visualised for $n = 2$ in Fig. 2.3.

Algorithm 2.9: POLARIZATION

Input: $\mathbf{m} \in \{0,1\}^n$
if $n = 1$ **then**
  Output: $\mathbf{m}$
else
  for $i \leftarrow 1$ **to** $\lfloor \frac{n}{2} \rfloor$ **do**
    $(\mathbf{m}[2i-1], u[2i]) = (\mathbf{m}[2i-1] \oplus \mathbf{m}[2i], \mathbf{m}[2i])$
  end
  $\mathbf{c} = \text{POLARIZATION}(\mathbf{m}[o]) + \text{POLARIZATION}(\mathbf{m}[e])$
end
Output: $\mathbf{c}$

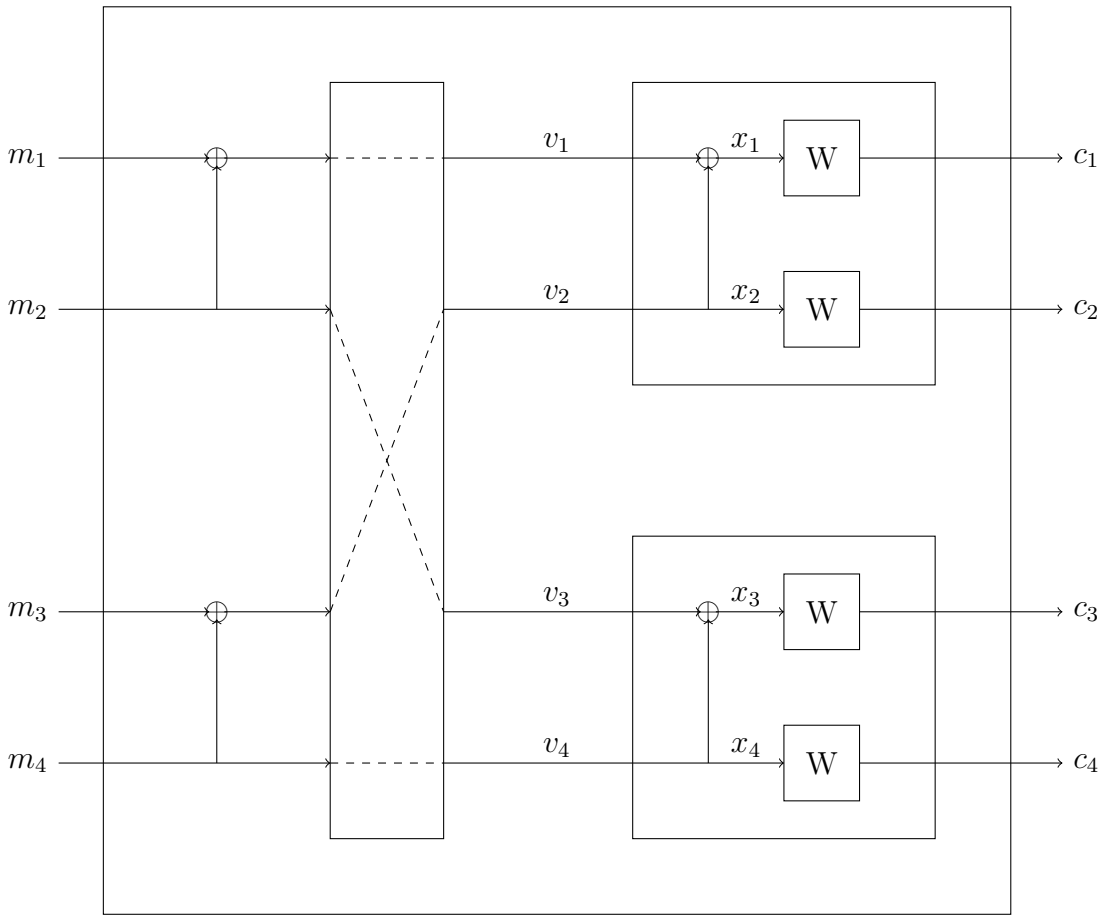

Figure 2.3: Polarization.

Encoding

To encode a message $\mathbf{m}$ of length k we first create a new vector $\mathbf{u}$ of length $N = 2^n$ for some $2^n \geq m$. We divide the bits of $\mathbf{u}$ into two groups, a frozen set, of size $2^n - m$ which is the bits with the lowest channel capacity, and a data set, of size m which is

the bits with the highest channel capacity. We then consider each bit of $\mathbf{u}$ in turn, setting it to 0 if it is a frozen bit or setting it to the value of the next message bit if it is a data bit. After setting the vector $\mathbf{u}$, we then transform it into the codeword $\mathbf{c}$ by apply the Polarization operation. E.g. if we want to encode the message $\mathbf{m} = [1, 1, 0, 0]$ as an 8 bit codeword, then the frozen set will be bits $\{1, 2, 3, 5\}$ and the data bits will be $\{4, 6, 7, 8\}$. This gives a vector $\mathbf{u} = [0, 0, 0, 1, 0, 1, 0, 0]$, which is polarised to give $\mathbf{c} = [0, 1, 1, 0, 0, 1, 1, 0]$. To determine which bits are frozen, we calculate the capacities of each bit and order them by capacity - the bits that have the lowest capacity are then frozen.

Decoding

The decoding algorithm (Algorithm 2.10) takes a received vector $\mathbf{r}$ and a vector of base likelihood ratios, LR, for each bit. The likelihood ratio for each bit, $LR[i]$, is $\frac{P(\mathbf{c}[i]=0|\mathbf{r}[i])}{P(\mathbf{c}[i]=1|\mathbf{r}[i])}$. The decoding algorithm works by iterating over each bit of $\mathbf{r}$, if $\mathbf{r}[i]$ is a frozen bit then $\hat{\mathbf{c}}[i]$ is set to 0, if it is a data bit then we recursively calculate the likelihood ratio $L_N[i](\mathbf{r}[1:N], \hat{\mathbf{c}}[1:i-1])$ using Algorithm 2.11. The new likelihood ratio is calculated differently for each recursion based on the value of i. If i is odd then

$$L_N[2i-1](\mathbf{r}[1:N], \hat{\mathbf{c}}[1:2i-2]) =$$

$$\frac{L_{N/2}[i](\mathbf{r}[1:N/2], \hat{\mathbf{c}}[1:2i-2]_o \oplus \hat{\mathbf{c}}[1:2i-2]_e) L_{N/2}[i](\mathbf{r}[N/2+1:N], \hat{\mathbf{c}}[1:2i-2]_e) + 1}{L_{N/2}[i](\mathbf{r}[1:N/2], \hat{\mathbf{c}}[1:2i-2]_o \oplus \hat{\mathbf{c}}[1:2i-2]_e) + L_{N/2}[i](\mathbf{r}[N/2+1:N], \hat{\mathbf{c}}[1:2i-2]_e)}.$$

If i is even then

$$L_N[2i](\mathbf{r}[1:N], \hat{\mathbf{c}}[1:2i-1]) =$$

$$L_{N/2}[i](\mathbf{r}[1:N/2], \hat{\mathbf{c}}[1:2i-1]_o \oplus \hat{\mathbf{c}}[1:2i-1]_e)^{1-2\hat{\mathbf{c}}[2i-1]} L_{N/2}[i](\mathbf{r}[N/2+1:N], \hat{\mathbf{c}}[1:2i-1]_e).$$

Intuitively, if i is odd then it calculates the ratio between the probability of both bits being the same versus both bits being different. If i is even then it calculates the ratio between either the ratio of the probability of both bits being 0 versus both bits being 1 or the ratio of the probability of the first bit being 1 and the second bit being 0 versus the probability of the first bit being 0 and the second

Algorithm 2.10: DECODING

Input: $\mathbf{r}, p$

$\hat{\mathbf{c}} \leftarrow [\,]$
for $i \leftarrow 1$ **to** N **do**
  $r \leftarrow \mathrm{L}(i, N, \mathbf{r}, \hat{\mathbf{c}}, p)$
  if $r \geq 1$ **then**
    $\hat{\mathbf{c}}[i] \leftarrow 0$
  else
    $\hat{\mathbf{c}}[i] \leftarrow 1$
  end
end

Output: $\hat{\mathbf{c}}$

Algorithm 2.11: L

Input: $i, N, \mathbf{r}, \hat{\mathbf{c}}, p$

if $n = 1$ **then**
  Output: $p[i]$
else
  $i' \leftarrow (i+1)//2$
  $a \leftarrow \mathrm{L}(i', N/2, \mathbf{r}[1 : N/2], \hat{\mathbf{c}}[1 : i-1]_o \oplus \hat{\mathbf{c}}[1 : i-1]_e, p)$
  $b \leftarrow \mathrm{L}(i', N/2, \mathbf{r}[N/2+1 : N], \hat{\mathbf{c}}[1 : i-1]_e, p)$
  if i is odd **then**
    $r \leftarrow \frac{ab+1}{a+b}$
  else
    $r \leftarrow a^{1-2\hat{\mathbf{c}}[i-1]}b$
  end
end

Output: r

being 1. If $L_N[i](\mathbf{r}[1 : N], \hat{\mathbf{c}}[1 : i-1]) < 1$ then $\hat{\mathbf{c}}[i] = 1$ else $\hat{\mathbf{c}}[i] = 0$. To get the received message $\hat{\mathbf{m}}$, we take the non-frozen bits of $\hat{\mathbf{c}}$. This recursive algorithm has a runtime of $O(N^2)$ however when using simple memoisation techniques, it is reduced to $O(N \log N)$ [Ari09]. As long as the rate of the polar code is below the capacity of the channel, then the probability of the decoder failing is $O(N^{-\frac{1}{4}})$. For Polar codes to be utilised for error correction by a cryptosystem, the cryptosystem needs to be considered as a channel, the capacity of the channel needs to be calculated and the channel polarization needs to be calculated. There has been some work on this already [WL21], however it has not been generalised beyond R-LWE.

CHAPTER 3

Using error correction for post quantum cryptography

Compared to classical public key cryptosystems, LWE based cryptosystems produce fairly large ciphertexts. Kyber and Saber both increase the size of the message by a factor of between 23 and 49 during encryption. Frodo however has a much larger blowup, at between 608-676 times the size of the input. This, combined with other performance issues, is why it is deemed to not be generally applicable and why it has not been standardised [AASA+20, AAC+22].

Frodo is designed to be as conservative as possible. Two main design decisions heavily contribute to the large increase in the size of the ciphertexts for Frodo over other lattice based cryptosystems. The first decision was to use plain LWE problem as the basis for Frodo. Although the structure of Ring-LWE (R-LWE) [LPR10] or Module-LWE (MLWE) [LS12] allows for much smaller ciphertexts, these problems are much newer and much less studied whilst LWE is one of the most studied problems in cryptography [AASA+19]. The second decision is to be conservative with the decryption failure rate, δ_{ct}.

In this chapter we examine the noise in LWE-based cryptosystems and how to manage it, as well as opening a discussion into how conservative cryptosystems need to be with the decryption failure rate. We also look at different methods of reducing

the size of the ciphertexts produced by LWE-based cryptosystems, including ones adapted for different use cases. We combine this with methods of calculating the decryption failure rate to allow us to recommend new parameter sets for Frodo that are up to 13% smaller without any loss of security and still remain conservative in terms of decryption failure rate. We also recommend parameter sets for situations that have specific requirements such as a negligible failure rates or varying message sizes whilst also being at least as secure as Frodo and as small as possible, opening Frodo up to become more widely applicable.

This chapter starts by discussing aspects of cryptosystem design in more detail and specifically discusses the methods by which noise is added to the cryptosystems in Section 3.1. We analyse methods for calculating the decryption failure rates in Section 3.2. In Section 3.3 we study the use of error correction, multi-bit encryption and rounding to reduce the size of the ciphertext. We then recommend parameter sets for different applications of LWE-based cryptosystems in Section 3.4.

The key novel contributions of this chapter are the use of Gray codes to reduce the number of bit errors when multi-bit encryption techniques are used, the full analysis of how various techniques could be applied to current KEMs (rather than to just a general scheme) with scripts to enable researchers to find improved parameter sets from a given starting point, and to provide specific parameter sets for these KEMs.

3.1 Secrets, errors and rounding

The decryption failure rate is a contributing factor towards the size of the ciphertexts, we now discuss the noise that causes decryption failures and how different parts of the cryptosystem contribute to it. We split the noise into its three key components: secrets, errors and rounding. Looking at the general LWE equation (Section 2.3.4) of $\mathbf{B} = \lfloor \mathbf{A} \cdot \mathbf{S} + \mathbf{E} \rceil_{q \to p}$, we refer to $\mathbf{S}$ as the secret, $\mathbf{E}$ as the error and $q \to p$ as the rounding. In the general algorithm the secrets are $\mathbf{S}$ in KeyGeneration (Algorithm 2.1) and $\mathbf{R}$ in Encryption (Algorithm 2.2), the errors are $\mathbf{E}_A$ in KeyGeneration and $\mathbf{E}'_B$ and $\mathbf{E}''_B$ in Encryption, and the rounding is $q \to p$ in KeyGeneration, and $q \to p$ and $q \to t$ in Encryption.

The secrets are typically both drawn from the same distribution, denoted χ_s, and the errors are similarly all drawn from the same distribution, denoted χ_e. Often χ_s and χ_e are the same distribution, as is the case with all cryptosystems in this thesis, but they don't need to be [CPL+17], and our analysis does not rely on this. The rounding is often by different amounts, especially between the two halves of the ciphertext, $\mathbf{B}'$ and $\mathbf{V}'$.

The round 3 LWE-based cryptosystems use the centred binomial distribution and the centred discrete Gaussian distribution for χ_s and χ_e. We denote the centred binomial distribution as Ψ_k, $k \in \mathbb{Z}^+$. It is a discrete, symmetrical distribution centred around 0 with a variance of $\frac{k}{2}$. For $X \sim \Psi_k$, X can take any integer value in the range $[-k, ..., k]$, with $P(X = x) = \binom{2k}{k+x} \cdot 2^{-2k}$.

The discrete centred Gaussian distribution, $N_\alpha(\sigma)$, is realised by sampling from a discrete distribution over $[-\alpha, \alpha]$ that approximates a Gaussian with standard deviation σ. For $X \sim N_\alpha(\sigma)$, x can take any value in $[-\alpha, \alpha]$ with

$$P(X = x) = \frac{1}{\sum_{k=-\infty}^{\infty} e^{\frac{-k^2}{2\sigma^2}}} e^{\frac{-x^2}{2\sigma^2}}.$$

See [BGPT19] for an overview of the error and secret distributions that can be utilised by LWE-based cryptosystems.

Rounding from q to p is done by discretizing each coefficient in the ring $\mathbb{Z}_q$ to the ring $\mathbb{Z}_p$. The noise added to the cryptosystem by rounding can be modelled as an almost uniform distribution, denoted $U_{q \to p}$, over the range $\left[-\lceil \frac{q}{2p} \rceil, ..., \lceil \frac{q}{2p} \rceil\right]$ with

$$P(X = x) = \begin{cases} \dfrac{p}{2q} & x = -\lceil \frac{q}{2p} \rceil \text{ or } \lceil \frac{q}{2p} \rceil, \\ \\ \dfrac{p}{q} & \text{otherwise.} \end{cases}$$

The exact distributions used by the cryptosystems can be found in Table 3.1.

- Frodo uses the LWE problem for its hardness, and so uses $\mathbf{B} = \mathbf{A} \cdot \mathbf{S} + \mathbf{E}$, where $\mathbf{A} \in \mathbb{Z}_q^{n \times n}$ and $\mathbf{B}, \mathbf{S}, \mathbf{E} \in \mathbb{Z}_q^{n \times \bar{n}}$. The ciphertext for Frodo is $(\mathbf{B}', \mathbf{V}')$, where $\mathbf{B}' \in \mathbb{Z}_q^{\bar{m} \times n}$ and $\mathbf{V}' \in \mathbb{Z}_q^{\bar{m} \times \bar{n}}$.

Cryptosystem	χ_s/χ_r	$\chi_e/\chi_{e'}$	Rounding $\mathbf{B}$	Rounding $\mathbf{B}'$	Rounding $\mathbf{V}'$
Frodo-640	$\frac{N_{12}(2.8)}{N_{12}(2.8)}$	$\frac{N_{12}(2.8)}{N_{12}(2.8)}$	N/A	N/A	N/A
Frodo-976	$\frac{N_{10}(2.3)}{N_{10}(2.3)}$	$\frac{N_{10}(2.3)}{N_{10}(2.3)}$	N/A	N/A	N/A
Frodo-1344	$\frac{N_6(1.4)}{N_6(1.4)}$	$\frac{N_6(1.4)}{N_6(1.4)}$	N/A	N/A	N/A
Kyber-512	Ψ_3/Ψ_3	Ψ_3/Ψ_2	N/A	$U_{3329\to 2^{10}}$	$U_{3329\to 2^4}$
Kyber-768	Ψ_2/Ψ_2	Ψ_2/Ψ_2	N/A	$U_{3329\to 2^{10}}$	$U_{3329\to 2^4}$
Kyber-1024	Ψ_2/Ψ_2	Ψ_2/Ψ_2	N/A	$U_{3329\to 2^{11}}$	$U_{3329\to 2^5}$
(Light)Saber	Ψ_5/Ψ_5	N/A	$U_{2^{13}\to 2^{10}}$	$U_{2^{13}\to 2^{10}}$	$U_{2^{10}\to 2^3}$
Saber	Ψ_4/Ψ_4	N/A	$U_{2^{13}\to 2^{10}}$	$U_{2^{13}\to 2^{10}}$	$U_{2^{10}\to 2^4}$
(Fire)Saber	Ψ_3/Ψ_3	N/A	$U_{2^{13}\to 2^{10}}$	$U_{2^{13}\to 2^{10}}$	$U_{2^{10}\to 2^6}$

Table 3.1: The secrets, errors and rounding for each LWE-based scheme.

- Kyber uses the MLWE problem for its hardness, and so uses $\mathbf{B} = \mathbf{A} \cdot \mathbf{S} + \mathbf{E}$, where $\mathbf{A} \in R_q^{k\times k}$ and $\mathbf{B}, \mathbf{S}, \mathbf{E} \in R_q^{k\times 1}$, however they do use rounding for compression. The ciphertext for Kyber is $(\mathbf{B}', \mathbf{V}')$, where $\mathbf{B}' \in R_p^{k\times 1}$ and $\mathbf{V}' \in R_t$.

- Saber uses the MLWR problem for its hardness, and so uses $\mathbf{B} = \lfloor \mathbf{A} \cdot \mathbf{S} \rceil_{q\to p}$, where $\mathbf{A} \in R_q^{k\times k}$, $\mathbf{S} \in R_q^{k\times 1}$, and $\mathbf{B} \in R_p^{k\times 1}$. The ciphertext for Saber is $(\mathbf{B}', \mathbf{V}')$, where $\mathbf{B}' \in R_p^{k\times 1}$ and $\mathbf{V}' \in R_t$.

3.2 Calculating decryption failure rates

In calculating the decryption failure rate we assume that each coefficient is independent. It has been shown that this assumption is not valid in practice [MFS20]. However the error that results is believed to be small [MFS20]. We therefore calculate the decryption failure rate assuming independence and take this into account when recommending parameter sets in Section 3.4.

3.2.1 Calculating $\delta_{\mathbf{bit}}$

The failure rate can be calculated from knowledge of the noise in the cryptosystem. We start by writing the rounding errors in terms of $\mathbf{B}, \mathbf{B}', \mathbf{S}, \mathbf{A}$ and $\mathbf{E}$.

$$\mathbf{U}_A = \mathbf{A} \cdot \mathbf{S} + \mathbf{E}_A - \mathbf{B} \qquad \text{Algorithm 2.1}$$

$$\mathbf{U}'_B = \mathbf{A}^\top \cdot \mathbf{R} + \mathbf{E}'_B - \mathbf{B}' \qquad \text{Algorithm 2.2}$$

$$\mathbf{U}''_B = \mathbf{B}^\top \cdot \mathbf{R} + \mathbf{E}''_B + \text{enc}(\mathbf{m}) - \mathbf{V}' \qquad \text{Algorithm 2.2}$$

The total noise in the cryptosystem is then

$$\begin{aligned}\mathbf{F} &= \mathbf{V}' - \mathbf{V}^\top - \frac{q}{2}\text{enc}(\mathbf{m}), \\ &= (\mathbf{E}_A + \mathbf{U}_A)^\top \cdot \mathbf{R} - \mathbf{S}^\top \cdot (\mathbf{E}'_B + \mathbf{U}'_B) + (\mathbf{E}''_B + \mathbf{U}''_B).\end{aligned}$$

To compute the distribution of possible total noise, χ_F, we recall that all terms in this expression are random variables drawn from either χ_s, χ_e or the rounding distributions defined in Section 3.1. The combined distribution is then

$$\chi_F = (((\chi_{e'} * U_{q\to p}) \cdot \chi_r)*^n) * ((\chi_s \cdot (\chi_e * U_{q\to p'}))*^n) * (\chi_{e'} * U_{p\to t}). \tag{3.1}$$

Where $\cdot$ represents taking the product, $*$ represents taking the convolution, and $*^n$ represents taking the convolution of a distribution with itself n times. The bit decryption failure rate, δ_{bit}, is then

$$\delta_{\text{bit}} := P\left(|f| > \frac{q}{4}\right), f \sim \chi_F. \tag{3.2}$$

The codeword decryption failure rate, δ_{ct}, for a message of length $|m|$, is determined from δ_{bit} as such:

$$\delta_{\text{ct}} = 1 - (1 - \delta_{\text{bit}})^{|m|}.$$

If an error correcting code that can correct c errors is used, then for an encoded message of length $|m|$, the failure rate is reduced to

$$\delta_{\text{ct}} = 1 - \sum_{i=0}^{c} \binom{|m|}{i} \delta_{\text{bit}}^i (1 - \delta_{\text{bit}})^{|m|-i}. \tag{3.3}$$

3.2.2 Failure boosting and attacks based on the failure rate

Failure boosting [DVV18, DGJ+19] is a process that uses precomputation to find message seeds with a higher chance of causing a decryption failure. We briefly summarize the approach here, more detail can be found in [DVV18]. The noise in the cryptosystem can be split into three parts based on when it is added, the noise added during KeyGeneration (Algorithm 2.1)

$$\mathbf{N}_K = \begin{pmatrix} -\mathbf{S} \\ \mathbf{E}_A + \mathbf{U}_A \end{pmatrix},$$

the noise added to the first half of the ciphertext ($\mathbf{B}'$)

$$\mathbf{N}_{B'} = \begin{pmatrix} \mathbf{E}'_B + \mathbf{U}'_B \\ \mathbf{R} \end{pmatrix},$$

and the noise added to the second half of the ciphertext ($\mathbf{V}'$)

$$\mathbf{N}_{V'} = \mathbf{E}''_B + \mathbf{U}''_B.$$

We can then write the noise in the cryptosystem as $\mathbf{F} = \mathbf{N}_K^\top \cdot \mathbf{N}_{B'} + \mathbf{N}_{V'}$. The precomputation is done by picking a threshold failure probability, f_t. A pair $(\mathbf{N}_{B'}, \mathbf{N}_{V'})$ with a δ_{bit} that is greater than f_t is considered a weak pair. Due to the use of the FO-transform, the pair $(\mathbf{N}_{B'}, \mathbf{N}_{V'})$ is generated pseudorandomly from a seed, so a bruteforce search of the seeds has to be done. The probability that a given seed generates a $(\mathbf{N}_{B'}, \mathbf{N}_{V'})$ pair with a failure probability greater than f_t is denoted by α. When a weak pair is used, we have a boosted decryption failure rate $\beta_{\text{bit}} = P(|\mathbf{F}| > \frac{q}{4} | (\mathbf{N}_{B'}, \mathbf{N}_{V'})) > f_t$, where $\beta_{\text{bit}} > \delta_{\text{bit}}$. This yields an estimated $\alpha^{-1}\beta_{\text{bit}}^{-1}$ work to find one failure. Failure boosting can be improved on by conditioning further with repeated failures [DRV20].

3.2.3 Calculating a target decryption failure rate

Having discussed how the decryption failure rate can be boosted, we now look at how this impacts the security of $\mathbf{N}_K$. We start by looking at what information can be gained from having a plaintext that causes a decryption failure . This information is referred to as a hint, and it typically takes the form of a coefficient of $\mathbf{N}_K$ having a specific value or lower bounding the magnitude of a coefficient. How these hints are specifically generated can be found in [DRV20, §4]. Having established these hints, they can be used to produce a version of the LWE instance where some secret values are known, this can then be attacked using standard methods, an overview of which can be found in [APS15]. The security of the reduced LWE instance can be calculated using either [ACD+18] or [DDGR20]. We refer to the security of the reduced problem as $\mathbf{N}_{K-\text{Simplified}(i)}$, where there have been i decryption failures. The overall cost of this attack is $\alpha^{-1}\beta_{\text{bit}}^{-1}\mathbf{N}_{K-\text{Simplified}(i)}$.

NIST suggests that for an attack to be considered feasible it should utilise at most 2^{64} chosen ciphertexts, applying the limit to the decryption failure attack, we bound $\beta_{\text{bit}}^{-1}i < 2^{64}$. If we want to be conservative and allow the attacker to achieve at most one decryption failure, then we require the boosted failure rate $\beta_{\text{ct}} < 2^{-64}$ and ideally for the overall work to find one failure ($\alpha^{-1}\beta_{\text{bit}}^{-1}$) to be greater than the desired security level of 2^{λ}.

3.3 Reducing the size of the ciphertexts

The primary mechanisms for reducing the size of ciphertexts in LWE-based KEMs, a reduction of n or q, come at the expense of increasing δ_{ct}, and therefore reducing security. By careful use of error correcting codes, employing suitable multi-bit encryption and utilising rounding, we can reduce the size of the ciphertext while controlling δ_{ct}.

3.3.1 Error correcting codes

Using error correcting codes doesn't directly reduce the size of the ciphertexts, however it does allow for other techniques to be used that reduce the ciphertext size

Name	n	q	p	Error Correction	|pt|	δ_{bit}	δ_{ct}	|ct|
LAC-128	512	251	2^4	BCH(255,128,17) + D2	128	$2^{-22.26}$	2^{-151}	712
LAC-128 (no ECC)	512	2^{10}	2^5	-	128	$2^{-169.23}$	2^{-162}	720
LAC-192	1024	251	2^4	BCH(511,256,17) + D2	256	$2^{-42.24}$	2^{-324}	1352
LAC-192 (no ECC)	1024	2^{10}	2^6	-	256	N/A	N/A	1472
LAC-256	1024	251	2^4	BCH(511,256,41) + D2	256	$2^{-20.01}$	2^{-302}	1464
LAC-256 (no ECC)	1024	2^{11}	2^3	-	256	N/A	N/A	1504

Table 3.2: The current LAC parameter sets and adapted versions without error correction (the adapted versions are less secure than their original counterparts).

without any impact on δ_{ct}. As error correcting codes introduce some redundancy, their use in reducing δ_{ct} does have diminishing returns. LAC, which was not selected for third round of the NIST post quantum cryptography (PQC) standardisation process, is a prime example of how error correction can help reduce the overall ciphertext size. Taking an alternate parameter set for LAC where no error correction is used but instead has a large enough modulus (q) so that δ_{bit} is small enough that δ_{ct} remains the same, we can see that the use of error correcting codes has led to a 8% size reduction in Table 3.2.

None of the cryptosystems in the third round use error correcting codes, however we will briefly mention some of the error correcting codes used in cryptosystems from the second round. The simplest error correcting code is the repetition code, where each bit is repeated $n-1$ times, which corrects up to $\lfloor \frac{n-1}{2} \rfloor$ errors. This is commonly extended to include soft decoding, LAC refers to the n-soft repetition code as Dn. This soft repetition code is used by NewHope and LAC, the latter of which is a candidate in another PQC standardization project [XD19], and whilst the soft repetition code used is often not enough to correct errors, it does still dramatically reduce the probability of a decryption failure by aggregating the result of several coefficients. Whilst no other soft error correcting codes are used, a good survey on how they could be used with NewHope can be found in [FPS19].

Other error correcting codes in use include BCH codes [BRC60] (LAC [LLJ+19]), Melas codes (Three Bears [Ham19]) and XEf codes [Saa17] (Round5 [GZB+19]). Although we won't discuss how the codes work, we provide a comparison of them in Table 3.3. Polar codes have also been suggested for R-LWE schemes [WL21],

Code	n	k	d	c	Rate
XEf with $\kappa = 128$	318	128	11	5	0.40
XEf with $\kappa = 192$	410	192	11	5	0.47
XEf with $\kappa = 256$	490	256	11	5	0.52
Melas	274	256	5	2	0.93
BCH(255,128,17)	192	128	17	8	0.67
BCH(511,256,17)	328	256	17	8	0.78
BCH(511,256,41)	427	256	41	20	0.60

Table 3.3: Overview of error correcting codes used by LWE-based cryptosystems. Where n is the size of the message after encoding by the error correcting code, k is the size of the plaintext, d is the minimum distance between codewords, c is the amount of errors that can be corrected, and rate is the fraction of the bits that form the plaintext.

although no schemes have made use of them. We propose that BCH codes should be used by LWE-based cryptosystems, as they are both space efficient and admit a constant time implementation [WR19].

3.3.2 Multi-bit encryption

Method

One method of increasing the throughput of the scheme is to use multi-bit encryption, where instead of encrypting one bit in each coefficient, multiple bits are encoded in each coefficient. This method is currently employed in cryptosystems that utilize unstructured lattices, such as Frodo [NAB+20].

This is done by changing the alphabet used to represent the message from $\{0, 1\}$ to $\{0, 1, ..., 2^b - 1\}$ for some $b \in \mathbb{Z}^+$, where $b = 1$ is the standard case. The Encryption (Algorithm 2.2) and Decryption (Algorithm 2.3) remain broadly the same, so we only highlight the differences.

During encryption, the message is normally added to the ciphertext as a fixed length number of symbols from $\{0, 1\}$ and multiplied by $\frac{q}{2}$. In Algorithm 2.2 in the generic scheme this is

$$\mathbf{V}' = \lfloor \mathbf{B}^\top \cdot \mathbf{R} + \mathbf{E}''_B + \frac{q}{2}\text{enc}(\mathbf{m}) \rceil_{q \to t}.$$

This is changed to

$$\mathbf{V}' = \lfloor \mathbf{B}^\top \cdot \mathbf{R} + \mathbf{E}''_B + \frac{q}{2^b}\text{enc}(\mathbf{m}) \rceil_{q \to t}.$$

Likewise, decryption changes from rounding from q to 2, to rounding from q to 2^b. This reduces the number of coefficients that are required to represent the message, thus reducing the size of $\mathbf{V}'$. Whilst decreasing the size of the ciphertext, this does also increase the decryption failure rate, as the threshold for which the noise causes a failure is decreased. Where before the decryption failure rate was

$$\delta_{\text{bit}} = P\left(|f| > \frac{q}{4}\right), f \sim \chi_F,$$

with multi-bit encryption it is

$$\delta_{\text{bit}} = P\left(|f| > \frac{q}{2 \cdot 2^b}\right), f \sim \chi_F.$$

Multi-bit encryption reduces the size of $\mathbf{V}'$ by a factor of b at the cost of decreasing the failure threshold by a factor of 2^{b-1}. Therefore it tends to work best for LWE-based cryptosystems where both halves of the ciphertext ($\mathbf{B}'$ and $\mathbf{V}'$) are similar in size than in cryptosystems where the second half of the ciphertext $\mathbf{V}'$ is much smaller, such as in M-LWE/M-LWR based cryptosystems.

Current usage

Frodo currently encodes 2, 3 and 4 bits into each coefficient for Frodo-640, Frodo-976 and Frodo-1344 respectively. This leads to a size reduction of 20-25% compared to equivalent parameter sets which don't use multi-bit encryption, Table 3.4 compares suggested parameter sets without multi-bit encryption, with the smaller original ones.

Use of Gray codes to reduce failures

Since $\frac{q}{2 \cdot 2^b}$ is already several standard deviations away, for example for Kyber-512 $q/2$ is approximately 83 standard deviations from the mean, χ_F, $P(|f| > \frac{3q}{2 \cdot 2^b})$ is

Scheme	n	$\bar{n}$	$\bar{m}$	p	b	$\lvert ct \rvert$(Bytes)
Frodo-640	640	8	8	2^{15}	2	9720
Frodo-640 without MB	640	12	11	2^{14}	1	12551
Frodo-976	976	8	8	2^{16}	3	15744
Frodo-976 without MB	976	14	14	2^{14}	1	24255
Frodo-1344	1344	8	8	2^{16}	4	21632
Frodo-1344 without MB	1344	16	16	2^{13}	1	35360

Table 3.4: Frodo parameter sets with and without the use of multi-bit encryption

negligible. This means that in our analysis, we assume that when a decryption failure occurs, a coefficient is only incorrectly decrypted as one of its neighbours, i.e. that

$$\frac{q}{2 \cdot 2^b} < |f| < \frac{3q}{2 \cdot 2^b}, f \sim \chi_F.$$

As b increases, the threshold at which a decryption failure occurs decreases, leading to a higher decryption failure rate for each coefficient. When $b > 1$ we not only have the possibility of a failure happening in any given coefficient, but also the possibility of a failure in more than one bit. An example of this for $b = 2$ is if 2 is incorrectly decrypted as 3 then there is a failure only in the least significant bit, however if a 2 is incorrectly decrypted as 1 then there is a failure in both bits. Since error correction operates at the bit level, we care about this difference. To reduce the impact of each failure we can ensure that each failure only causes a failure in one bit by encoding each coefficient with a Gray code [Gra]. A Gray code maps integers to binary representations such that any two consecutive integers differ by only one bit. As we will be dealing with mostly small values of b, we have provided the first few values for a Gray code.

0 = 0000	1 = 0001	2 = 0011	3 = 0010
4 = 0110	5 = 0111	6 = 0101	7 = 0100
8 = 1100	9 = 1101	10 = 1111	11 = 1110
12 = 1010	13 = 1011	14 = 1001	15 = 1000

The expected number of bit errors for one coefficient failure without using Gray

code is

$$\left(\sum_{i=1}^{b-1} i\frac{1}{2^i}\right) + b\frac{2}{2^b} = 2 - \frac{2}{2^b}$$

For $b = 2$ the expected number of errors is 1.5, and as b tends towards ∞ the expected number of errors tends to 2. By using Gray code we fix the expected number of failures to 1, a reduction of between a third and a half depending on the value of b in use.

Since Frodo is based on LWE rather than M-LWE, we propose that the use of multi-bit encryption should be expanded as much as possible with Frodo and that it should incorporate the use of Gray codes.

3.3.3 Rounding

The most commonly used method for reducing the size of the ciphertext is to introduce rounding. We discussed it's impact on the decryption failure rate in Section 3.1, and now focus on how it can impact the ciphertext size. This can be done to both halves ($\mathbf{B}'$ and $\mathbf{V}'$) or just to $\mathbf{V}'$. For non-LWR-based cryptosystems, rounding is typically only done on $\mathbf{V}'$, since this has a much smaller impact on the decryption failure rate. As it has a low impact on the decryption failure rate, $\mathbf{V}'$ is often rounded quite severely, e.g. Kyber rounds from 2^{10} to 2^3. Rounding only reduces the size of the ciphertext by a logarithmic factor (e.g. rounding $\mathbf{V}'$ from 2^9 to 2^3 only reduces the size of $\mathbf{V}'$ by a factor of 3) and so does need to be done quite severely to be effective.

There is a trade off if using rounding, as more severe rounding is used to reduce the size of the ciphertext, more error correction is needed to reduce the impact of the rounding. This leads to a balancing act of using enough error correction to keep δ_{bit} under control, whilst also using enough rounding such that the size reduction from rounding is greater than the increase in size caused by the error correcting code. By utilising a BCH code we propose rounding $\mathbf{B}'$ a small amount and $\mathbf{V}'$ as much as possible to reduce the size of the ciphertext.

3.4 Proposed parameter sets

To find parameter sets that present minimally sized ciphertexts, we performed a sweep over all parameters. In doing this we found a minimal parameter set that is over 1KB smaller than the initial parameters and a parameter set with a more negligible failure rate ($\sim 2^{-1000}$) that are only $10 - 20\%$ larger than the initial parameter sets.

We briefly review the parameters for Frodo:

$|pt|$ represents the size of the plaintext,

q represents the initial modulus,

t represents the rounding modulus (only used in the latter half of the encryption),

n represents the matrix dimension,

$\bar{m} \times \bar{n}$ represents the size of the matrix used to encode the bits,

b represents the number of bits per coefficient,

c represents the number of errors corrected by the error correcting code in use,

ecc represents the number of bits added during the encoding of the error correcting code and,

$|m| = |\text{pt}| + ecc \leq b \cdot \bar{m} \cdot \bar{n}$ represents the overall length of the encoded message to be encrypted.

3.4.1 Parameter sweep

To generate the parameter space we took the original Frodo parameters and then varied the values of q from 10 to 16, p from 10 to q, t from 2 to p, b from 1 to 8, c from 0 to 30 - the most errors that can be corrected with $ecc < 256$. We then calculate the failure distribution using Eq. (3.1) and calculate δ_{ct} using Eqs. (3.2) and (3.3), and the security level of the parameter set (using [ACD+18]) rejecting all parameter sets that don't meet the same level of security as the initial parameter set.

As part of providing the code that performs the parameter sweep, we have also provided a test script that checks the correctness of the code.

Finally we have created an interactive plot of these parameter sets of failure rate against ciphertext size, allowing for filtering by the maximum amount of errors corrected, minimum decryption failure rates and other similar properties.

3.4.2 Utilising the plots

From looking at the parameter plots Figs. 3.1 to 3.3 we can see that there exists a range of parameter sets with different decryption failure rates whilst only differing in size by a small amount. This allows us to easily recommend parameter sets that are small but are also conservative in terms of both decryption failure rates and the amount of errors corrected. From the overall plot, we have also produced the Pareto front of the plot, and so can also easily suggest parameter sets with increasingly negligible failure rates without a dramatic increase in the size of the ciphertexts.

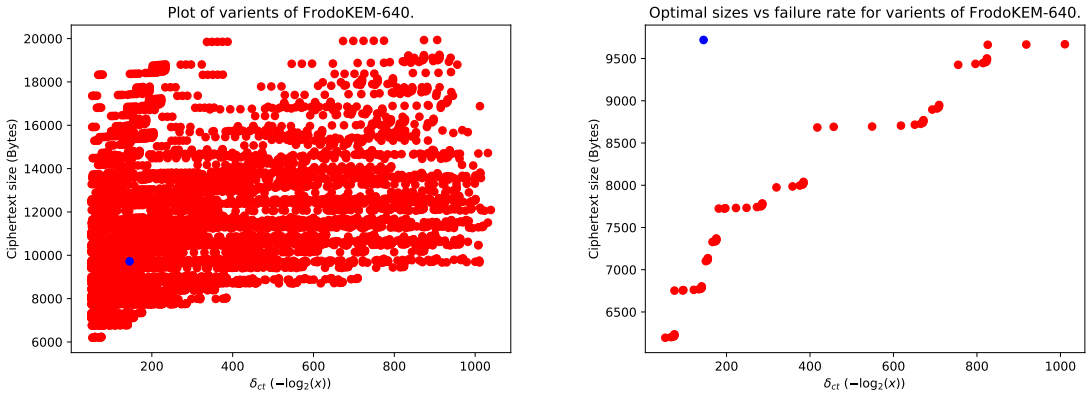

Figure 3.1: A plot of all parameter sets for FrodoKEM-640, and a plot of all minimal size parameter sets for each δ_{ct}, with the Frodo-640 parameter set highlighted in blue.

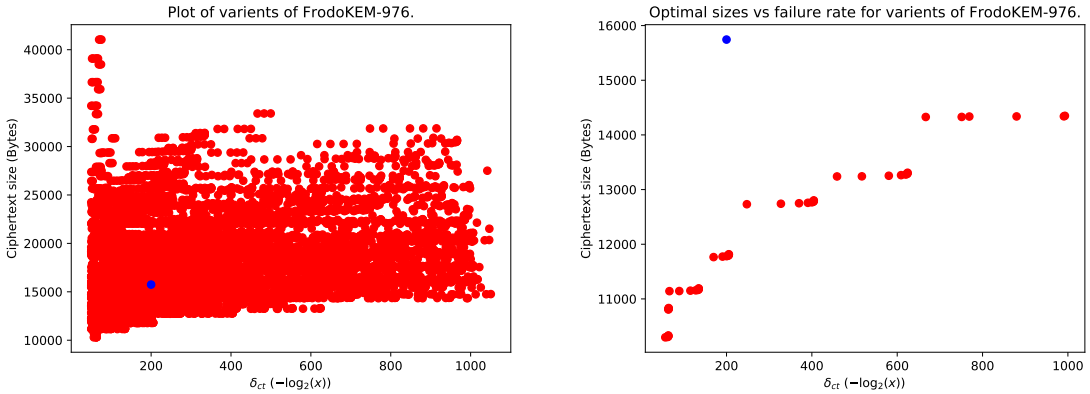

Figure 3.2: A plot of all parameter sets for FrodoKEM-976, and a plot of all minimal size parameter sets for each δ_{ct}, with the Frodo-976 parameter set highlighted in blue.

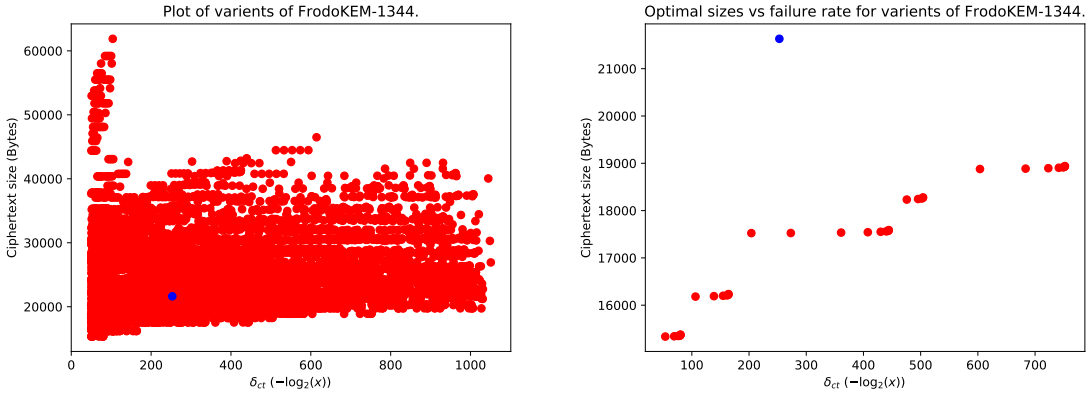

Figure 3.3: A plot of all parameter sets for FrodoKEM-1344, and a plot of all minimal size parameter sets for each δ_{ct}, with the Frodo-1344 parameter set highlighted in blue.

The nature of the plots allows for the further fine-tuning of the Frodo parameters for specific situations that have different constraints.

In Table 3.6 we recommend some parameter sets based on the initial Frodo parameter sets. For each parameter set we recommend two more fine-tuned parameter sets, one where we vary the error correction, multi-bit encryption used and the amount of rounding, and another where we also increase the amount of message bits contained in each coefficient.

	BIKE	ClassicMcEliece	HQC
I	1573	90	4481
III	3115	208	9026
IV	8246	208	14469

Table 3.5: The ciphertext size (in Bytes) for BIKE [ABB+22], Classic McEliece [ABC+22] and HQC [AAB+22] for level I, III and IV parameter sets.

In Table 3.5 we give the ciphertext sizes of the current code-based cryptosystems in Round 4 of the NIST PQC standardisation process. Whilst none of the lattice-based parameter sets that we propose come close to the efficiency of Classic McEliece, our fine-tuned Frodo parameter sets are comparable in size to HQC. Our fine-tuned Kyber and Saber parameter sets substantially outperform both HQC and BIKE.

Scheme	n	$\lvert\text{pt}\rvert$ (bits)	q	p	t	c	b	$\lvert\text{ct}\rvert$ (Bytes)	$\frac{\lvert\text{ct}\rvert}{\lvert\text{pt}\rvert}$	δ_{ct}
Frodo-640	640	128	2^{15}	2^{15}	2^{15}	0	2	9720	608	2^{-145}
with EC	640	128	2^{15}	2^{12}	2^{5}	1	2	7724	483	2^{-181}
with EC and MB	640	128	2^{15}	2^{11}	2^{7}	9	3	7102	444	2^{-150}
LowFailureRate	640	128	2^{15}	2^{11}	2^{11}	5	2	8040	503	2^{-384}
LowestFailureRate	640	128	2^{15}	2^{12}	2^{5}	10	2	9669	605	2^{-1010}
HigherFailureRate	640	128	2^{15}	2^{12}	2^{12}	4	3	6803	426	2^{-140}
Frodo-976	976	192	2^{16}	2^{16}	2^{16}	0	3	15744	656	2^{-200}
with EC	976	192	2^{16}	2^{12}	2^{8}	2	3	11782	491	2^{-200}
with EC and MB	976	192	2^{16}	2^{12}	2^{8}	2	3	11782	491	2^{-200}
LowFailureRate	976	192	2^{16}	2^{13}	2^{13}	2	3	12802	534	2^{-404}
LowerFailureRate	976	192	2^{16}	2^{13}	2^{7}	7	3	14349	598	2^{-992}
LowestFailureRate	976	192	2^{16}	2^{13}	2^{7}	8	3	14351	598	$< 2^{-1000}$
HighFailureRate	976	192	2^{16}	2^{12}	2^{6}	2	3	11765	491	2^{-169}
HigherFailureRate	976	192	2^{16}	2^{13}	2^{9}	3	4	11164	466	2^{-132}
Frodo-1344	1344	256	2^{16}	2^{16}	2^{16}	0	4	21632	676	2^{-253}
with EC	1344	256	2^{16}	2^{13}	2^{6}	3	4	17526	548	2^{-273}
with EC and MB	1344	256	2^{16}	2^{13}	2^{6}	3	4	17526	548	2^{-273}
LowFailureRate	1344	256	2^{16}	2^{13}	2^{13}	3	4	17588	550	2^{-444}
LowerFailureRate	1344	256	2^{16}	2^{14}	2^{14}	3	4	18941	592	2^{-752}
LowestFailureRate	1344	256	2^{16}	2^{13}	2^{6}	0	3	19721	617	$< 2^{-1000}$
HighFailureRate	1344	256	2^{16}	2^{13}	2^{3}	2	4	17498	547	2^{-204}
HigherFailureRate	1344	256	2^{16}	2^{12}	2^{4}	3	4	16164	506	2^{-154}

Table 3.6: Some fine-tuned Frodo parameter sets.

Scheme	n	$\lvert pt \rvert$ (bits)	q	p	t	c	b	$\lvert ct \rvert$ (Bytes)	$\frac{\lvert ct \rvert}{\lvert pt \rvert}$	δ_{ct}
Frodo-640 with 256 bit pt	640	256	2^{15}	2^{12}	2^5	7	3	9667	303	2^{-148}
With Lower Failure rate I	640	256	2^{15}	2^{12}	2^{11}	15	3	10741	336	2^{-451}
With Lower Failure rate II	640	256	2^{15}	2^{11}	2^5	16	2	12445	389	2^{-906}
Frodo-976 with 128 bit pt	976	128	2^{16}	2^{12}	2^5	4	3	10283	643	2^{-219}
With Lower Failure rate I	976	128	2^{16}	2^{13}	2^{13}	4	3	11192	700	2^{-678}
With Lower Failure rate II	976	128	2^{16}	2^{14}	2^{14}	4	3	12053	754	2^{-889}
Frodo-976 with 256 bit pt	976	256	2^{16}	2^{13}	2^6	8	4	14336	448	2^{-204}
With Lower Failure rate I	976	256	2^{16}	2^{13}	2^{13}	11	4	14419	451	2^{-411}
With Lower Failure rate II	976	256	2^{16}	2^{12}	2^{12}	8	3	14805	463	2^{-621}
With Lower Failure rate III	976	256	2^{16}	2^{13}	2^7	7	3	15954	499	2^{-990}
Frodo-1344 with 128 bit pt	1344	128	2^{16}	2^{13}	2^6	3	4	13134	821	2^{-276}
With Lower Failure rate I	1344	128	2^{16}	2^{13}	2^{12}	4	4	13166	823	2^{-560}
With Lower Failure rate II	1344	128	2^{16}	2^{14}	2^{14}	4	4	14184	887	2^{-945}
With Lower Failure rate III	1344	128	2^{16}	2^{15}	2^8	4	4	15161	948	2^{-1035}

Table 3.7: Some fine-tuned Frodo parameter sets for varying message size.

Scheme	n	k	p	q	t	c	b	$\lvert ct \rvert$ (Bytes)	$\frac{\lvert ct \rvert}{\lvert pt \rvert}$	δ_{ct}
Kyber-512	256	3/2	3329	2^{10}	2^4	0	1	768	24	2^{-139}
with EC	256	2	3329	2^8	2^2	12	1	603	19	2^{-144}
with EC and MB	256	2	3329	2^8	2^4	25	2	628	20	2^{-149}
Kyber-768	256	3	3329	2^{10}	2^4	0	1	1088	34	2^{-164}
with EC	256	3	3329	2^7	2^3	28	1	856	27	2^{-165}
with EC and MB	256	3	3329	2^8	2^4	27	2	889	28	2^{-168}
Kyber-1024	256	4	3329	2^{11}	2^5	0	1	1568	49	2^{-174}
with EC	256	4	3329	2^8	2^2	20	1	1131	25	2^{-184}
with EC and MB	256	4	3329	2^9	2^3	25	2	1239	39	2^{-186}
Light-Saber	256	2	2^{13}	2^{10}	2^3	0	1	736	23	2^{-120}
with EC	256	2	2^{13}	2^9	2^2	7	1	656	21	2^{-121}
with EC and MB	256	2	2^{13}	2^9	2^3	27	2	666	21	2^{-124}
Saber	256	3	2^{13}	2^{10}	2^4	0	1	1088	34	2^{-136}
with EC	256	3	2^{13}	2^8	2^3	23	1	939	29	2^{-140}
with EC and MB	256	3	2^{13}	2^9	2^4	20	2	971	30	2^{-141}
Fire-Saber	256	4	2^{13}	2^{10}	2^6	0	1	1472	46	2^{-165}
with EC	256	4	2^{13}	2^8	2^3	27	1	1205	38	2^{-167}
with EC and MB	256	4	2^{13}	2^9	2^4	25	2	1268	40	2^{-180}

Table 3.8: Some fine-tuned Kyber and Saber parameter sets, for 256 bit plaintexts.

CHAPTER 4

Securing linear algebra

Linear algebra forms the basis of many algorithms for decoding error correcting codes and decryption for code based cryptography , including for BCH codes - which we proposed for use in Chapter 3, and Reed-Solomon and Reed-Muller codes - which are used in HQC [AAB+22]. In order for these algorithms to be used in secure implementations, the underlying linear algebra algorithms need to be secured against side channel attacks. There are a number of gadgets in the literature already that have been shown to be t-NI or t-SNI secure. We briefly highlight each of these here and explain them.

We make use of the following gadgets:

G_1	SNIMul	a t-SNI([BBD+16]) secure algorithm for multiplication.
$G_{1\_}$	NIMul	a t-NI([BBP+16]) secure algorithm for multiplication.
$\hat{G_1}$	SNIAnd	a t-SNI([BBD+16]) secure algorithm for logical AND.
$\hat{G_{1\_}}$	NIAnd	a t-NI([BBP+16]) secure algorithm for logical AND.
G_2	SNIAdd	a t-SNI(Lemma 4.1.1) secure algorithm for addition.
$G_{2\_}$	NIAdd	a t-NI secure algorithm for addition.
$\hat{G_2}$	SNIXOR	a t-SNI(Lemma 4.1.1) secure algorithm for logical XOR.
$\hat{G_{2\_}}$	NIXOR	a t-NI secure algorithm for logical XOR.
G_3	SNIInv	a t-SNI([BBD+16]) secure algorithm for inverting an element of a field.
$G_{4\_}$	NISwap	a t-NI secure algorithm for swapping the values of two variables.
G_6	SNIGadget6	a t-SNI(Lemma 4.1.5) secure algorithm that is the inner loop of LUPDecomposition.
G_7	SNIGadget7	a t-SNI(Lemma 4.1.6) secure algorithm that is the middle loop of LUPDecomposition.
G_8	SNIMagnitudeComparator	a t-SNI(Lemma 4.1.3) secure algorithm for returning the shared result of $a \leq b$.
G_9	SNIRef	a t-SNI([BBD+16]) secure algorithm for mask refreshing.
G_{10}	SNILUPDecomp	a t-SNI(Section 4.1 Theorem 4.1.1) secure algorithm that returns the LUP Decomposition of a matrix.
G_{11}	SNILUPDeterminant	a t-SNI(Section 4.1 Theorem 4.1.2) secure algorithm that returns the determinant of a matrix.
G_{12}	SNIGadget12	a t-SNI(Lemma 4.1.7) secure algorithm that is the inner loop of LUPSolve.
G_{13}	SNILUPSolve	a t-SNI(Section 4.1 Theorem 4.1.3) secure algorithm that solves a series of linear equations.

Algorithm 4.1: SNIAdd

Input: $\mathbf{a}_0, \mathbf{b}_0 \in GF(2^n)$
Result: $\mathbf{c}_1 \in GF(2^n)$
1 **for** $i \leftarrow 1$ **to** $t+1$ **do**
2 $\quad \mathbf{c}_0^{(i)} \leftarrow (\mathbf{a}_0^{(i)} + \mathbf{b}_0^{(i)})$
3 **end**
4 $\mathbf{c}_1 \leftarrow \text{SNIRef}(\mathbf{c}_0)$
Output: $\mathbf{c}_1$

4.1 Masked matrix operations

The first matrix operation we try to mask is matrix decomposition, since it is often used as a component of other algorithms. In order to mask matrix decomposition, we break it down in a number of smaller gadgets, and also utilise a number of gadgets from the literature. The initial algorithm for finding the determinant using LUP Decomposition [GVL89, Chapter 2], which can be found in Algorithm 4.10, decomposes the matrix in place allowing the determinant to be calculated at the end by multiplying together the values in the diagonal. Our masked decomposition algorithm, which can be found in Algorithm 4.6, takes $\mathbf{M}$ as input, where $\mathbf{M}$ is the masked matrix M using Boolean masking, performs the decomposition in place and returns $\mathbf{M}$. Our masking works for any matrix over finite fields of characteristic 2. We have run experiments to validate the security of the masking for matrices over $GF(2^7)$ and $GF(2^8)$

We implement elements of $GF(2^n)$ as a polynomial of degree n, where each coefficient is either 0 or 1. In this implementation addition is equivalent to XOR and multiplication is equivalent to AND. This allows us to use Boolean masking, although we will still refer to each operation as addition and multiplication.

The first gadget we need is secure addition. Which we make by composing standard shared addition with a mask refresh, the algorithm for this is given in Algorithm 4.1.

Theorem 4.1.1 (t-SNI of Algorithm 4.1)**.** *Let* $\mathbf{a}_0^{(.)}$ *and* $\mathbf{b}_0^{(.)}$ *be the inputs and* $\mathbf{c}_1^{(.)}$ *be the output to* SNIAdd *where* $\boldsymbol{a}, \boldsymbol{b}$ *and* $\boldsymbol{c}$ *are all elements of* $GF(2^n)$*. For any set of* t' *intermediate variables and any subset* O *of output variables such that* $t' + |O| \leq t$*, there exists a subset* I *of input variables such that* $|I| \leq t'$ *and the* t'

intermediate variables and the output variables can be perfectly simulated using the I input variables.

Proof. It's clear to see that lines 1 to 3 of Algorithm 4.1 are t-NI, as the share $\mathbf{c}_0^{(i)}$ can be simulated using the corresponding input shares - $\mathbf{a}_0^{(i)}, \mathbf{b}_0^{(i)}$. As we then apply SNIRef to the shares before outputting, then by Proposition 1 the algorithm is t-SNI secure. □

4.1.1 Magnitude comparator

In this section we will consider a binary string $\mathbf{a}$ as an array, where $\mathbf{a}[0]$ is the highest order bit and $\mathbf{a}[1]$ is the lowest order bit. A magnitude comparator [CM07, Section 4.8] takes two binary strings, a and b, as input and, typically, has three outputs one for each of $a < b$, $a = b$, $a > b$. We give the pseudocode for this algorithm in Algorithm 4.2. However in this instance we only care about $a < b$, and so only consider the computation required for that output. We start by giving a 2 bit magnitude comparator (Algorithm 4.3), and then show how multiple comparators can be chained together to give an n bit magnitude comparator (Algorithm 4.4). Finally we give a gadget that converts coefficients of non-binary strings into individual bits (Fig. 4.3), allowing this construction to be extended to non-binary strings, and so can be used by all finite fields.

Algorithm 4.2: 2BitMagnitudeComparator as per [CM07]

Input: a,b

Result: Boolean values for $a > b, a = b, a < b$ respectively (r_0, r_1, r_2)

1 $x[0] \leftarrow \text{Or}(\text{And}(a[0], b[0]), \text{And}(\text{Not}(a[0]), \text{Not}(b[0])))$

2 $x[1] \leftarrow \text{Or}(\text{And}(a[1], b[1]), \text{And}(\text{Not}(a[1]), \text{Not}(b[1])))$

3 $r[0] \leftarrow \text{Or}(\text{And}(a[0], \text{Not}(b[0])), \text{And}(x[0], a[1], \text{Not}(b[1])))$

4 $r[1] \leftarrow \text{And}(x[0], x[1])$

5 $r[2] \leftarrow \text{Or}(\text{And}(\text{Not}(a[0]), b[0]), \text{And}(x[0], \text{Not}(a[1]), b[1]))$

Output: $(r[0], r[1], r[2])$

To create our secure magnitude comparator, we start by editing the standard algorithm to only return the first output, i.e. $a > b$, and rewrite it to use only (S)NIAnd, (S)NIXOR, (S)NINot. We realise the (S)NINot($\mathbf{a}$) gadget as (S)NIXOR($\mathbf{a}, \mathbf{1}$). We can then replace these gadgets with their secure counterparts. In order to argue about the security of the overall construction, we start by showing that the individual 2 bit comparator is secure.

Theorem 4.1.2 (t-SNI of SNI2BitMagnitudeComparatorWithCascade (SNI2BMCC)) *Let* $\mathbf{a}_0^{(.)}, \mathbf{b}_0^{(.)}, \mathbf{c}_0^{(.)}$ *be the inputs and* $\mathbf{r}_1^{(.)}$ *be the output to* SNI2BMCC*, where* $\boldsymbol{a}$ *and*

b *are bitstrings of length 2 and* ***c*** *is a single bit.. For any set of t' intermediate variables and any subset O of output variables such that $t' + |O| \leq t$, there exists a subset I of input variables such that $|I| \leq t'$ and the t' intermediate variables and the output variables can be perfectly simulated using the I input variables.*

Proof. We consider SNI2BMCC as a sequence of smaller gadgets in Fig. 4.1, where every gadget is either t-NI or t-SNI. The input c is only used once, and whilst the remaining inputs are used twice, they are only used by a t-NI gadget once - the remaining times being to a t-SNI version of SNIXOR. Only one intermediate variable is used more than once, and again it is only used once by a t-NI gadget. The last gadget used is also a t-SNI gadget, and so by Proposition 1 SNI2BMCC is also t-SNI secure. □

Algorithm 4.3: SNI2BitMagnitudeComparator

Input: $\mathbf{a}$,$\mathbf{b} \in [0,1]^2$

Result: Masked Boolean value $\mathbf{r} \in [0,1]$

1 $\mathbf{c}_0 \leftarrow \text{NINot}(\mathbf{b})$

2 $\mathbf{d}_0[0] \leftarrow \text{NIAnd}(\mathbf{a}[0], \mathbf{c}_0[0])$

3 $\mathbf{d}_0[1] \leftarrow \text{SNIXOR}(\mathbf{a}[0], \mathbf{b}[0])$

4 $\mathbf{d}_1[1] \leftarrow \text{NINot}(\mathbf{d}_0[0])$

5 $\mathbf{d}_2[1] \leftarrow \text{NIAnd}(\mathbf{d}_1[1], \mathbf{a}[1])$

6 $\mathbf{d}_3[1] \leftarrow \text{NIAnd}(\mathbf{d}_2[1], \mathbf{c}_0[0])$

7 $\mathbf{r}_0 \leftarrow \text{SNIXOR}(\mathbf{d}_0[0], \mathbf{d}_3[1])$

Output: $\mathbf{r}_0$

Algorithm 4.4: Sec2BitMagnitudeComparatorWithCascade

Input: $\mathbf{a}$,$\mathbf{b} \in [0,1]^2$,$\mathbf{c} \in [0,1]$

Result: Masked boolean value $\mathbf{r} \in [0,1]$

1 $\mathbf{n}_0 \leftarrow \text{NINOT}(\mathbf{b})$

2 $\mathbf{d}_0[0] \leftarrow \text{NIAND}(\mathbf{a}[0], \mathbf{n}_0[0])$

3 $\mathbf{d}_0[1] \leftarrow \text{SNIXOR}(\mathbf{a}[0], \mathbf{b}[0])$

4 $\mathbf{d}_1[1] \leftarrow \text{NINOT}(\mathbf{d}_0[0])$

5 $\mathbf{d}_2[1] \leftarrow \text{NIAND}(\mathbf{d}_1[1], \mathbf{a}[1])$

6 $\mathbf{d}_3[1] \leftarrow \text{NIAND}(\mathbf{d}_2[1], \mathbf{n}_0[0])$

7 $\mathbf{d}_0[2] \leftarrow \text{NIXOR}(\mathbf{a}[1], \mathbf{b}[1])$

8 $\mathbf{d}_1[2] \leftarrow \text{NINOT}(\mathbf{d}_0[2])$

9 $\mathbf{d}_2[2] \leftarrow \text{SNIAND}(\mathbf{d}_1[2], \mathbf{d}_1[1])$

10 $\mathbf{d}_3[2] \leftarrow \text{NIAND}(\mathbf{d}_2[2], \mathbf{c})$

11 $\mathbf{r}_0 \leftarrow \text{NIXOR}(\mathbf{d}_0[0], \mathbf{d}_3[1])$

12 $\mathbf{r}_1 \leftarrow \text{SNIXOR}(\mathbf{r}_0, \mathbf{d}_3[2])$

Output: $\mathbf{r}_1$

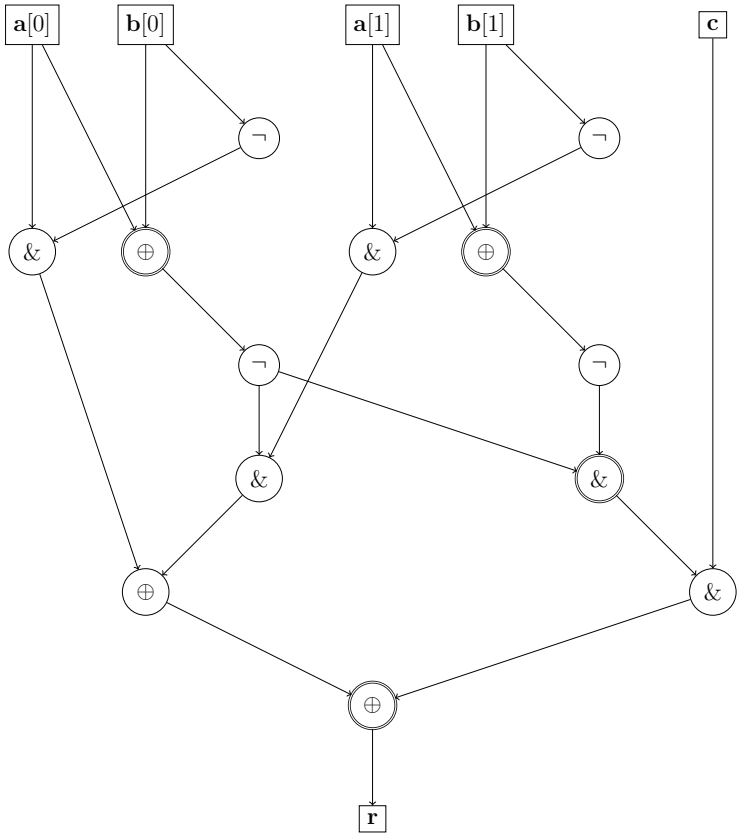

Figure 4.1: 2-bit magnitude comparator with cascading bit considers as a sequence of t-(S)NI gadgets.

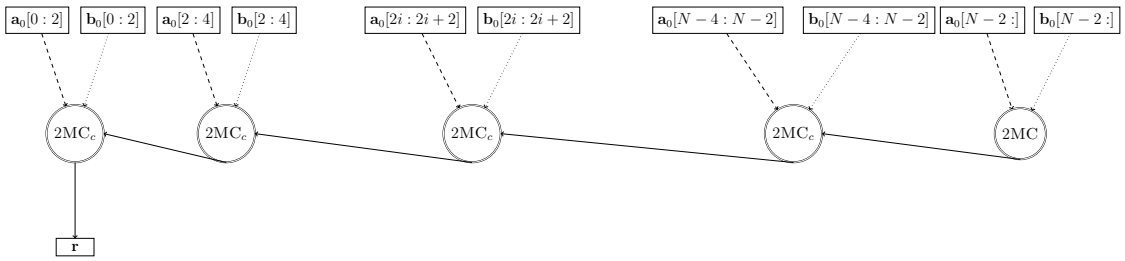

Figure 4.2: N-bit magnitude comparator - considered as a sequence of t-SNI cascading 2-bit magnitude comparators.

The larger N bit magnitude comparator is created by splitting the strings into blocks of two bits and using these as the inputs for the smaller 2 bit magnitude comparators. These smaller comparators are chained together with the output of the lowest order comparator cascading into the penultimate comparator. We consider this overall construction as a sequence of t-SNI gadgets in Fig. 4.2. As the magnitude comparator is not commutative, we use a dashed line to represent the first input (a) and a dotted line to represent the second input (b), where the magnitude comparator

is calculating the result of $a > b$.

Theorem 4.1.3 (t-SNI of SNINBitMagnitudeComparator (SNIMC))**.** *Let* $\mathbf{a}_0^{(.)}, \mathbf{b}_0^{(.)}$ *be the inputs and* $\mathbf{r}_1^{(.)}$ *be the output to* SNIMC *, where* $\boldsymbol{a}$ *and* $\boldsymbol{b}$ *are masked bitstrings and* $\boldsymbol{r}$ *is a single masked bit. For any set of* t' *intermediate variables and any subset* O *of output variables such that* $t' + |O| \leq t$*, there exists a subset* I *of input variables such that* $|I| \leq t'$ *and the* t' *intermediate variables and the output variables can be perfectly simulated using the* I *input variables.*

Proof. We consider SNIMC as a sequence of smaller gadgets in Fig. 4.2, where every gadget is t-SNI. As every gadget is t-SNI, then by Proposition 1 SNIMC is also t-SNI. □

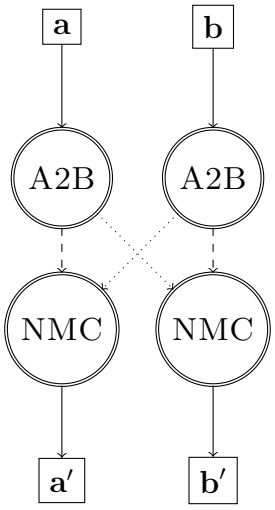

Figure 4.3: a gadget for converting non-binary coefficients into boolean values, given as a series of t-SNI gadgets.

In order to extend the the magnitude comparator to non-binary strings $\mathbf{a}, \mathbf{b}$, we have created a gadget that will take as input $\mathbf{a}[i], \mathbf{b}[i]$ and return two boolean values $\mathbf{a}'[i], \mathbf{b}'[i]$. The output will be such that all three inequalities $(<, =, >)$ are preserved, i.e. $\mathbf{a}[i] < \mathbf{b}[i] \iff \mathbf{a}'[i] < \mathbf{b}'[i]$. The gadget takes the inputs and considers each input itself as a binary string, in order to do this securely a gadget is used that converts between arithmetic and boolean masking. We then apply the standard magnitude comparator gadget to the two inputs, and the output will be $(\mathbf{1}, \mathbf{0}), (\mathbf{0}, \mathbf{0})$, or $(\mathbf{0}, \mathbf{1})$ as appropriate - where each value is also masked.

Theorem 4.1.4 (t-SNI of SNICoeff2Bin (SNIC2B))**.** *Let* $\mathbf{a}_0^{(.)}, \mathbf{b}_0^{(.)}$ *be the inputs and* $\mathbf{c}_1^{(.)}$,$\mathbf{d}_1^{(.)}$ *be the outputs to* SNIC2B*, where* $\boldsymbol{a}$ *and* $\boldsymbol{b}$ *are elements of* $GF(q^n)$ *for some prime* q *and are masked using arithmetic masking, and* $\boldsymbol{c}$ *and* $\boldsymbol{d}$ *are bitstrings masked using Boolean masking. For any set of* t' *intermediate variables and any subset* O *of output variables such that* $t' + |O| \leq t$*, there exists a subset* I *of input variables such that* $|I| \leq t'$ *and the* t' *intermediate variables and the output variables can be perfectly simulated using the* I *input variables.*

Proof. We consider SNIC2B as a sequence of smaller gadgets in Fig. 4.2, where every gadget is t-SNI. As every gadget is t-SNI, then by Proposition 1 SNIC2B is also t-SNI. □

4.1.2 LUP decomposition

LUP decomposition is a decomposition algorithm that converts a matrix into a decomposed matrix $(L-E)+U$, where $L-E$ is a lower triangular matrix without the diagonal and U is an upper triangular matrix such that LU equals some permutation of the rows of the input matrix. We give the algorithm for LUP decomposition in Algorithm 4.5, with a rough guide of the masked version of the algorithm in Algorithm 4.6. In order to argue about the security of our masked version of the LUP decomposition algorithm, we split the algorithm into smaller gadgets. We consider each nested for loop to be a separate gadget in order to discuss whether each iteration is independent or not and what changes therefore need to be made.

Algorithm 4.5: LUPDecomposition

Input: Matrix M

Result: Decomposed matrix M, permutation vector p

```
1
2  n ← Dim(M)
3  for i ← 0 to n do
4  |  p[i] ← i
5  end
6  p[n] ← 0
7  for i ← 1 to n do
8  |  maxA ← 0
9  |  imax ← i
10 |  for k ← i to n do
11 |  |  if M[k][i] ≥ maxA then
12 |  |  |  maxA ← M[k][i]
13 |  |  |  imax ← k
14 |  |  end
15 |  end
16 |  Swap(p[i], p[imax])
17 |  p[n] ← p[n] + 1
18 |  for k ← i to n do
19 |  |  Swap(M[i][k], M[imax][k])
20 |  end
21 |  for j ← i + 1 to n do
22 |  |  M[j][i] ← M[j][i]/M[i][i]
23 |  |
24 |  |  for k ← i + 1 to n do
25 |  |  |  M[j][k] ← M[j][k] −
   |  |  |    (M[j][i]M[i][k])
26 |  |  end
27 |  end
28 end
```

Output: M, p

Algorithm 4.6: SNILUPDecomp

Input: Matrix $\mathbf{M} \in GF(2^m)^{n \times n}$

Result: Decomposed matrix $\mathbf{M} \in GF(2^m)^{n \times n}$, permutation vector p

```
1  n ← Dim(M)
2  for i ← 0 to n do
3  |  p[i] ← i
4  end
5  p[n] ← 0
6  for i ← 1 to n do
7  |  maxA ← 0
8  |  imax ← i
9  |  for k ← i to n do
10 |  |  (maxA, imax) ←
   |  |    G8(M, maxA, imax)
11 |  |
12 |  |
13 |  end
14 |  NISwap(p[i], p[imax])
15 |  p[n] ← p[n] + 1
16 |  for k ← i to n do
17 |  |  NISwap(M[i][k], M[imax][k])
18 |  end
19 |  for j ← i + 1 to n do
20 |  |  a ← G3(M[i][i])
21 |  |  M[j][i] ←
   |  |    G1(M[j][i], M[i][i])
22 |  |  for k ← i + 1 to n do
23 |  |  |  M[j][k] ←
   |  |  |    G2(M[j][k], −G1(M[j][i], M[i][k]))
24 |  |  end
25 |  end
26 end
```

Output: $\mathbf{M}, p$

We now consider how to secure the inner loop of LUP Decomposition.

Algorithm 4.7: SNIGadget6

Input: $\mathbf{M}_0 \in GF(2^m)^{n \times n}$

Result: $\mathbf{M}_2 \in GF(2^m)^{n \times n}$

1 $\mathbf{a}_1 \leftarrow \text{SNIInv}(\mathbf{M}_0[i][i])$

2 $\mathbf{M}_1[j][i] \leftarrow \text{SNIMul}(\mathbf{M}_0[j][i], \mathbf{a}_1)$

3 **for** $k \leftarrow i+1$ **to** n **do**

4 $\quad \mathbf{M}_2[j][k] \leftarrow \text{SNIAdd}(\mathbf{M}_2[j][k], -\text{SNIMul}(\mathbf{M}_0[j][i], \mathbf{M}_0[i][k]))$

5 **end**

Output: $\mathbf{M}_2$

Theorem 4.1.5 (t-SNI of Gadget 6)**.** *Let $\mathbf{M}_0^{(.)}$ be the input and $\mathbf{M}_3^{(.)}$ be the output to Gadget 6, where $\boldsymbol{M} \in GF(2^m)^{n \times n}$ and is masked using Boolean masking. For any set of t' intermediate variables and any subset O of output variables such that $t' + |O| \leq t$, there exists a subset I of input variables such that $|I| \leq t'$ and the t' intermediate variables and the output variables can be perfectly simulated using the I input variables.*

Proof. All iterates i, j, k, n are public variables. As the iterations of the inner loop are all independent of each other, we consider them to be executed in parallel (lines 3-5 in Algorithm 4.7). We model Gadget 6 as a sequence of t-SNI secure gadgets, as depicted in Fig. 4.4. All input variables are immediately inputted into a t-SNI gadget. Whilst the intermediate variables corresponding to the output of G_1 is reused multiple times, each time it is used by a t-SNI gadget, and so by Proposition 1 the entire gadget is t-SNI. Moreover, due to the fact that every gadget call contains at least one of the inputs to G_6, we can state that every version of G_5 used must be t-SNI for G_6 to be t-SNI. □

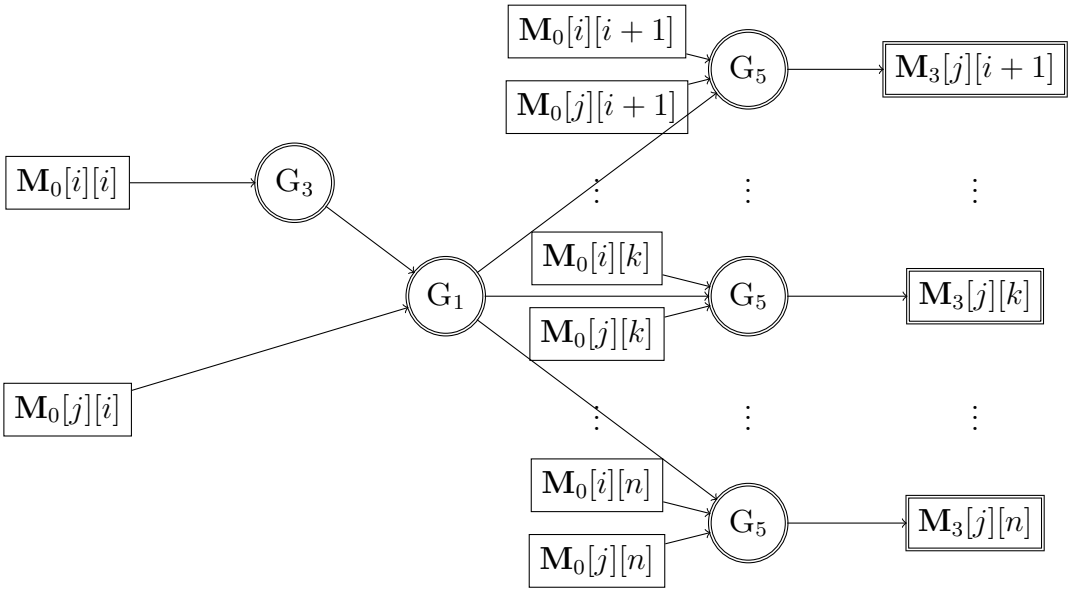

Figure 4.4: Gadget 6 - considered as a sequence of t-SNI gadgets.

The middle loop of LUP Decomposition, Gadget 7, only makes use of Gadget 6, which is t-SNI secure as shown above in Lemma 4.1.5.

Algorithm 4.8: GADGET7

Input: $\mathbf{M}_0$

1 **for** $j \leftarrow i+1$ **to** n **do**

2 $\quad \mathbf{M}_3[j][i+1,\ldots,n] \leftarrow$
$\quad \mathrm{SNIGADGET6}(\mathbf{M}_0[j][i], \mathbf{M}_0[i][i], \mathbf{M}_0[j][i+1,\ldots,n], \mathbf{M}_0[i][i+1,\ldots,n])$

3 **end**

Output: $\mathbf{M}_3$

Algorithm 4.9: GADGET7P (n)

Input: $\mathbf{M}_0 \in GF(2^m)^{n\times n}$

1 $\mathbf{M}_2 \leftarrow \mathrm{SNIGADGET6P}(n, \mathbf{M}_1)$

Output: $\mathbf{M}_3$

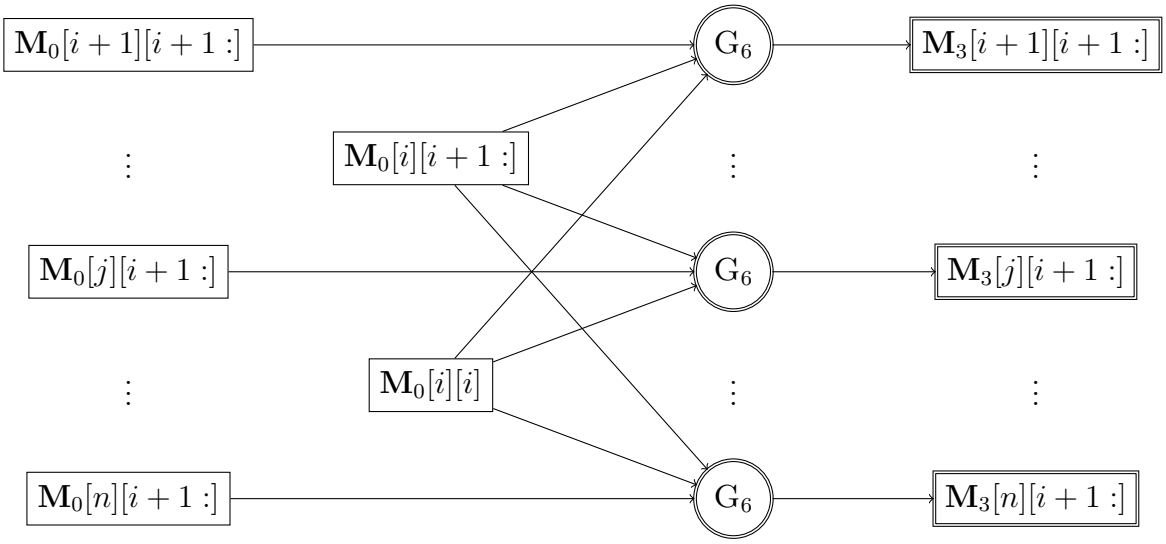

Figure 4.5: Gadget 7 - considered as a sequence of t-SNI gadgets.

Theorem 4.1.6 (t-SNI of SNIGadget7)**.** *Let $\mathbf{M}_0^{(.)}$ be the input and $\mathbf{M}_3^{(.)}$ be the output to Gadget 7, where $\boldsymbol{m} \in GF(2^m)^{n\times n}$. For any set of t' intermediate variables and any subset O of output variables such that $t' + |O| \leq t$, there exists a subset I of input variables such that $|I| \leq t'$ and the t' intermediate variables and the output variables can be perfectly simulated using the I input variables.*

Proof. All iterates i, j, n are public variables. As the iterations of the inner loop are all independent of each other, we consider them to be executed in parallel and summarize them (lines 1-3 in Algorithm 4.8) into one gadget (line 1 in Algorithm 4.9). We model Gadget 7 as a sequence of t-SNI secure gadgets, as depicted in Fig. 4.5. As the only gadgets used as part of this construction are t-SNI, then by Proposition 1 the entire gadget is t-SNI. Moreover, due to the fact that every gadget call shares an input to G_7, we can state that every version of G_6 used must be t-SNI for G_7 to be t-NI or t-SNI. □

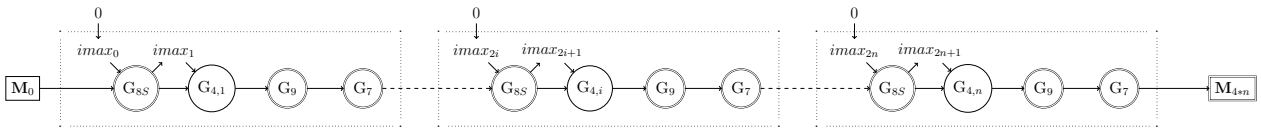

Figure 4.6: MaskedLUPDecomposition - considered as a sequence of t-(S)NI gadgets.

Gadget10 is full SNILUPDecomposition. We build Gadget10 from a

number of gadgets - NISwap, denoted G_{4-}, is a t-NI gadget, and can be made t-SNI by applying a t-SNI mask refresh, G_9, after its usage. The permutation vector p is public and unmasked, and so we leave out of the depiction in Fig. 4.6. As Fig. 4.6 contains every aspect of SNILUPDecomposition that uses secret values as part of the computation, it suffices to show that Fig. 4.6 is t-SNI.

Theorem 4.1.1 (t-SNI of SNILUPDecomposition)**.** *Let* $\mathbf{M}_0^{(.)}$ *be the input and* $\mathbf{M}_{4n}^{(.)}$ *be the output to* Gadget10 *, where* $\boldsymbol{M} \in GF(2^m)^{(n\times n)}$*. For any set of* t' *intermediate variables and any subset* O *of output variables such that* $t' + |O| \leq t$*, there exists a subset* I *of input variables such that* $|I| \leq t'$ *and the* t' *intermediate variables and the output variables can be perfectly simulated using the* I *input variables.*

Proof. As the iterations of the for loop are not independent of each other, we consider them to be executed in series. There are then modelled as a sequence of t-(S)NI gadgets, as depicted in Fig. 4.6. As every gadget used in the construction is t-NI or t-SNI, and the final gadget call is t-SNI, then by Proposition 1 SNILUPDecomposition is also t-SNI. We also note that whilst for SNILUPDecomposition to be t-SNI, we only require the final call to G_7 to be t-SNI, due to the structure of G_7 it will always be t-SNI.

□

4.1.3 Determinant

To calculate the determinant of a matrix we use the LUP decomposition algorithm to decompose the matrix, and the determinant is the product of the leading diagonal of the decomposed matrix. This algorithm can be found in Algorithm 4.10 and the masked version can be found in Algorithm 4.11.

Algorithm 4.10: LUPDeterminant

Input: Matrix M

Result: Determinant det

1 $n \leftarrow \text{Dim}(M)$

2 $M, p \leftarrow$ LUPDecomposition(M)

3 $det \leftarrow M[1][1]$

4 **for** $i \leftarrow 2$ **to** n **do**

5 $det \leftarrow M[i][i]det$

6

7 **end**

8 $det \leftarrow (-1)^{p[n]}det$

Output: det

Algorithm 4.11: SNILUPDeterminant

Input: Matrix $\mathbf{M} \in GF(2^m)^{n \times n}$

Result: Determinant $\mathbf{det} \in GF(2^m)$

1 $n \leftarrow \text{Dim}(\mathbf{M})$

2 $\mathbf{M}_1, p \leftarrow \text{SNILUPDecomp}(\mathbf{M}_0)$

3 $\mathbf{det}_0 \leftarrow \mathbf{M}_1[1][1]$

4 **for** $i \leftarrow 2$ **to** n **do**

5 $\mathbf{det}_{i-1} \leftarrow$ NIMul$(\mathbf{M}_1[i][i], \mathbf{det}_{i-2})$

6 **end**

7 $\mathbf{det}_n \leftarrow \text{NIMul}((-1)^{p[n]}, \mathbf{det}_{n-1})$

Output: $\mathbf{det}_n$

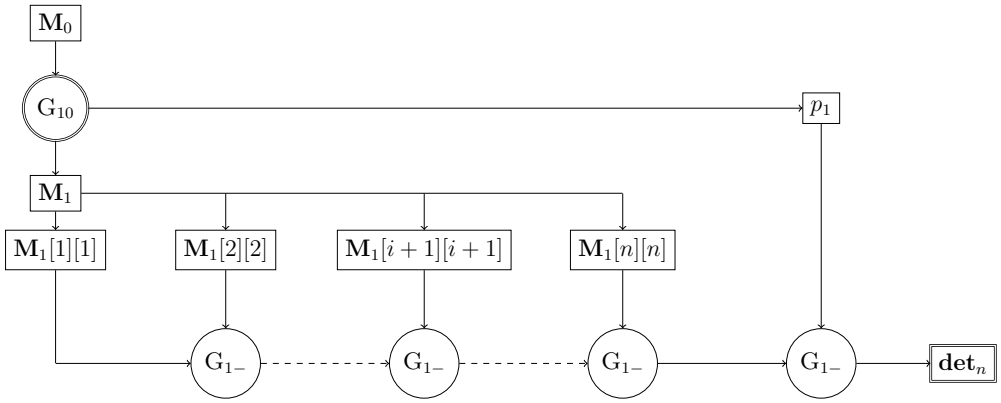

Figure 4.7: Matrix Determinant - considered as a sequence of t-SNI gadgets.

Theorem 4.1.2 (t-SNI of SNILUPDeterminant)**.** *Let* $\mathbf{M}_0$ *be the input and* $\mathbf{det}_n$ *be the output of* SNILUPDeterminant *where* $\boldsymbol{M} \in GF(2^m)^{n \times n}$ *and* $\boldsymbol{det} \in GF(2^m)$*. For any set of* t' *intermediate variables and any subset* O *of output variables such that* $t' + |O| \leq t$*, there exists a subset* I *of input variables such that* $|I| \leq t'$ *and the* t' *intermediate variables and the output variables can be perfectly simulated using the* I *input variables.*

Proof. As the iterations of the for loop are not independent of each other, we consider them to be executed in series. These are then modelled as a sequence of t-(S)NI

gadgets, as depicted in Fig. 4.7. As no variable is used more than once, and the input is only used by a t-SNI gadget, then by Proposition 1 SNILUPDeterminant is also t-SNI. Moreover we note that a t-NI version of this gadget can be made by only using a t-NI version of G_{10}.

□

4.1.4 Solving linear equations

In order to solve linear equations $Ax = b$, we first decompose the matrix A using LUP decomposition, giving $LUx = Pb$. We then solve $Ly = Pb$ to get y and finally $Ux = y$ to get x. We give this algorithm in Algorithm 4.12 and the masked counterpart in Algorithm 4.13. In order to argue about the security of the masked algorithm, we break each nested for loop into a separate gadget to best assess their independence.

Algorithm 4.12: LUPSolve

Input: Matrix M, vector b

Result: Vector x, s.t. $Mx = b$

```
1  n ← Dim(M)
2  (M, p) ←
     LUPDecomposition(M)
3  for i ← 0 to n do
4  |   x[i] ← b[p[i]]
5  |   for k ← 0 to i do
6  |   |   x[i] ← x[i] − M[i][k]x[k]
7  |   |
8  |   end
9  end
10 for i ← n − 1 to 0 do
11 |   x[i] ← b[p[i]]
12 |   for k ← i + 1 to n do
13 |   |   x[i] ← x[i] − M[i][k]x[k]
14 |   |
15 |   end
16 |   x[i] ← x[i]/M[i][i]
17 |
18 end
```

Output: x

Algorithm 4.13: SNILUPSolve

Input: Matrix $\mathbf{M} \in GF(2^m)^{n \times n}$, vector $\mathbf{b} \in GF(2^m)^n$

Result: Vector $\mathbf{x}$, s.t. $\mathbf{Mx} = \mathbf{b}$

```
1  n ← Dim(M)
2  (M, p) ← SNILUPDecomp(M)
3  for i ← 0 to n do
4  |   x[i] ← b[p[i]]
5  |   for k ← 0 to i do
6  |   |   x[i] ←
   |   |     SNIAdd(x[i], −SNIMul(M[i][k], x[k]))
7  |   end
8  end
9  for i ← n − 1 to 0 do
10 |   x[i] ← b[p[i]]
11 |   for k ← i + 1 to n do
12 |   |   x[i] ←
   |   |     SNIAdd(x[i], −SNIMul(M[i][k], x[k]))
13 |   end
14 |   a = SNIInv(M[i][i])
15 |   x[i] ← SNIMul(x[i], a)
16 end
```

Output: $\mathbf{x}$

Algorithm 4.14: GADGET12

Input: $\mathbf{M}_0 \in GF(2^m)^{(n\times n)}$, $\mathbf{x}_0 \in GF(2^m)^n$

1 $\mathbf{x}_1[i] \leftarrow \mathbf{b}[p[i]]$

2 **for** $k \leftarrow u$ **to** $v-1$ **do**

3 $\quad \mathbf{x}_{u-k+1}[i] \leftarrow \text{NIADD}(\mathbf{x}_{u-k}[i], -\text{NIMUL}(\mathbf{M}_0[i][k], \mathbf{x}_1[k]))$

4 **end**

5 $\mathbf{x}_{u-v}[i] \leftarrow \text{SNIADD}(\mathbf{x}_{u-v-1}[i], -\text{NIMUL}(\mathbf{M}_0[i][v], \mathbf{x}_1[v]))$

Output: $\mathbf{x}_{u-v}$

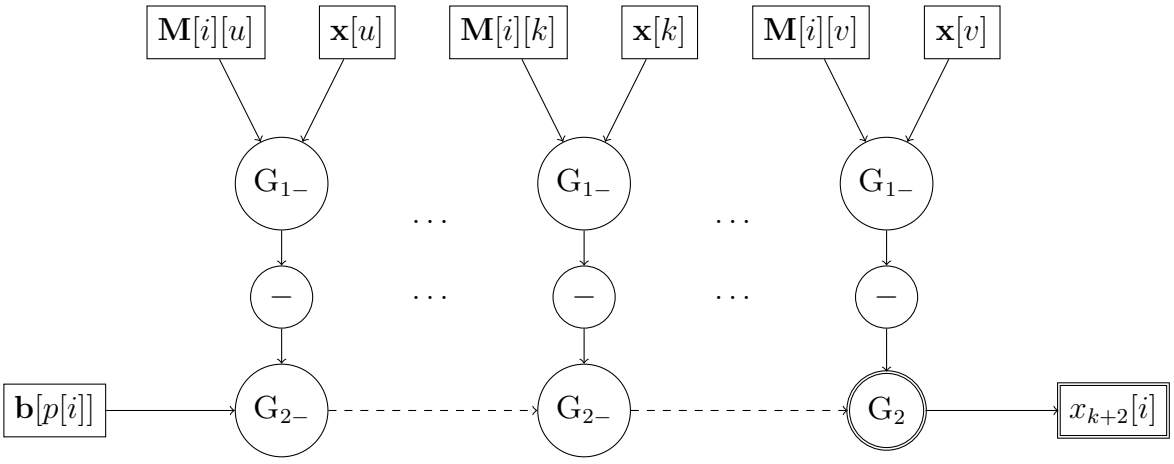

Figure 4.8: Gadget 12 - considered as a sequence of t-SNI gadgets.

Theorem 4.1.7 (t-SNI of Gadget 12)**.** *Let $\boldsymbol{M}_0^{(\cdot)}[i][u,\ldots,v]$, $\boldsymbol{b}_0^{(\cdot)}[p[i]]$ and $\boldsymbol{x}_0^{(\cdot)}[u,\ldots,v]$ be the input, and $\boldsymbol{x}_{k+1}^{(\cdot)}[i]$ be the output to Gadget 12, where $\boldsymbol{M} \in GF(2^m)^{(n\times n)}$, $\boldsymbol{b}, \boldsymbol{x} \in GF(2^m)^n$ are masked using Boolean masking and $p \in GF(2^m)^n$ is public. For any set of t' intermediate variables and any subset O of output variables such that $t'+|O| \leq t$, there exists a subset I of input variables such that $I \leq t'$ and the t' intermediate variables and the O output variables can be perfectly simulated using the I input variables.*

Proof. All iterates i, k, u, v, and the vector p are public variables. Since $k \in [u,\ldots,v]$, no assignment of a variable is used more than once. Since the only gadgets used are t-SNI or t-NI, and the last gadget is t-SNI, then by Proposition 1 the entire gadget is t-SNI. □

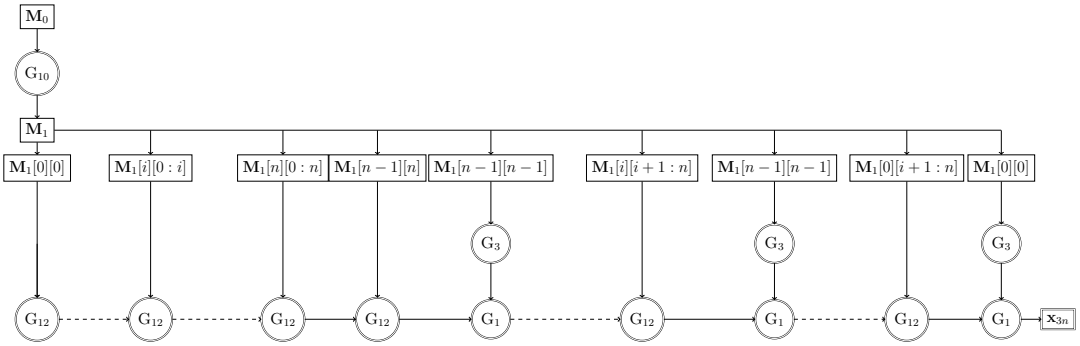

Figure 4.9: SNILUPSolve - considered as a sequence of t-(S)NI gadgets.

Theorem 4.1.3 (t-SNI of SNILUPSolve)**.** *Let* $\hat{\mathbf{M}}_0^{(.)}$ *be the input and* $\mathbf{x}_{3n}^{(.)}$ *be the output to* G_{12}*, where* $\boldsymbol{M} \in GF(2^m)^{(n\times n)}, \boldsymbol{x} \in GF(2^m)^n$*. For any set of* t' *intermediate variables and any subset* O *of output variables such that* $t' + |O| \leq t$*, there exists a subset* I *of input variables such that* $|I| \leq t'$ *and the* t' *intermediate variables and the output variables can be perfectly simulated using the* I *input variables.*

Proof. The only gadgets that make up SNILUPSolve are SNIMul, SNIInv, SNILUPDecomp and G_{12}. As all of these gadgets are t-SNI and each intermediate variable is only an input to one gadget, then by Proposition 1 SNILUPSolve is t-NI. Moreover because the first gadget the input passes through is t-SNI, then SNILUPSolve is also t-SNI. □

4.2 Results

We give a brief overview of the time and randomness complexity of the smaller gadgets in Table 4.1. The complexity of G_8 depends on the bit length of the values being compared, we focus on the case where all values are 8 bits. Inverse's complexity depends on the size of the finite field used, for this example we use $GF(2^8)$.

Gadget	$\mathcal{T}$	$\mathcal{R}$
SNIMul/SNIAnd	$3t^2$	$\frac{t^2}{2}$
NIMul/NIAnd	$\frac{7t^2}{4}$	$\frac{t^2}{4}+t$
SNIRef	t^2	$\frac{t^2}{2}$
NIRef	$2t$	t
SNIAdd/SNIXOR	t^2+t	$\frac{t^2}{2}$
NIAdd/NIXOR	t	–
NINot	1	–
SNIInv	$12t^2+3t$	$3t^2+2t$
NISwap	t	–
G_8	$44t^2+6t$	$16t^2+15t$

Table 4.1: The time and randomness complexity of small gadgets.

We now discuss the time complexity and the randomness complexity of the main algorithms.

LUPDecomposition. We estimate the time complexity of LUPDecomposition, $\mathcal{T}_{\text{LUP}}$, as

$$\begin{aligned}\mathcal{T}_{\text{LUP}} &= n(n\mathcal{T}_{G_8} + n\mathcal{T}_{\text{NISwap}} + n\mathcal{T}_{\text{SNIInv}} + n^2(\mathcal{T}_{\text{NIAdd}} + \mathcal{T}_{\text{SNIMul}})) \\ &= n^2\mathcal{T}_{G_8} + n^2\mathcal{T}_{\text{NISwap}} + n^2\mathcal{T}_{\text{SNIInv}} + n^3\mathcal{T}_{\text{NIAdd}} + n^3\mathcal{T}_{\text{SNIMul}} \\ &= n^2(44t^2+6t) + n^2(t) + n^2(12t^2+3t) + n^3(3t) + n^3(3t^2) \\ &= O(n^3t^2)\end{aligned}$$

We estimate the randomness complexity of LUPDecomposition, $\mathcal{R}_{\text{LUP}}$, as

$$\begin{aligned}\mathcal{R}_{\text{LUP}} &= n(n\mathcal{R}_{\text{G}_8} + n\mathcal{R}_{\text{NISwap}} + n\mathcal{R}_{\text{SNIInv}} + n^2(\mathcal{R}_{\text{NIAdd}} + \mathcal{R}_{\text{SNIMul}})) \\ &= n^2\mathcal{R}_{\text{G}_8} + n^2\mathcal{R}_{\text{NISwap}} + n^2\mathcal{R}_{\text{SNIInv}} + n^3\mathcal{R}_{\text{NIAdd}} + n^3\mathcal{R}_{\text{SNIMul}} \\ &= n^2(16t^2 + 15t) + n^2(3t^2 + 2t) + n^3(t^2) \\ &= O(n^3t^2)\end{aligned}$$

Determinant calculation. We now estimate the time and randomness complexity of LUPDeterminant, $\mathcal{T}_{\text{LUPDet}}$, assuming that LUPDecomposition has already been performed.

$$\begin{aligned}\mathcal{T}_{\text{LUPDet}} &= n\mathcal{T}_{\text{NIMul}} \\ &= n(\frac{7t^2}{4}) \\ &= O(nt^2)\end{aligned}$$

$$\begin{aligned}\mathcal{R}_{\text{LUPDet}} &= n\mathcal{R}_{\text{NIMul}} \\ &= n(\frac{t^2}{4}) \\ &= O(nt^2)\end{aligned}$$

Solving linear equations. We now estimate the time and randomness complexity of LUPSolve, $\mathcal{T}_{\text{LUPSolve}}$, assuming that LUPDecomposition has already been performed.

$$\begin{aligned}\mathcal{T}_{\text{LUPSolve}} &= 2n(n\mathcal{T}_{\text{NIMul}} + (n-1)\mathcal{T}_{\text{NIAdd}} + \mathcal{T}_{\text{SNIAdd}}) + n\mathcal{T}_{\text{SNIInv}} + n\mathcal{T}_{\text{SNIMul}} \\ &= 2n^2\mathcal{T}_{\text{NIMul}} + n(n-1)\mathcal{T}_{\text{NIAdd}} + n\mathcal{T}_{\text{SNIAdd}} + n\mathcal{T}_{\text{SNIInv}} + n\mathcal{T}_{\text{SNIMul}} \\ &= 2n^2(\frac{7t^2}{4}) + n(n-1)t + n(t^2 + t) + n(12t^2 + 3t) + n(\frac{7t^2}{4}) \\ &= O(n^2t^2)\end{aligned}$$

$$\begin{aligned}
\mathcal{R}_{\text{LUPSolve}} &= 2n(n\mathcal{R}_{\text{NIMul}} + (n-1)\mathcal{R}_{\text{NIAdd}} + \mathcal{R}_{\text{SNIAdd}}) + n\mathcal{R}_{\text{SNIInv}} + n\mathcal{R}_{\text{SNIMul}} \\
&= 2n^2\mathcal{R}_{\text{NIMul}} + n(n-1)\mathcal{R}_{\text{NIAdd}} + n\mathcal{R}_{\text{SNIAdd}} + n\mathcal{R}_{\text{SNIInv}} + n\mathcal{R}_{\text{SNIMul}} \\
&= 2n^2(\frac{t^2}{2}) + n(\frac{t}{2}) + n(3t^2 + 2t) + n(\frac{t^2}{2}) \\
&= O(n^2t^2)
\end{aligned}$$

From this we can see that our masked implementation only has an overhead of order t^2 over the standard implementation, this is inline with the most efficient masked multiplication gadgets. Using this we can now look to creating masked implementations of the BCH code decoding algorithm, which relies heavily on these linear algebra algorithms.

CHAPTER 5

Securing BCH codes

Most LWE-based cryptosystems have the possibility of a decryption failure, where the ciphertext is decrypted incorrectly. These decryption failures are caused by the noise introduced during key generation and encryption. Knowing the exact decryption failure rate is important [DVV19], as it has been shown that it is possible to boost the probability [DRV20] and to utilise failures as part of attacks [DGJ+19].

A number of round 2 schemes introduced error correction to reduce the decryption failure rate [Ham19, LLJ+19, GZB+19], making the scheme more secure to decryption failure attacks and also to allow for the use of smaller moduli, thus reducing the size of ciphertexts. There has also been further work specifically looking at the effectiveness of using a number of different error correcting codes to improve R-LWE schemes [WL21, FPS19], which also generalises to other LWE-based schemes.

As well as being mathematically secure, it is important for the implementations of the schemes to be secure, including against side-channel attacks. Several of the KEMs that utilise error correcting codes have been broken by performing side-channel attacks on the decoding of the error correcting code, with various attacks exploiting information gained from timing [DTVV19], power or electromagnetic (EM) [RRCB20]. There also exists successful side-channel attacks against some

of the code based cryptosystems, e.g. HQC [SHR+22].

Related work. One of the main method of preventing power analysis attacks is to use masking [CJRR99], and there is a large body of work applying this to secure a number of schemes, e.g. Kyber [BGR+21], Saber [BDK+20]. Another method used to secure against side-channel attacks is blinding [Koc96], where the input is changed to appear random to make the leakage unpredictable, this is used very often in protecting RSA against timing attacks. A constant time implementation of BCH code decoding was introduced in [WR19] that is secure against timing attacks. However we are not aware of any defence against power analysis or EM analysis attacks for error correcting codes.

Contributions. To the best of our knowledge, there has been no analysis on how to mask any error correcting code. In this chapter we present the first analysis to realise a complete masked error correcting code - specifically binary BCH codes, as part of this we also present masked versions of LUP decomposition and an algorithm for finding the determinant of a matrix. We present techniques for constructing both first- and higher-order masking schemes with formal proofs in the probing model. Rather than focusing on a specific parameter set, our masking techniques are extended for the general BCH code, although with a focus on how they would be used in conjunction with a LWE-based KEM - specifically ensuring that they would be compatible with the masked versions of Kyber and Saber.

In order to mask the BCH code decoding algorithm we break it down into the following smaller components.

- **Syndrome calculation.** The BCH code decoding algorithm requires the syndromes to be calculated from the message. We give an algorithm to calculate the syndrome that is compatible with a masked message and is masked in such a way that not only are the message and syndrome values hidden, but also whether the syndrome is the all zero vector.
- **Adapted Peterson Algorithm.** Whilst a constant time version of the Berlekamp–Massey decoding algorithm does exist [WR19], it has other limita-

tions in terms of masking. We instead adapt the Peterson algorithm to make it constant time and mask it to hide both the locations of the errors corrected and the number of errors being corrected.

- **Chien Search.** Finally we mask the Chien search algorithm. In order to do this in such a way that the number of errors is hidden, as well as the locations of the errors, we ensure that we always have the maximum (or maximum - 1) number of errors.

Several of the attacks against LWE based schemes that exploit side-channel information from the error correcting codes, only utilised the knowledge that an error had been corrected, rather than needing to know any of the values. The main challenge with masking error correcting codes that are used in LWE based schemes has been around ensuring that no information relating to the number of errors being corrected is leaked. In order to show that we manage this, we rely mostly on experimental proofs, whilst also using proofs in the probing model to show that the values are hidden. We present an implementation of the first-order masking of specific BCH codes with parameters chosen to be compatible with LWE-based KEMs. This is used to conduct experimental verification of our first-order implementation by using ELMO [MOW17] to emulate the power consumption of the implementation and assessing the leakage using the Test Vector Leakage Assessment (TVLA) [GGJR+11]. Our masked decoding algorithm is 4.90x slower than the constant time implementation presented in [WR19]. The overheads for the, now optimised, side-channel resistant implementations of Saber and Kyber are 2.5x [BDK+20], and 3.5x [BGR+21] respectively. At the moment neither of these schemes use error correction, however we argue in Chapter 3 that they would both benefit from utilising it.

We make use of the following gadgets:

G_1	SNIMul	a t-SNI([BBD+16]) secure algorithm for multiplication.
G_{1-}	NIMul	a t-NI([BBP+16]) secure algorithm for multiplication.
$\hat{G_1}$	SNIAnd	a t-SNI([BBD+16]) secure algorithm for logical AND.
$\hat{G_{1-}}$	NIAnd	a t-NI([BBP+16]) secure algorithm for logical AND.
G_2	SNIAdd	a t-SNI(Lemma 4.1.1) secure algorithm for addition.
G_{2-}	NIAdd	a t-NI secure algorithm for addition.
$\hat{G_2}$	SNIXOR	a t-SNI(Lemma 4.1.1) secure algorithm for logical XOR.
$\hat{G_{2-}}$	NIXOR	a t-NI secure algorithm for logical XOR.
G_3	SNIInv	a t-SNI([BBD+16]) secure algorithm for inverting an element of a field.
G_{4-}	NISwap	a t-NI secure algorithm for swapping the values of two variables.
G_8	SNIGadget8	a t-SNI(Lemma 4.1.3) secure algorithm for returning the shared result of $a \leq b$.
G_9	SNIRef	a t-SNI([BBD+16]) secure algorithm for mask refreshing.
G_{10}	SNILUPDecomp	a t-SNI(Theorem 4.1.1) secure algorithm that returns the LUP Decomposition of a matrix.
G_{11}	SNILUPDeterminant	a t-SNI(Theorem 4.1.2) secure algorithm that returns the determinant of a matrix.
G_{13}	SNILUPSolve	a t-SNI(Theorem 4.1.3) secure algorithm that solves a series of linear equations.
G_{14}	SNISyndromeCalculation	a t-SNI(Lemma 5.2.1) secure algorithm that returns the Syndrome of the received message.
G_{15}	SNIAdaptedPeterson	a t-SNI(Lemma 5.2.3) secure algorithm that returns the error location polynomial.
G_{16}	SNIChienSearch	a t-SNI(Lemma 5.2.4) secure algorithm that returns the locations of the errors.
G_{17}	SNIBCHDecoding	a t-SNI(Theorem 5.2.1) secure algorithm that decodes and corrects the input message.
G_{18-}	NIPetersonMatrix	a t-NI secure algorithm that creates the Peterson matrix.

5.1 Method

In order to protect against attacks, we first consider the weaknesses of the BCH decoding algorithm. Three main issues mean that BCH decoding is susceptible to timing and power analysis attacks. The first issue is that it is possible to pick specific codewords that maximise the statistical differences in power usage, regardless of how many errors there are, and that the attacker can pick any two messages to maximise this. The second issue is the number of errors present, especially for timing leakage, as most aspects of the algorithms have a run time that is variable in the number of errors. The leaks from errors are exacerbated even further when one of the messages has no errors, as this leads to all the syndromes being 0. The final issue is that even the location of the errors causes leakage, which is most noticeable when in one message all the errors are grouped at the start and in the other they are all grouped at the end. Since in both the CPA and the CCA setting the attacker can choose the plaintext, we do not need to consider masking the BCH encoding algorithm.

In order to combat these issues we first blind the input by generating a random, valid, codeword, which we then xor with the received message, making it impossible for an attacker to pick specific codewords that maximise the efficacy of the attack. This has no impact on the value of the syndrome, which is only influenced by the number and location of errors. However it does impact the values of shares of the syndrome, in the case where no masking is employed then the all 0 codeword maps to every share being 0, however all other valid codewords map to shares which are not all 0. Secondly to reduce the impact of the syndromes being 0, we mask each syndrome using arithmetic masking and adapt the algorithms to work on masked variables (see Section 5.2.1).

Finally, to reduce issues about the number of errors present, we keep adding errors until there are a total of e or $e-1$ errors in the message (see Section 5.2.2). We use the Peterson algorithm instead of the Berlekamp-Massey algorithm, as our initial tests suggest that it is easier to distinguish between the error locations when using the Berlekamp-Massey algorithm than when using the Peterson algorithm. An overview of our masked algorithm can be found in Algorithm 6.3, this can be easily compared with the standard BCH code decoding algorithm using the Peterson

Algorithm 5.1: BCHDecoding

Input: Received vector $r(x)$
Result: Codeword $r(x)$

$S \leftarrow \text{Syndromes}(r(x))$
$M \leftarrow \text{GeneratePetersonMatrix}(S)$
$det \leftarrow 0$
while $det == 0$ **do**
  $det \leftarrow \text{LUPDeterminant}(M)$
  if $det == 0$ **then**
    $M \leftarrow M[0:-2][0:-2]$
  end
end
$\Lambda(x) \leftarrow \text{LUPSolve}(M)$

$\mathbf{r}(x) \leftarrow \text{ChienSearch}(\Lambda(x), \mathbf{r}(x))$
Output: $\mathbf{r}(x)$

Algorithm 5.2: SNIBCHDecoding

Input: Received vector $\mathbf{r}_0(x)$
Result: Codeword $\mathbf{r}(x)$

$\mathbf{c}'(x) \leftarrow \text{RandomCodeword}$
$\mathbf{r}_1(x) \leftarrow \mathbf{r}_0(x) \oplus \mathbf{c}'(x)$
$\mathbf{S} \leftarrow \text{SNISyndromes}(\mathbf{r}_1(x))$
$\mathbf{M} \leftarrow \text{NIPetersonMatrix}(\mathbf{S})$

$\mathbf{M} \leftarrow \text{SNIAdaptedPeterson}(\mathbf{M})$

$\mathbf{\Lambda}(x) \leftarrow \text{SNILUPSolve}(\mathbf{M})$
$\mathbf{r}_2(x) \leftarrow \text{SNIChienSearch}(\mathbf{\Lambda}(x), \mathbf{r}_1(x))$
$\mathbf{r}_3(x) \leftarrow \mathbf{r}_2(x) \oplus \mathbf{c}'(x)$
Output: $\mathbf{r}3(x)$

algorithm in Algorithm 5.1.

Before using LUP-decomposition as our algorithm for checking if a matrix is singular, we considered a number of other algorithms. The first algorithm we considered was one for calculating $\det(A+B)$ without calculating $A+B$ [Mar90]. Whilst this would be very easy to mask and would have a very low randomness complexity, unfortunately it is a very inefficient algorithm. Another algorithm considered was the Bareiss algorithm [Bar68], which calculates the leading principal minors $[M]_{1,1}$ to $[M]_{n,n}$ in $O(N^3)$, however if any of the leading principal minors are 0, then this algorithm will set all subsequent leading principal minors to 0. This would lead to an incorrect determinant being calculated, or by permuting rows - in which case we are no longer calculating the principal minors of the original matrix, and so any savings made are lost.

Our masked BCH code decoding algorithm works for all binary BCH codes, however our experiments are specifically for the BCH(255,128,8) code.

5.2 Masking BCH code decoding

In this section we present the masked versions of the main components of the BCH code decoding algorithm.

5.2.1 Syndrome calculation

When there are no errors, each value in the syndrome is 0, which has been shown to be exploitable in Ravi et al [RRCB20]. To avoid each value being 0, we split each $S[i]$ into shares. We adapt the algorithm in the following way. The syndrome $\mathbf{S}$ is a $2e-$tuple, where each $\mathbf{S}[i]$ is calculated as $\mathbf{S}[i] = \sum_{k=0}^{n-1} e_k(\alpha^i)^k$, where only $\mathbf{e}$ and $\mathbf{S}$ are secret variables. This can be considered as splitting the received vector into shares and calculating the syndromes on each share of the codeword individually. The masked syndrome calculation algorithm makes use of three smaller gadgets: the t-SNI gadget SNIMul, or G_1, the t-NI gadget NIMul, or G_{1-}, and the t-NI gadget NIAdd, or G_{2-}. Since each $\mathbf{S}[i]$ can be calculated independently, we consider the algorithm to calculate some $\mathbf{S}[i]$ rather than calculating $\mathbf{S}$. This is modelled as a sequence of t-(S)NI gadgets in Fig. 5.1.

Theorem 5.2.1 (t-SNI of SNISyndromeCalculation)**.** *Let $\mathbf{e}_0$ be the input and $\mathbf{S}[i]_n$ be the output of Syndrome calculation, where $\boldsymbol{e}$ is a bitstring, masked using Boolean masking, and each $\boldsymbol{S}[i] \in GF2^m$. For any set of t' intermediate variables and any subset O of output variables such that $t' + |O| \leq t$, there exists a subset I of input variables such that $|I| \leq t'$ and the t' intermediate variables and the output variables can be perfectly simulated using the I input variables.*

Proof. These are then modelled as a sequence of t-(S)NI gadgets, as depicted in Fig. 5.1. As all the constituent gadgets are t-NI or t-SNI and no variable is used more than once, then by Proposition 1 the entire gadget is t-NI. As the final gadget used it t-SNI then by Proposition 1 the algorithm is also t-SNI, moreover only the final call to G_2 needs to be t-SNI for the entire gadget to be t-SNI.

□

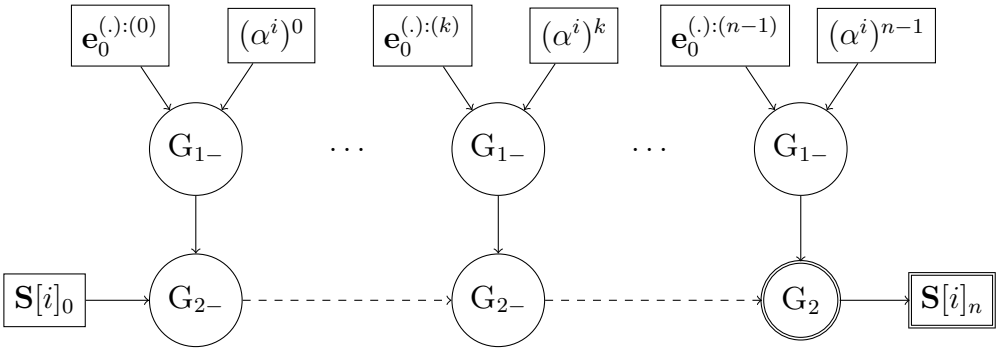

Figure 5.1: SNISyndromeCalculation - considered as a sequence of t-(S)NI gadgets.

5.2.2 Adapted Peterson algorithm

We take the Peterson algorithm and adapt it into our Adapted Peterson algorithm, the pseudocode for which can be found in Algorithm 5.3. The main change that we make is that instead of reducing the size of the matrix until it is non-singular and then stopping, we keep reducing the size of the matrix until it is empty and if at any point the reduced matrix is singular then we add errors. This is then run a further $e - 2$ times. The first part of this ensures that the adapted algorithm is constant time, and the second part of this ensures that after the algorithm has finished that we have added errors until there are either e or $e - 1$ errors overall.

For each iteration the algorithm calculates the syndromes using SNISyndrome-Calculation and uses the SNILUPDeterminant to calculate the determinants, and we then check if each determinant is 0 by using G_8 with the public input of 0 for the second input. We then generate 2 errors and if the determinant is 0 we add them to the message, we store all randomly generated errors to ensure that we never add an error to the same location twice. As there is a chance that the error the algorithm adds corrects an error that we've added to the codeword, we repeat this process $e - 2$ times, this guarantees we always have e or $e - 1$ errors. We then use SNILUPSolve to calculate the error location polynomial. We use

NIPetersonMatrix (G_{18-}) to generate the shared Peterson matrix $\mathbf{A}$ as follows:

$$\mathbf{A}^{(i)} = \begin{pmatrix} 0 & 0 & 0 & \dots & 0 & 0 \\ \mathbf{S}^{(i)}[2] & \mathbf{S}^{(i)}[1] & 0 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ \mathbf{S}^{(i)}[2e-2] & \mathbf{S}^{(i)}[2e-3] & \mathbf{S}^{(i)}[2e-4] & \dots & \mathbf{S}^{(i)}[e] & \mathbf{S}^{(i)}[e-1] \end{pmatrix}$$

however for $i = 1, \forall j \in [1, t+1]$ we set $A^{(1)}[j][2j-1] = 1$.

Algorithm 5.3: SNIAdaptedPeterson

Input: Received message $r(x)$

Result: Error location polynomial $\mathbf{\Lambda}(x)$, received message with e errors $r(x)$

```
for i ← 1 to e − 2 do
    S ← SNISyndromeCalculation(r(x))
    M ← NIPetersonMatrix(S)
    for j ← 1 to e − 1 do
        det ← SNILUPDeterminant(M)
        r(x) ← AddErrors(r(x), M)
        M ← M[0 : −1][0 : −1]
    end
end
S ← SNISyndromeCalculation(r(x))
M ← NIPetersonMatrix(S)
Λ(x) ← SNILUPSolve(M, S)
```

Output: $\mathbf{\Lambda}(x), r(x)$

Theorem 5.2.2. *The* SNIAdaptedPeterson *is correct and produces a valid error location polynomial for the location of the initial errors and the added errors.*

Proof. To show that our adapted version of the Peterson algorithm is correct, it suffices to justify that, assuming the input had at most e errors, our algorithm never increases the total number of errors to be greater than e, and that the error

location polynomial we create is correct for the initial errors combined with the errors we have added. Our adapted Peterson algorithm works in a similar method to the normal Peterson algorithm, in the sense that it repeatedly calculates the determinant of the matrix, and as well as removing two rows and two columns also adding two errors - with the exception of the first iteration where we only add one error. As we are adding errors at the same rate with which we remove rows and columns, we add errors if and only if there are less than $e-1$ errors. Given that there are at most e errors, and that SNILUPSolve is correct, $\mathbf{\Lambda}(x)$ will be the valid error location polynomial. □

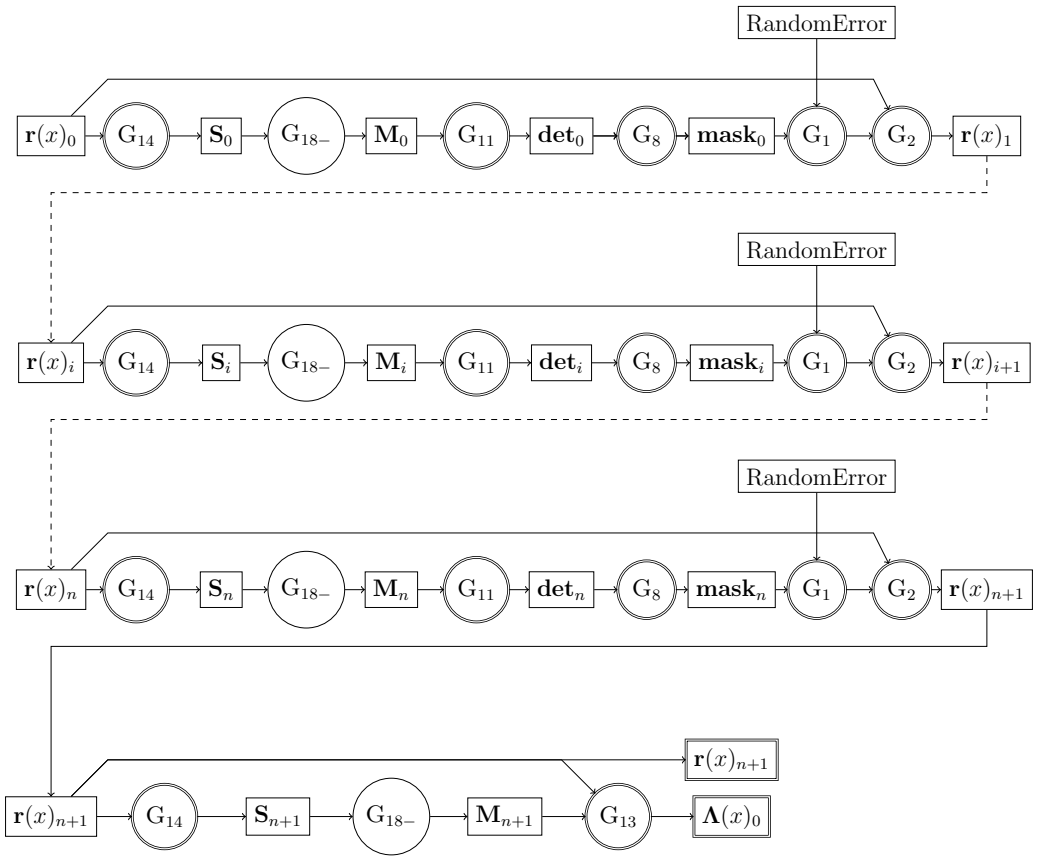

Figure 5.2: Adapted Peterson Algorithm - considered as a sequence of t-(S)NI gadgets. N.B $n = 2e-3$

Theorem 5.2.3 (t-SNI of SNIAdaptedPeterson.)**.** *Let $\boldsymbol{r}(x)_0^{(.)}$ be the input and $\mathbf{\Lambda}_0^{(.)}$ and $\boldsymbol{r}(x)_{n+1}$ be the outputs to* G_{15} *, where $\boldsymbol{r}$ is a bitstring, masked using Boolean masking and Λ is a tuple of elements from $GF(2^m)$. For any set of t' intermediate variables and any subset O of output variables such that $t' + |O| \leq t$, there exists a*

subset I of input variables such that $|I| \leq t'$ and the t' intermediate variables and the output variables can be perfectly simulated using the I input variables.

Proof. The only gadgets that make up the SNIAdaptedPeterson are SNISyndromeCalcul and G_2. As all of these gadgets are t-NI and each intermediate variable is only an input to at most one t-NI gadget, then by Proposition 1 SNIAdaptedPeterson is t-NI. Moreover because the last gadget the output passes through is t-SNI, then SNIAdaptedPeterson is also t-SNI. □

5.2.3 Chien search

When masking Chien search there are two aspects of the variables that we are trying to hide, the number of errors and the location of the errors. In order to hide the location of the corrected errors, we mask the Chien search algorithm by replacing each addition and multiplication gadget with its t-SNI equivalent. SNIAdaptedPeterson ensures that we will always have $e - 1$ or e errors, to help reduce the impact of the number of errors on the runtime, and apply masking to ensure that it is secure. In Fig. 5.3 we consider each instance of $G'_{1\_}$ to also be taking a public variable α as the second input and take each $G'_{2\_}$ to be adapted to take $2t$ inputs which it sums rather than just 2 inputs.

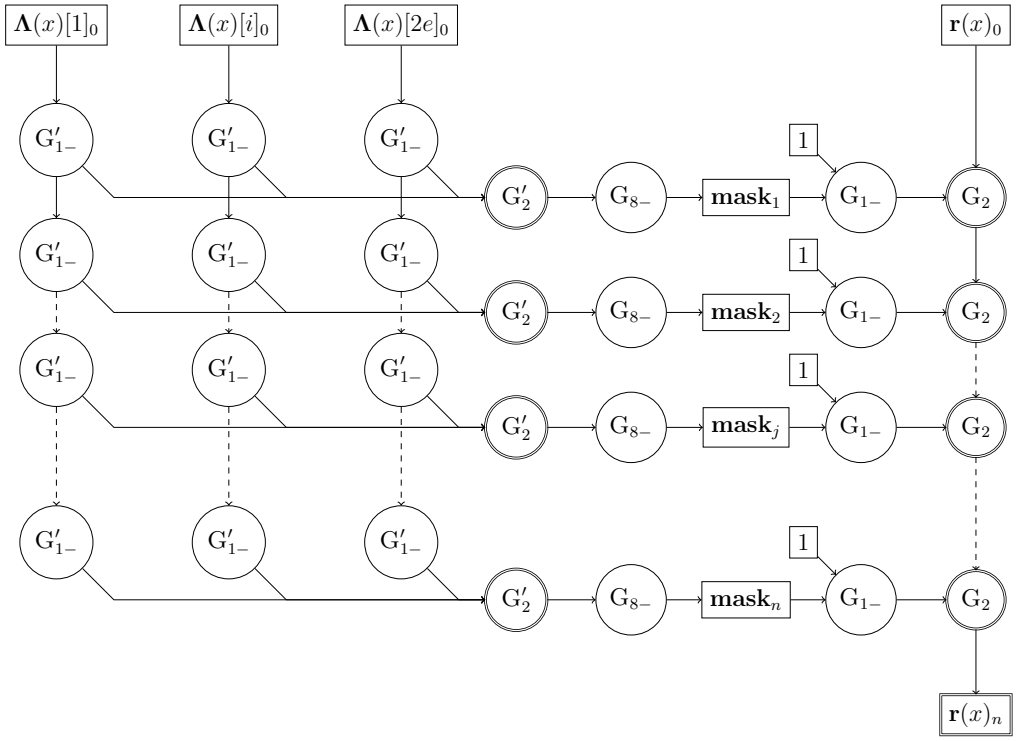

Figure 5.3: SNIChienSearch - considered as a sequence of t-(S)NI gadgets.

Theorem 5.2.4 (t-SNI of SNIChienSearch)**.** *Let $\mathbf{\Lambda}_0^{(.)}$ and $\boldsymbol{r}(x)_0$ be the inputs and $\boldsymbol{r}(x)_n$ be the output to G_{16}. For any set of t' intermediate variables and any subset O of output variables such that $t' + |O| \leq t$, there exists a subset I of input variables such that $|I| \leq t'$ and the t' intermediate variables and the output variables can be perfectly simulated using the I input variables.*

Proof. The only gadgets that make up the SNIChienSearch are NIMul, SNIAdd, NIAdd and G_8. As all of these gadgets are t-NI and the only time an intermediate variable is reused is the output of each G'_{1-} which are only used as input to at most one t-NI gadget, then by Proposition 1 SNIChienSearch is t-NI. Moreover because the last gadget the output passes through is t-SNI, then SNIChienSearch is also t-SNI. □

5.2.4 BCH decoding

We now present theoretical security proofs for the entire BCH decoding process, as well as give some experimental validation for our first order masked implementation.

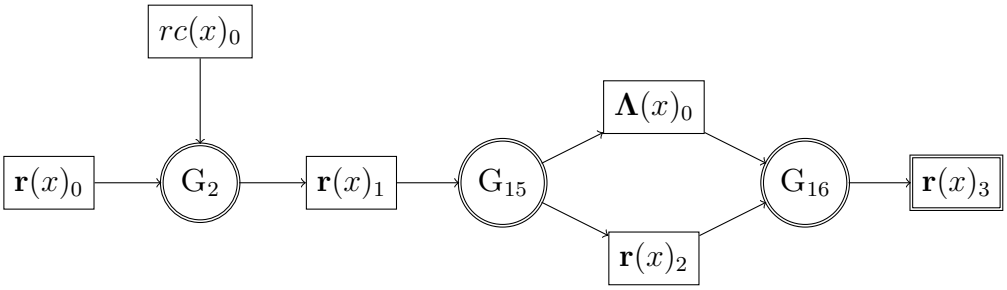

Figure 5.4: SNIBCHDecoding - considered as a sequence of t-SNI gadgets.

Theorem 5.2.1 (t-SNI of SNIBCHDecoding)**.** *Let $\boldsymbol{r}(x)_0$ be the input and $\boldsymbol{r}(x)_n$ be the output to G_{17}. For any set of t' intermediate variables and any subset O of output variables such that $t' + |O| \leq t$, there exists a subset I of input variables such that $|I| \leq t'$ and the t' intermediate variables and the output variables can be perfectly simulated using the I input variables.*

Proof. The only gadgets that make up SNIBCHDecoding are SNIAdd, SNIAdapted-Peterson and SNIChienSearch. As all of these gadgets are t-SNI and each intermediate variable is only an input to at most one t-NI gadget, then by Proposition 1 SNIBCHDecoding is t-NI. Moreover because the last gadget the output passes through is t-SNI, then SNIBCHDecoding is also t-SNI. □

5.2.5 Experimental results

For the experiments we have take the first order masked version of the implementation. For each component of the decoding process and for the entire decoding process we collect 1000 traces for running the component on the all 0 codeword and 1000 traces for running the component on the all 0 codeword with 8 errors all in the same byte. All traces were collected using ELMO [MOW17], with the ELMO energy model flag set rather than the Hamming Weight model flag. Using these we perform Welch's t-test and, setting the threshold to ± 5.730 [DZD+18], we display the results below. From this we can see that our decoding algorithm is secure against power analysis attacks.

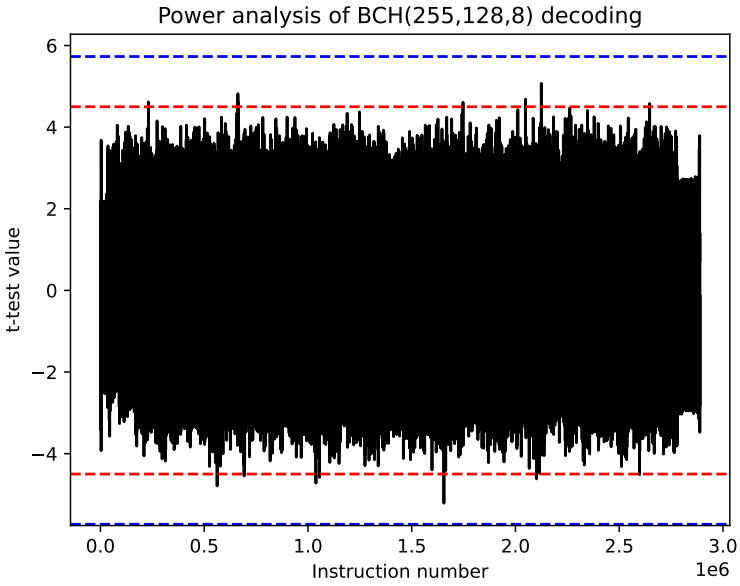

Figure 5.5: TVLA results for BCH(255,128,8) code between BCH code decoding for the all 0 codeword, and the all 0 codeword with 8 errors all in the same byte for our masked implementation. The inner dashed line is at ±4.5 and the outer dashed line is at ±5.730.

We give a brief overview of the time and randomness complexity of the smaller gadgets in Table 5.1. The complexity of G_8 depends on the bit length of the values being compared, we focus on the case where all values are 8 bits. INVERSE's complexity depends on the size of the finite field used, for this example we use $GF(2^8)$.

Gadget	$\mathcal{T}$	$\mathcal{R}$
SNIMul/SNIAnd	$3t^2$	$\frac{t^2}{2}$
NIMul/NIAnd	$\frac{7t^2}{4}$	$\frac{t^2}{4}+t$
SNIRef	t^2	$\frac{t^2}{2}$
NIRef	$2t$	t
SNIAdd/SNIXOR	t^2+t	$\frac{t^2}{2}$
NIAdd/NIXOR	t	–
NINot	1	–
SNIInv	$12t^2+3t$	$3t^2+2t$
NISwap	t	–
G_8	$44t^2+6t$	$16t^2+15t$
SNISyn	$\frac{7nt^2}{4}+t(n-1)+3t^2$	$\frac{nt^2}{4}+\frac{t^2}{2}$
SNIChien	$(2ne+n)(\frac{11t^2+4t}{4})$ $+n(44t^2+6t)$	$(2ne+n)(\frac{3t^2}{4}+t)$ $+n(16t^2+15t)$

Table 5.1: The time and randomness complexity of small gadgets.

We now discuss the time and randomness complexity of BCH code decoding, where e is the maximum number of errors that can be corrected.

We estimate the time complexity of SNIBCHDecoding, $\mathcal{T}_{\text{BCH}}$, as

$$\begin{aligned}
\mathcal{T}_{\text{BCH}} &= \mathcal{T}_{\text{Syn}} + \mathcal{T}_{\text{G}_{15}} + \mathcal{T}_{\text{SNIChien}} \\
&= \mathcal{T}_{\text{Syn}} + (e-2)(\mathcal{T}_{\text{Syn}}) + (e-2)(e-1)(\mathcal{T}_{\text{LUP}} + \mathcal{T}_{\text{Det}}) + \mathcal{T}_{\text{LUP}} + \mathcal{T}_{\text{Solve}} + \mathcal{T}_{\text{SNIChien}} \\
&= O(e\mathcal{T}_{\text{Syn}} + e^2(\mathcal{T}_{\text{LUP}} + \mathcal{T}_{\text{Det}}) + \mathcal{T}_{\text{Solve}} + \mathcal{T}_{\text{SNIChien}}) \\
&= O(net^2 + e^2(e^3t^2 + et^2) + e^2t^2 + net^2) \\
&= O(net^2 + e^5t^2)
\end{aligned}$$

We estimate the randomness complexity of SNIBCHDecoding, $\mathcal{R}_{\text{BCH}}$, as

$$\begin{aligned}\mathcal{R}_{\text{BCH}} &= \mathcal{R}_{\text{Syn}} + \mathcal{R}_{\text{G}_1 5} + \mathcal{R}_{\text{SNIChien}} \\ &= O(e\mathcal{R}_{\text{Syn}} + e^2(\mathcal{R}_{\text{LUP}} + \mathcal{R}_{\text{Det}}) + \mathcal{R}_{\text{Solve}} + \mathcal{R}_{\text{SNIChien}}) \\ &= O(net^2 + e^2(e^3t^2 + et^2) + e^2t^2 + net^2) \\ &= O(net^2 + e^5t^2)\end{aligned}$$

Whilst it is clear that the masked implementation has a larger overhead than would be optimal for masking over the standard implementation, two sections of the algorithm - syndrome calculation and Chien search - are optimal. Both the syndrome calculation and Chien search are used as part of several decoding algorithms, including for Reed-Solomon and Reed-Muller code, and are used independently of the use of the Peterson algorithm or the Berlekamp-Massey algorithm. The Peterson algorithm has the largest overhead and, whilst it is inefficient, we hope that this initial masked implementation will encourage further work to optimise it.

CHAPTER 6

Securing polar codes

In the previous chapter we discussed a method for securing the BCH code decoding algorithm against side-channel attacks. Whilst BCH codes are very strong error correcting codes and are efficient in terms of space, the Peterson decoding algorithm is inefficient in terms of run time ($O(e^3+n)$, and $\Theta(e^4t^2+nt^2)$ when masked). Even faster algorithm for decoding BCH codes, such as the Berlekamp-Massey algorithm, have a run time of $O(e^2+n)$. Polar codes are a type of soft error correcting codes, that are optimal for some types of channel, and have a decoding algorithm that runs in $O(n^2)$ time naively and can be optimised to $O(n \log n)$. Note however that the dominant component of BCH code decoding is in terms of e - the number of errors that the code can decode, whilst for polar codes the dominant term is n - the length of the codeword.

Similarly to the previous chapter, we will present a masked version of the Polar code decoding algorithm and present security proofs for it in the probing model. We also verify the security of the first order masked implementation experimentally by generating traces using ELMO [MOW17] and assessing the leakage using the TVLA [GGJR+11].

We make use of the following gadgets:

G_1	SNIMul	a t-SNI([BBD+16]) secure algorithm for multiplication. This is converted to work for $\mathbb{Q}_1$ in Section 6.3.2.
G_{1-}	NIMul	a t-NI([BBP+16]) secure algorithm for multiplication. This is converted to work for $\mathbb{Q}_1$ in Section 6.3.2.
$\hat{G}_1$	SNIAnd	a t-SNI([BBD+16]) secure algorithm for logical And. This is converted to work for $\mathbb{Q}_1$ in Section 6.3.2.
$\hat{G}_{1-}$	NIAnd	a t-NI([BBP+16]) secure algorithm for logical And. This is converted to work for $\mathbb{Q}_1$ in Section 6.3.2.
G_2	SNIAdd	a t-SNI(Lemma 4.1.1) secure algorithm for addition. This is converted to work for $\mathbb{Q}_1$ in Section 6.3.2.
G_{2-}	NIAdd	a t-NI secure algorithm for addition. This is converted to work for $\mathbb{Q}_1$ in Section 6.3.2.
$\hat{G}_2$	SNIXOR	a t-SNI(Lemma 4.1.1) secure algorithm for logical XOR. This is converted to work for $\mathbb{Q}_1$ in Section 6.3.2.
$\hat{G}_{2-}$	NIXOR	a t-NI secure algorithm for logical XOR. This is converted to work for $\mathbb{Q}_1$ in Section 6.3.2.
G_8	SNIMagnitude Comparator	a t-SNI(Lemma 4.1.3) secure algorithm for returning the shared result of $a \leq b$.
G_9	SNIRef	a t-SNI([BBD+16]) secure algorithm for mask refreshing.
G_O	SNIOdd	a t-SNI(Lemma 6.4.2) secure algorithm for Line 7
G_E	SNIEven	a t-SNI(Lemma 6.4.3) secure algorithm for Line 9
G_{21}	SNIArithmetic	a t-SNI(Lemma 6.4.4) secure algorithm that makes up Algorithm 6.2
G_{22}	SNIBoolean	a t-SNI(Lemma 6.4.1) secure algorithm that makes up Algorithm 6.2
G_{23}	SNIL	a t-SNI(Theorem 6.5.1) secure algorithm for Algorithm 6.2
G_{24}	SNIDecoding	a t-SNI(Theorem 6.5.1) secure algorithm for Algorithm 6.1

6.1 Decoding polar codes

The decoding algorithm for polar codes has two main components, an iterative method (Algorithm 6.1) that iterates over each bit in the received codeword and

Algorithm 6.1: DECODING

Input: $\hat{\mathbf{cw}}, \mathbf{p}$

$\hat{\mathbf{u}} \leftarrow [\,]$
for $i \leftarrow 1$ **to** N **do**
  if i is a frozen bit **then**
    $\mathbf{u}[i] \leftarrow 0$
  else
    $r \leftarrow \mathrm{L}(i, N, \hat{\mathbf{cw}}, \hat{\mathbf{u}}, \mathbf{p})$
    if $r \geq 1$ **then**
      $\mathbf{u}[i] \leftarrow 0$
    else
      $\mathbf{u}[i] \leftarrow 1$
    end
  end
end

Output: $\hat{\mathbf{u}}$

Algorithm 6.2: L

Input: $i, N, \hat{\mathbf{cw}}, \hat{\mathbf{u}}, \mathbf{p}$

if $N = 1$ **then**
  Output: $p[i]$
else
  $i' \leftarrow (i+1)//2$
  $a \leftarrow \mathrm{L}(i', N/2, \hat{\mathbf{cw}}[1 : N/2], \hat{\mathbf{u}}[1 : i-1]_o \oplus \hat{\mathbf{c}}[1 : i-1]_e, \mathbf{p})$
  $b \leftarrow \mathrm{L}(i', N/2, \hat{\mathbf{cw}}[N/2+1 : N], \hat{\mathbf{u}}[1 : i-1]_e, \mathbf{p})$
  if i is odd **then**
    $r \leftarrow \frac{ab+1}{a+b}$
  else
    $r \leftarrow a^{1-2\hat{\mathbf{u}}[i-1]}b$
  end
end

Output: r

calls a recursive algorithm L (Algorithm 6.2) to calculate the value of the bit. The recursive algorithm L can be seen as working through the polarisation algorithm and trying to establish the output and the other input of the XOR gate that it is an input to.

6.2 Method

There are three main weaknesses in the polar code decoding algorithm that we first raise before discussing how to secure the algorithm. The first issue is that it is

possible to pick specific codewords that maximise the statistical differences in power usage, regardless of how many errors there are, and the attacker can pick any two messages to maximise this. The second issue is that the decoding algorithm uses division, which can't be converted into a secure gadget since we are not using a finite field. Finally standard gadgets are only designed for finite fields, whereas the decoding algorithm is over the reals.

We make a number of changes to the decoding algorithm in order to make it secure. First we blind the input by xoring a randomly generated codeword with our received codeword and edit the likelihood ratio vector accordingly. Secondly we edit the L function to make it inversionless, this can be done simply by considering the numerator and denominator separately. Finally we create floating point versions of several small gadgets.

We give the masked version of the masked SNIDecoding algorithm (Algorithm 6.3) and the masked SNIL (Algorithm 6.4). In order to be concise we represent the masked version of an operation by putting a box around the relevant symbol, e.g. replacing $\cdot$ with $\boxdot$.

Finally in order to better argue about how shares are reused we convert the recursive algorithm L into a group of circuits of simple gadgets.

6.3 Masking polar codes

In this section we present the changes that are made to the algorithm in more detail and argue about their security.

6.3.1 Blinding

Unlike with BCH codes where the parity bits are separate to the data bits, in the polar code all the bits are intermixed during the polarization process. This, combined with the use of LRs, makes the blinding process a little bit more involved. We give the algorithms for blinding and unblinding the received codeword below in Algorithms 6.5 and 6.6. In order to generate a codeword to blind, we first choose a random string, **rw**, which we polarise to give the codeword **rcw**. **rcw** is xored with

Algorithm 6.3: SNIDecoding

Input: $\hat{\mathbf{cw}}, \mathbf{p}$
$\hat{\mathbf{u}} \leftarrow [\,]$
$(\hat{\mathbf{cw}}, \mathbf{p}) \leftarrow \text{Blind}(\hat{\mathbf{cw}}, \mathbf{p})$
for $i \leftarrow 1$ **to** N **do**
  if i is a frozen bit **then**
    $\hat{\mathbf{u}}[i] \leftarrow 0$
  else
    $(\mathbf{r}_\text{N}, \mathbf{r}_\text{D}) \leftarrow \text{SNIL}(i, N, \hat{\mathbf{cw}}, \hat{\mathbf{u}}, \mathbf{p})$
    $\hat{\mathbf{u}}[i] \leftarrow \text{SNIMagnitudeComparator}(\mathbf{r}_N, \mathbf{r}_D)$
  end
end
$\hat{\mathbf{cw}} \leftarrow \text{Unblind}(\hat{\mathbf{cw}})$
Output: $\hat{\mathbf{u}}$

Algorithm 6.4: SNIL

Input: $i, N, \hat{\mathbf{cw}}, \hat{\mathbf{u}}, \mathbf{p}$
if $N = 1$ **then**
  Output: $\mathbf{p}[1]$
else
  $ip \leftarrow (i+1)//2$
  $(\mathbf{a}_\text{N}, \mathbf{a}_\text{D}) \leftarrow \text{SNIL}(ip, N/2, \hat{\mathbf{cw}}[1:N/2], \hat{\mathbf{u}}[1:2i-1]_o \boxplus \hat{\mathbf{u}}[1:2i-1]_e, \mathbf{p}[1:N/2])$
  $(\mathbf{b}_\text{N}, \mathbf{b}_\text{D}) \leftarrow \text{SNIL}(ip, N/2, \hat{\mathbf{cw}}[N/2+1:N], \hat{\mathbf{u}}[1:2i-2]_e, \mathbf{p}[N/2+1:N])$
  if i is odd **then**
    $\mathbf{r}_\text{N} \leftarrow (\mathbf{a}_\text{N} \boxdot \mathbf{b}_\text{N}) \boxplus (\mathbf{a}_\text{D} \boxdot \mathbf{b}_\text{D})$
    $\mathbf{r}_\text{D} \leftarrow (\mathbf{a}_\text{N} \boxdot \mathbf{b}_\text{D}) \boxplus (\mathbf{a}_\text{D} \boxdot \mathbf{b}_\text{N})$
  else
    $\mathbf{r}_\text{N} \leftarrow \mathbf{a}_\text{N}^{1-2\hat{\mathbf{u}}[2i-1]} \boxdot \mathbf{b}_\text{N}$
    $\mathbf{r}_\text{D} \leftarrow \mathbf{a}_\text{D}^{1-2\hat{\mathbf{u}}[2i-1]} \boxdot \mathbf{b}_\text{D}$
  end
end
Output: $(\mathbf{r}_\text{N}, \mathbf{r}_\text{D})$

Algorithm 6.5: BLIND

Input: $\hat{\mathbf{cw}}, \mathbf{p}$

$\mathbf{rw} \leftarrow \text{RANDOMWORD}()$
$\mathbf{rcw} \leftarrow \text{POLARIZE}(\mathbf{rw})$
$\hat{\mathbf{cw}} \leftarrow \hat{\mathbf{cw}} \boxplus \mathbf{rcw}$
for $i \leftarrow 1$ **to** N **do**
  for $j \leftarrow 1$ **to** t **do**
    if $\mathbf{rcw}[i]^{(j)} = 1$ **then**
      $\mathbf{p}[i]_{\mathrm{N}} \leftarrow -\mathbf{p}[i]_{\mathrm{N}} \mod 1$
      $\mathbf{p}[i]_{\mathrm{D}} \leftarrow -\mathbf{p}[i]_{\mathrm{D}} \mod 1$
  end
end

Output: $\hat{\mathbf{cw}}, \mathbf{p}$

Algorithm 6.6: UNBLIND

Input: $\hat{\mathbf{cw}}$

$\hat{\mathbf{cw}} \leftarrow \hat{\mathbf{cw}} \boxplus \mathbf{rw}$

Output: $\hat{\mathbf{cw}}$

the received codeword to blind it, we then also need to update the LRs to take into account that we've flipped some of the bits. We update the LRs by iterating over the shares of **rcw** and if the jth share of the ith bit is 1 then we replace each share of the LR for that bit with 1 minus it's value. This effectively swaps the numerators and denominators around. Whilst this does leak the random codeword we use for blinding, this doesn't leak any information about the decoding and still stops the adversary having control over the codeword used.

6.3.2 Security of small gadgets

One of the challenges with masking polar codes is that we can't apply standard arithmetic masking, which is primarily designed for $\mathbb{Z}_q$. Rather than using $\mathbb{Z}_q$ we consider all of the arithmetic values to be members of $\mathbb{Q}/\mathbb{Z}$, which we will refer to as $\mathbb{Q}_1$. We discretise the $\mathbb{Q}_1$ to the values that can be represented by a C float, we represent this by $\lfloor \rceil_g$, where g represents the size of the float. It's easy to see that this gives us $\mathbb{Z}_{2^g}/2^g$. Whilst normally floats would be avoided, due to their susceptibility to timing attacks, the discretisation that we use turns this into an implementation of fixed point decimals. We give the altered algorithm for SECREF

in Algorithm 6.7. The changes to SecMul and other small gadgets are similar. Due to the use of fixed point decimals, there is the potential for rounding errors. Whilst our correctness experiments for first order masking, $N = 8$ and $g = 16$ did not reveal any errors that led to a decoding failure. However larger values of N and higher order masking may lead to more rounding errors and potential decoding failures, increasing the granularity g can reduce this risk.

Algorithm 6.7: SNIRef

Input: $\mathbf{a}_0 \in \mathbb{Q}_1^{t+1}$

for $i \leftarrow 0$ **to** 2 **do**
  $\mathbf{c}_0^{(i)} \leftarrow \lfloor \mathbf{a}_0^{(i)} \rceil_g$
end
for $i \leftarrow 0$ **to** t **do**
  for $j \leftarrow 1$ **to** t **do**
    $b_{(t+1)i+j} \xleftarrow{\$} [0, 2^g)$
    $r_{(t+1)i+j} \leftarrow \frac{b_{(t+1)i+j}}{2^g}$
    $\mathbf{c}^{(i)}_{(t+1)i+j} \leftarrow \mathbf{c}^{(i)}_{(t+1)i+j-1} + r_{(t+1)i+j}$
    $\mathbf{c}^{(j)}_{(t+1)i+j} \leftarrow \mathbf{c}^{(j)}_{(t+1)i+j-1} - r_{(t+1)i+j}$
  end
end

Output: $\mathbf{c}_{t^2+2t}$

6.3.3 Making L division free

Since we can't create a masked division gadget, we instead edit the SNIL algorithm to remove divisions. We separate the LRs into the numerator and the denominator, and perform all operations on them independently. In order to do this, we also have to modify the calculations we do. We replace $r = \frac{ab+1}{a+b}$ with

$$r_{\mathrm{N}} = (a_{\mathrm{N}} \cdot b_{\mathrm{N}}) + (a_{\mathrm{D}} \cdot b_{\mathrm{D}})$$

and

$$r_{\mathrm{D}} = (a_{\mathrm{N}} \cdot b_{\mathrm{D}}) + (a_{\mathrm{D}} \cdot b_{\mathrm{N}}).$$

We replace $r = a^{\{1,-1\}}b$ with

$$r_{\mathrm{N}} = a_{\mathrm{N}}b_{\mathrm{N}}, r_{\mathrm{D}} = a_{\mathrm{D}}b_{\mathrm{D}}$$

or

$$r_{\mathrm{N}} = a_{\mathrm{D}}b_{\mathrm{N}}, r_{\mathrm{D}} = a_{\mathrm{N}}b_{\mathrm{D}}$$

as appropriate. The final check as part of SNIDecoding is changed from if $r > 1$ to if $r_{\mathrm{D}} > r_{\mathrm{N}}$.

6.3.4 Calculating the likelihood ratios

We would normally assume that the likelihood ratios for each bit would be an input to the decoding algorithm, however to better integrate it with current LWE-based cryptosystems we consider an approximate method to create the masked likelihood ratios. We approximate for the likelihood ratio of a LWE-based scheme where the coefficient $\hat{x}$ is drawn from some distribution $\mathcal{D}$ with a support $0, q)$ as $P(x = 1|\hat{x}) = \left|\frac{\hat{x}}{q/2}\right|, P(x = 0|\hat{x}) = 1 - \left|\frac{\hat{x}}{q/2}\right|$.

6.4 Converting to circuits

Since SNIL is recursive, it is very difficult to directly consider the interactions between shares as the algorithm recurses. In order to better argue about this we convert the overall recursive tree into two circuits, a Boolean circuit and an arithmetic circuit.

The Boolean circuit represents how the codeword is split up and xored together. For each call to SNIL we have N different Boolean circuits, each one representing a different path through the recursion tree. The Boolean circuit has the first i bits of $\hat{\mathbf{u}}$ as its input. At each stage it either xors $\hat{\mathbf{u}}_o$ with $\hat{\mathbf{u}}_e$ or it only moves $\hat{\mathbf{u}}_e$ onto the next round, dependent on if the recursion tree that this Boolean circuit represents recursed down either the left or the right branch. We give an example of the Boolean circuits produced for $N = 4, i = 4$ in Fig. 6.1.

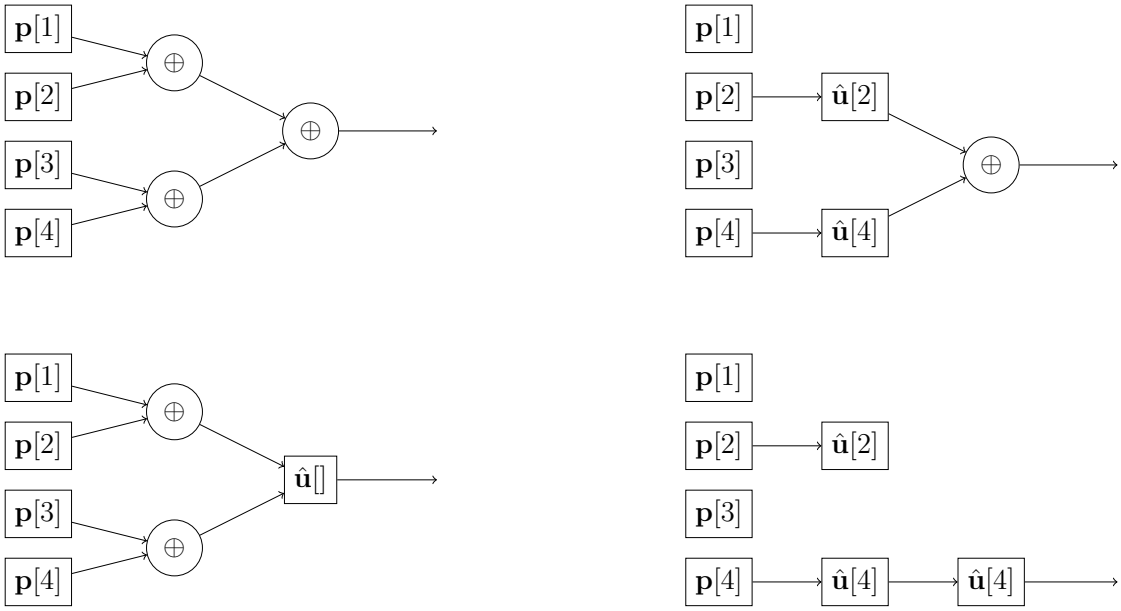

Figure 6.1: The Boolean circuits for $N = 4, i = 4$.

The arithmetic circuit follows a similar idea, except rather than it representing the operations that happen to the codeword, it represents how the likelihood ratios for each input bit are combined to get the final likelihood ratio for the corresponding codeword bit. For each initial call to L we have just one arithmetic circuit, which takes as input the likelihood ratios for each bit, and combines the likelihood ratios using either the odd arithmetic gadget, $\mathrm{G_O}$, or the even arithmetic gadget, $\mathrm{G_E}$, depending on the value of i corresponding to that recursive call.

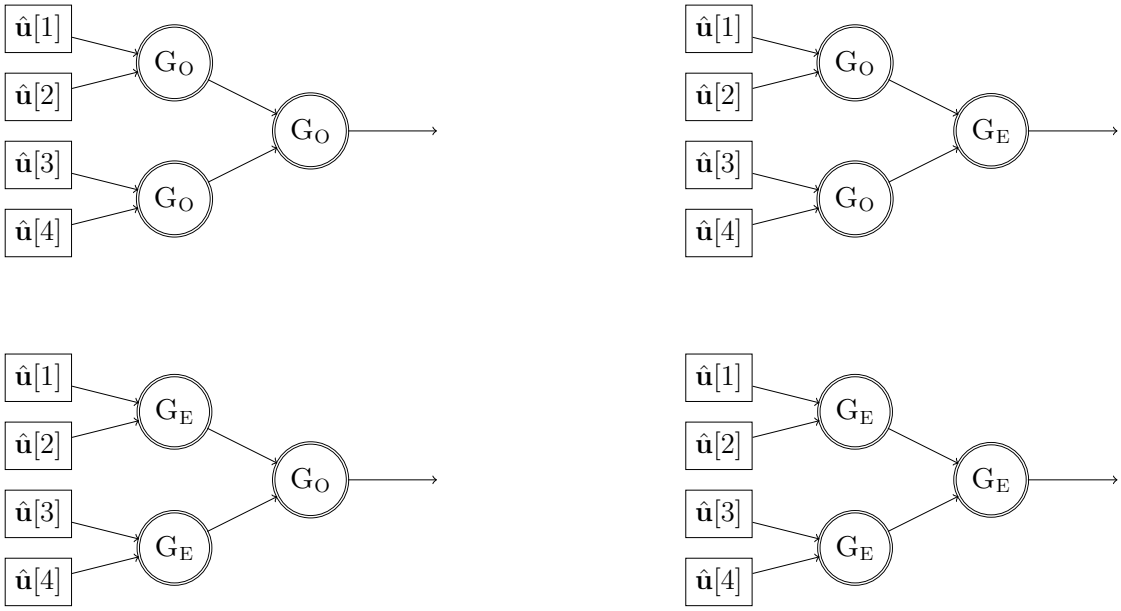

Figure 6.2: The arithmetic circuits for $N = 4$, with $i = 1$ and $i = 2$ on the top row, and $i = 3$ and $i = 4$ on the bottom row.

The Boolean circuit is completely independent of the arithmetic circuit. The arithmetic circuit does take a value from the Boolean circuit when the odd arithmetic gadget is used, however it does not change which shares are used. For these reasons we can argue about the security of the masked versions of each of these circuits separately.

6.4.1 Boolean circuit

The Boolean circuit is just the reverse of the polarization circuit, and so consists only of XOR gates. We can replace these with their t-NI secure masked counterpart NIXOR. Whilst the input to the circuit is a binary string, the output is always a single bit.

The Boolean circuit component of the decoding algorithm will always consist of a binary forest, with each leaf being a bit from $\hat{\mathbf{u}}$, and the outputs can be the output of any of the NIXOR gates. A mask refresh is applied before the result is outputted.

Theorem 6.4.1 (t-SNI of the Boolean circuit BC)**.** *Let* $\hat{\mathbf{u}}[0:i]_0^{(.)}$ *be the input and* $\mathbf{x}_0^{(.)}$ *be the outputs to* BC*. For any set of* t' *intermediate variables and any subset* O

of output variables such that $t' + |O| \leq t$, there exists a subset I of input variables such that $|I| \leq t'$ and the t' intermediate variables and the output variables can be perfectly simulated using the I input variables.

Proof. The circuit consists of just two types of gates, NIXOR, which is t-NI secure, and SNIREF, which is t-SNI secure. The only time an intermediate variable is used more than once is when it is both outputted and also used for the next XOR gate, since we apply a mask refresh before the variable is outputted, each variable is used at most once by a non-t-SNI gadget. Therefore by Proposition 1 the Boolean circuit is t-NI. Moreover because each output passes through a SNIREF gadget immediately prior to being outputted it is also t-SNI. □

6.4.2 Arithmetic circuits

In order to mask the entire recursive algorithm we first consider the two different computations that are done, we dub these the odd arithmetic gadget and the even arithmetic gadget. The odd arithmetic gadget calculates $\frac{ab+1}{a+b}$ and the even gadget computes $a^{1-2u[i-1]}b$. Normally both of these computations contain division, however to make it easier to mask the algorithm we apply the computation to the numerator and denominator separately. This is done by splitting a into a_{N} and a_{D}, and simplifying as follows.

$$\begin{aligned}\frac{ab+1}{a+b} &= \frac{\frac{a_{\mathrm{N}}b_{\mathrm{N}}}{a_{\mathrm{D}}b_{\mathrm{D}}} + 1}{\frac{a_{\mathrm{N}}}{a_{\mathrm{D}}} + \frac{b_{\mathrm{N}}}{b_{\mathrm{D}}}} \\ &= \frac{\frac{a_{\mathrm{N}}b_{\mathrm{N}}}{a_{\mathrm{D}}b_{\mathrm{D}}} + \frac{a_{\mathrm{D}}b_{\mathrm{D}}}{a_{\mathrm{D}}b_{\mathrm{D}}}}{\frac{a_{\mathrm{N}}b_{\mathrm{D}}}{a_{\mathrm{D}}b_{\mathrm{D}}} + \frac{a_{\mathrm{D}}b_{\mathrm{N}}}{a_{\mathrm{D}}b_{\mathrm{D}}}} \\ &= \frac{a_{\mathrm{N}}b_{\mathrm{N}} + a_{\mathrm{D}}b_{\mathrm{D}}}{a_{\mathrm{N}}b_{\mathrm{D}} + a_{\mathrm{D}}b_{\mathrm{N}}}\end{aligned}$$

Typically this would cause the values being masked to increase in size with each multiplication, requiring the masks to increase in size at a commensurate rate, to avoid this we perform a semi-private division after the computation is complete, ensuring that the inputs and outputs are both of the same fixed size. Specifically for the even arithmetic circuit we avoid raising $\mathbf{a}$ to the power of either a masked 1

or a masked -1, we apply standard techniques to switch between $\frac{\mathbf{a}_N}{\mathbf{a}_D}$ and $\frac{\mathbf{a}_D}{\mathbf{a}_N}$.

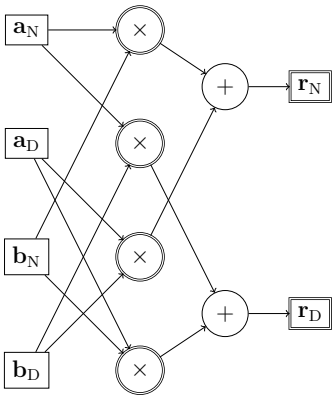

Figure 6.3: The odd arithmetic gadget.

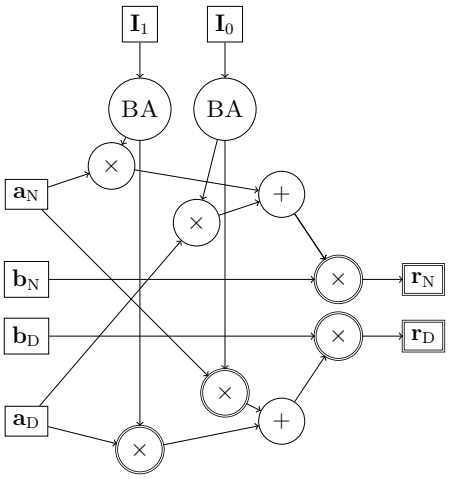

Figure 6.4: The even arithmetic gadget.

Theorem 6.4.2 (t-SNI of the odd arithmetic gadget ODD)**.** *Let* $\hat{\mathbf{a}}^{(.)}_{D,0}$,$\hat{\mathbf{a}}^{(.)}_{N,0}$,$\hat{\mathbf{b}}^{(.)}_{D,0}$,$\hat{\mathbf{b}}^{(.)}_{N,0}$ *be the inputs and* $\hat{\mathbf{r}}^{(.)}_{D,0}$,$\hat{\mathbf{r}}^{(.)}_{N,0}$ *be the outputs to* ODD*. For any set of* t' *intermediate variables and any subset* O *of output variables such that* $t' + |O| \leq t$*, there exists a subset* I *of input variables such that* $|I| \leq t'$ *and the* t' *intermediate variables and the output variables can be perfectly simulated using the* I *input variables.*

Proof. The circuit consists of three types of gates, addition and semi-public division, which are t-NI secure, and multiplication, which is t-SNI secure. Each input is used twice, and each time it used as an input to t-SNI secure multiplication. Therefore by Proposition 1 the Boolean circuit is t-NI. Moreover because each input passes through a t-SNI gadget initially, it is therefore also t-SNI. The gadget could be optimised further and remain t-NI by replacing two of the t-SNI multiplication gadgets, e.g. the first and third, with the t-NI versions. □

Theorem 6.4.3 (t-SNI of the even arithmetic gadget EVEN)**.** *Let* $\hat{\mathbf{a}}^{(.)}_{D,0}$, $\hat{\mathbf{a}}^{(.)}_{N,0}$, $\hat{\mathbf{b}}^{(.)}_{D,0}$, $\hat{\mathbf{b}}^{(.)}_{N,0}$, $\hat{\mathbf{I}}^{(.)}_{1,0}$, $\hat{\mathbf{I}}^{(.)}_{0,0}$ *be the inputs and* $\hat{\mathbf{r}}^{(.)}_{D,0}$, $\hat{\mathbf{r}}^{(.)}_{N,0}$ *be the outputs to* SNIEVEN*. For any set of* t' *intermediate variables and any subset* O *of output variables such that* $t' + |O| \leq t$*, there exists a subset* I *of input variables such that* $|I| \leq t'$ *and the* t' *intermediate variables and the output variables can be perfectly simulated using the* I *input variables.*

Proof. The circuit consists of four types of gates, addition, which is t-NI secure, multiplication , logical AND, and B2A conversion which are t-SNI secure. Each input, except $\mathbf{b}$, is used twice, and only copy of each input is used by a t-NI gadget (NIAND) with the remaining copies being inputs to the t-SNI secure SNIAND. Therefore by Proposition 1 the arithmetic circuit is t-NI. Moreover because each input passes through a t-SNI gadget before it is used, it is therefore also t-SNI. The gadget could be optimised further and remain t-NI by replacing the two t-SNI multiplication gadgets with their t-NI counterparts. □

6.4.3 Overall

Having looked at the individual SNIODD and SNIEVEN gadgets, we can now consider the entire recursive algorithm. For each i, the recursive algorithm can be considered as a tree created from the individual circuits, where each leaf represents the base case and therefore the inputs to the function, and the root represents the first call and therefore the output. The base case consists of returning the likelihood ratio tuple $(\mathbf{p}[j]_{\mathrm{N}}, \mathbf{p}[j]_{\mathrm{D}})$, and all internal nodes represent the use of either the odd or even arithmetic gadget to combine the outputs of it's two children.

Theorem 6.4.4 (t-SNI of the arithmetic circuit SNIARITHMETIC)**.** *Let* $\hat{\mathbf{u}}[0:i]_0^{(\cdot)}$ *be the input and* $\mathbf{x}_0^{(\cdot)}$ *be the outputs to* SNIARITHMETIC*. For any set of* t' *intermediate variables and any subset* O *of output variables such that* $t' + |O| \leq t$*, there exists a subset* I *of input variables such that* $|I| \leq t'$ *and the* t' *intermediate variables and the output variables can be perfectly simulated using the* I *input variables.*

Proof. For each i the arithmetic circuit is a tree with N leaves, where each leaf is a pair $(\mathbf{a}_{\mathrm{N}}, \mathbf{a}_{\mathrm{D}})$ from the array $\mathbf{p}$, each internal node is either SNIODD or SNIEVEN, both of which are t-SNI. Where each internal node is SNIEVEN, and the $\mathbf{I}$ input is taken from the Boolean circuit SNIBOOLEAN. Since every gadget is t-SNI then by Proposition 1 the entire gadget is t-SNI. □

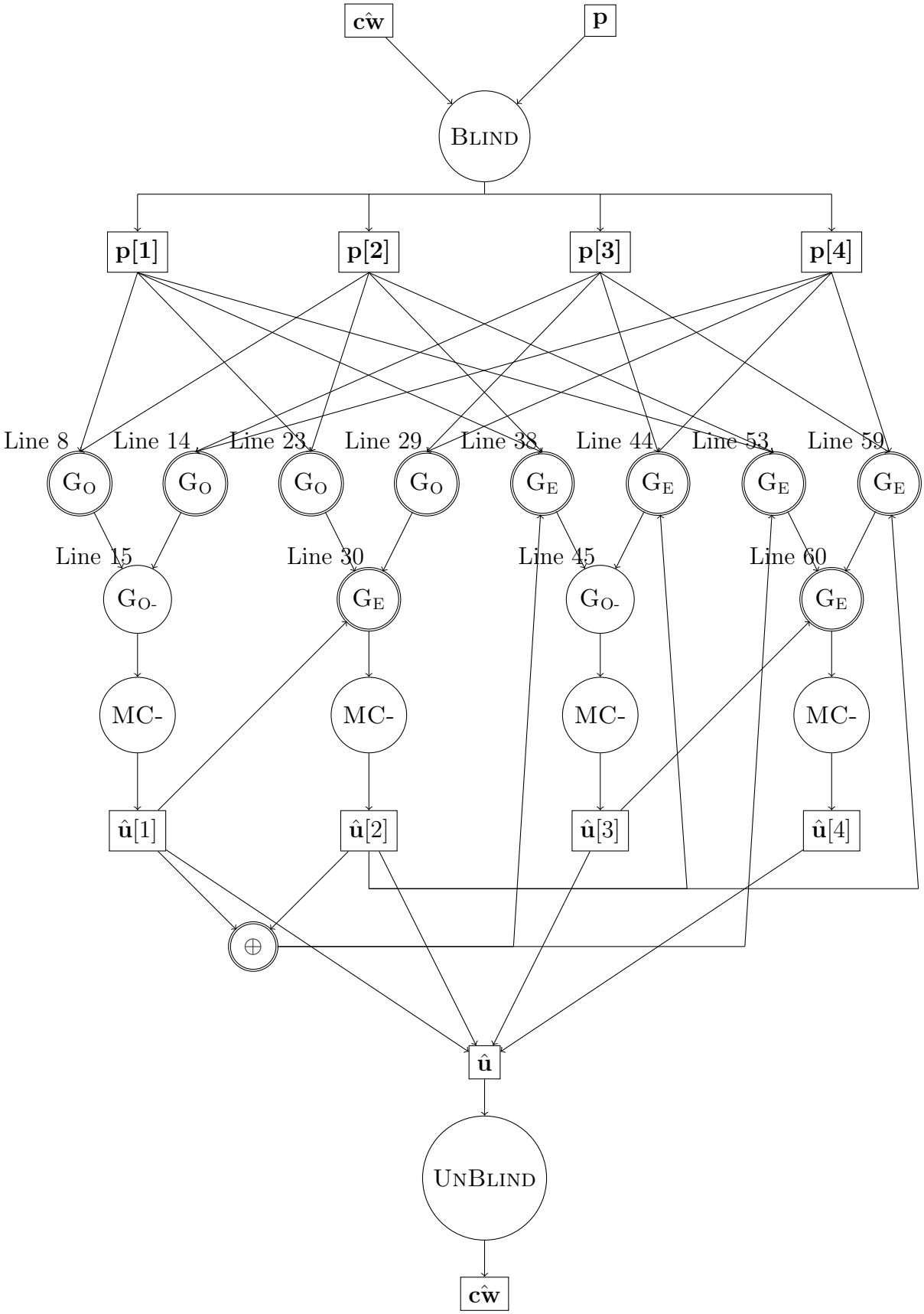

Figure 6.5: The full circuit diagram for Polar code decoding for $N = 4$

$i \leftarrow 1$
$r \leftarrow \mathrm{L}(i = 1, N = 4, \mathbf{c\hat{w}}, \hat{\mathbf{u}} = [\,], p)$
  $a \leftarrow \mathrm{L}(i = 1, N = 2, \mathbf{c\hat{w}}[1:3], \hat{\mathbf{u}} = [\,], p)$
    $a \leftarrow \mathrm{L}(i = 1, N = 1, \mathbf{c\hat{w}}[1], \hat{\mathbf{u}} = [\,], p)$
      Return $p[1]$
    $b \leftarrow \mathrm{L}(i = 1, N = 1, \mathbf{c\hat{w}}[2], \hat{\mathbf{u}} = [\,], p)$
      Return $p[2]$
    Return $\frac{p[1]p[2]+1}{p[1]+p[2]}$
  $b \leftarrow \mathrm{L}(i = 1, N = 2, \mathbf{c\hat{w}}[3:], \hat{\mathbf{u}} = [\,], p)$
    $a \leftarrow \mathrm{L}(i = 1, N = 1, \mathbf{c\hat{w}}[3], \hat{\mathbf{u}} = [\,], p)$
      Return $p[3]$
    $b \leftarrow \mathrm{L}(i = 1, N = 1, \mathbf{c\hat{w}}[4], \hat{\mathbf{u}} = [\,], p)$
      Return $p[4]$
    Return $\frac{p[3]p[4]+1}{p[3]+p[4]}$
  Return $\frac{ab+1}{a+b}$
$i \leftarrow 2$
$r \leftarrow \mathrm{L}(i = 2, N = 4, \mathbf{c\hat{w}}, \hat{\mathbf{u}} = [\hat{u}[1]], p)$
  $a \leftarrow \mathrm{L}(i = 1, N = 2, \mathbf{c\hat{w}}[1:3], \hat{\mathbf{u}} = [\hat{u}[1]], p)$
    $a \leftarrow \mathrm{L}(i = 1, N = 1, \mathbf{c\hat{w}}[1], \hat{\mathbf{u}} = [\hat{u}[1]], p)$
      Return $p[1]$
    $b \leftarrow \mathrm{L}(i = 1, N = 1, \mathbf{c\hat{w}}[2], \hat{\mathbf{u}} = [\,], p)$
      Return $p[2]$
    Return $\frac{p[1]p[2]+1}{p[1]+p[2]}$
  $b \leftarrow \mathrm{L}(i = 1, N = 2, \mathbf{c\hat{w}}[3:], \hat{\mathbf{u}} = [\,], p)$
    $a \leftarrow \mathrm{L}(i = 1, N = 1, \mathbf{c\hat{w}}[3], \hat{\mathbf{u}} = [\,], p)$
      Return $p[3]$
    $b \leftarrow \mathrm{L}(i = 1, N = 1, \mathbf{c\hat{w}}[4], \hat{\mathbf{u}} = [\,], p)$
      Return $p[4]$
    Return $\frac{p[3]p[4]+1}{p[3]+p[4]}$
  Return $(a)^{1-2\hat{u}[1]}\, b$

$i \leftarrow 3$

$r \leftarrow \mathrm{L}(i = 3, N = 4, \mathbf{c\hat{w}}, \mathbf{\hat{u}} = [\hat{u}[1], \hat{u}[2]], p)$

  $a \leftarrow \mathrm{L}(i = 2, N = 2, \mathbf{c\hat{w}}[1:3], \mathbf{\hat{u}} = [\hat{u}[1] \oplus \hat{u}[2]], p)$

    $a \leftarrow \mathrm{L}(i = 1, N = 1, \mathbf{c\hat{w}}[1], \mathbf{\hat{u}} = [\hat{u}[1] \oplus \hat{u}[2]], p)$

      Return $p[1]$

    $b \leftarrow \mathrm{L}(i = 1, N = 1, \mathbf{c\hat{w}}[2], \mathbf{\hat{u}} = [\,], p)$

      Return $p[2]$

    Return $(p[1])^{1-2(\hat{u}[1]\oplus\hat{u}[2])}\, p[2]$

  $b \leftarrow \mathrm{L}(i = 2, N = 2, \mathbf{c\hat{w}}[3:], \mathbf{\hat{u}} = [\hat{u}[2]], p)$

    $a \leftarrow \mathrm{L}(i = 1, N = 1, \mathbf{c\hat{w}}[3], \mathbf{\hat{u}} = [\hat{u}[2]], p)$

      Return $p[3]$

    $b \leftarrow \mathrm{L}(i = 1, N = 1, \mathbf{c\hat{w}}[4], \mathbf{\hat{u}} = [\,], p)$

      Return $p[4]$

    Return $(p[3])^{1-2\hat{u}[2]}\, p[4]$

  Return $\frac{ab+1}{a+b}$

$i \leftarrow 4$

$r \leftarrow \mathrm{L}(i = 4, N = 4, \mathbf{c\hat{w}}, \mathbf{\hat{u}} = [\hat{u}[1], \hat{u}[2], \hat{u}[3]], p)$

  $a \leftarrow \mathrm{L}(i = 2, N = 2, \mathbf{c\hat{w}}[1:3], \mathbf{\hat{u}} = [\hat{u}[1] \oplus \hat{u}[2], \hat{u}[3]], p)$

    $a \leftarrow \mathrm{L}(i = 1, N = 1, \mathbf{c\hat{w}}[1], \mathbf{\hat{u}} = [\hat{u}[1] \oplus \hat{u}[2]], p)$

      Return $p[1]$

    $b \leftarrow \mathrm{L}(i = 1, N = 1, \mathbf{c\hat{w}}[2], \mathbf{\hat{u}} = [\hat{u}[2]], p)$

      Return $p[2]$

    Return $(p[1])^{1-2(\hat{u}[1]\oplus\hat{u}[2])}\, p[2]$

  $b \leftarrow \mathrm{L}(i = 2, N = 2, \mathbf{c\hat{w}}[3:], \mathbf{\hat{u}} = [\hat{u}[3]], p)$

    $a \leftarrow \mathrm{L}(i = 1, N = 1, \mathbf{c\hat{w}}[3], \mathbf{\hat{u}} = [\hat{u}[3]], p)$

      Return $p[3]$

    $b \leftarrow \mathrm{L}(i = 1, N = 1, \mathbf{c\hat{w}}[4], \mathbf{\hat{u}} = [\,], p)$

      Return $p[4]$

    Return $(p[3])^{1-2u[2]}\, p[4]$

  Return $(a)^{1-2u[3]}\, b$

6.5 Implementation and evaluation

We now consider the security of the entire decoding algorithm. For each iteration of the for loop in Lines 3, 7, 9 and 10 of Algorithm 6.3 we create one Boolean circuit and one arithmetic circuit. As each copy of the Boolean circuit uses the same inputs and the first gates of each circuit is only t-NI, we further adapt the Boolean circuit to apply a mask refresh to the input before it is used by the XOR gates. We also make use of a t-SNI SNIMagnitudeComparator gadget. We give the full SNIDecoding algorithm as a sequence of gadget for a small example in Fig. 6.5. For this example we've set $N = 4$ and run SNIL on every bit, rather than just the non-frozen bits.

Theorem 6.5.1 (t-SNI of SNIDecoding)**.** *Let* $\hat{\mathbf{cw}}_0^{(.)}$ *and* $\hat{\mathbf{p}}_0^{(.)}$ *be the input and* $\hat{\mathbf{u}}_0^{(.)}$ *be the output to* MaskedDecoding*. For any set of* t_{MD} *intermediate variables and any subset* O *of output variables such that* $t_{\mathrm{MD}} + |O| \leq t$*, there exists a subset* I *of input variables such that* $|I| \leq t_{\mathrm{MD}}$ *and the* t_{MD} *intermediate variables and the output variables can be perfectly simulated using the* I *input variables.*

Proof. We consider each iteration of the for loop to be run in series, as each gadget is t-SNI (Lemmas 6.4.1 and 6.4.4) and every input is used only once unless it's used by a t-SNI gadget, then by Proposition 1 SNIDecoding is t-NI. Moreover as the last gadget call before each bit it outputted is t-SNI, SNIDecoding is t-SNI. □

For the experiments we have take the first order masked version of the implementation. For each component of the decoding process and for the entire decoding process we collect 3000 traces for running the component on the all 0 codeword and 3000 traces for running the component on the all 0 codeword with 1 error. Using these we perform Welch's t-test and, setting the threshold to ± 5.730 [DZD+18], we display the results below. From this we can see that our decoding algorithm is secure against power analysis attacks. All traces were collected using ELMO [MOW17], with the ELMO energy model flag set rather than the Hamming Weight model flag.

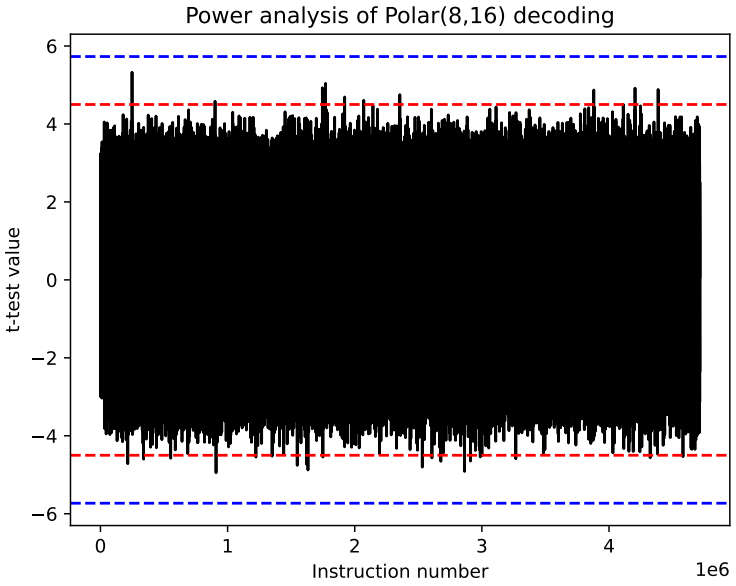

Figure 6.6: TVLA results for Polar(8,16) code between BCH code decoding for the all 0 codeword, and the all 0 codeword with 1 error for our masked implementation. The inner dashed line is at ±4.5 and the outer dashed line is at ±5.730.

We give a brief overview of the time and randomness complexity of the smaller gadgets in Table 6.1.

Gadget	$\mathcal{T}$	$\mathcal{R}$
SNIMul/SNIAnd	$3(tg)^2$	$\frac{(tg)^2}{2}$
NIMul/NIAnd	$\frac{7(tg)^2}{4}$	$\frac{(tg)^2}{4} + (tg)$
SNIRef	$(tg)^2$	$\frac{(tg)^2}{2}$
NIRef	$2t$	t
SNIAdd/SNIXOR	$(tg)^2 + t$	$\frac{(tg)^2}{2}$
NIAdd/NIXOR	t	–
NINot	1	–
G_8	$44t^2 + 6t$	$16t^2 + 15t$
SNIOdd	$12(tg)^2 + 2t$	$2(tg)^2$
NIOdd	$\frac{19(tg)^2}{2} + 2t$	$\frac{3(tg)^2}{2} + 2tg$
SNIEven	$\frac{31(tg)^2}{2} + 2t$	$\frac{5(tg)^2}{2} + 2(tg)$
NIEven	$12(tg)^2 + 2t$	$2(tg)^2 + 4tg$

Table 6.1: The time and randomness complexity of small gadgets.

Complexity. We estimate the run time complexity of SNIDecoding, $\mathcal{T}_{G_{24}}$, as

$$\begin{aligned}\mathcal{T}_{G_{24}} &= \frac{N(N-1)}{8}(\mathcal{T}_{\text{SNIOdd}} + \mathcal{T}_{\text{NIOdd}} + 2\mathcal{T}_{\text{SNIEven}} + \mathcal{T}_{\text{SNIXOR}}) + \frac{N}{2}\mathcal{T}_{G_8} \\ &= O(N^2((tg)^2 + (tg)^2 + (tg)^2 + (tg)^2) + N(t^2)) \\ &= O((Ntg)^2)\end{aligned}$$

We estimate the randomness complexity of SNIDecoding, $\mathcal{R}_{G_{24}}$, as

$$\begin{aligned}\mathcal{R}_{G_{24}} &= \frac{N(N-1)}{8}(\mathcal{R}_{\text{SNIOdd}} + \mathcal{R}_{\text{NIOdd}} + 2\mathcal{R}_{\text{SNIEven}} + \mathcal{R}_{\text{SNIXOR}}) + \frac{N}{2}\mathcal{T}_{G_8} \\ &= O(N^2((tg)^2 + (tg)^2 + (tg)^2) + N(t^2)) \\ &= O((Ntg)^2)\end{aligned}$$

Both the run time and random complexity can be reduced by using memoization techniques, such as those suggested for the (unmasked) decoding algorithm in [Ari09].

CHAPTER 7

Cost of using error correcting codes

Having discussed how error correcting codes can be used to reduce the size of ciphertexts and how to secure various error correcting codes against side-channel attacks, we now examine thoroughly the cost of using these secure error correcting codes. We start by discussing in more detail how the side-channel resistant BCH code can be used in the ways mentioned in Chapter 3, and give a more detailed analysis on the effectiveness of the masked algorithm. We then move on to polar codes, providing an analysis on methods of puncturing polar codes. In this section we also discuss different methods for modeling cryptosystems as noisy channels, and the impact this has on the effectiveness of polar codes. Finally we give an analysis of the cost, with respect to run time, of applying the masked versions of the BCH codes from Chapter 5 and polar codes from Chapter 6 in the manner suggested in Chapter 3.

7.1 Modelling KEMs as noisy channels

Polar codes take the noisy channel being used into consideration when choosing which bits should be frozen before encoding, additionally the decoder uses the noise

model to help calculate the likelihood ratios. We aim to model the encryption and decryption of the message as a noisy channel, where the message being encrypted is the input to the channel, the channel consists of applying the encryption and the decryption operations, and the resultant message is the output to the channel. We will consider the channel to be memoryless, since whilst it has been shown that the decryption failure rates of individual bits in a LWE-based cryptosystem are not independent, the resultant error is small [MFS20]. We start by considering the cryptosystem as a Binary Symmetric Channel (BSC), since this is the simplest model of a noisy channel. This model is also a good approximation of using a KEM with hard decoding. We will then consider the cryptosystem to be an Additive White Gaussian Noise (AWGN) channel, which acts as a better approximation when soft decoding is being used. Whilst there has been previous work on modelling the R-LWE cryptosystem as a noisy channel [WL21, MPWZ23], these models have not been extended to the general LWE cryptosystems, and are near enough to the BSC and AWGN channel that we will only consider the idealised models.

7.1.1 BSC

The Binary Symmetric Channel considers each bit being sent over the channel individually. The channel flips the bit with some crossover probability p, and the value of the bit is preserved with probability $(1-p)$. The channel capacity, i.e. the best rate that a code can achieve over the channel, is $1 - H(p)$. There are linear codes that do achieve the channel capacity [For65].

For the case of modelling a LWE-based cryptosystem as a BSC we set $p = \delta_{\text{bit}}$, the decryption failure rate for an arbitrary bit. Since we consider the cryptosytem to round the value of the final decoded coefficient to the nearest bit, this acts as a reasonable model, however this is not entirely accurate. In order to create a more accurate model for soft decoding algorithms, such as polar codes, we also consider the Additive White Gaussian Noise channel, however this increases the complexity of the model.

7.1.2 AWGN

The Additive White Gaussian Noise channel is an analogue channel where we consider the values of the input and output to be continuous and the noise added to the channel to be drawn from a Gaussian distribution. This accurately matches the general LWE cryptosystem, where the noise added is often considered to be Gaussian in nature, and the outputted message is continuous in nature. For modeling LWE based cryptosystems as AWGN channels, we apply the central limit theorem to approximate the noise that is added as a Gaussian centered around 0. This gives signal to noise ratio (SNR) of the AWGN as $\frac{1}{\sigma^2}$, where σ^2 is the variance of the Gaussian distribution.

7.2 Using BCH codes

In Chapter 3 we suggested using BCH codes because they use a small number of bits per error that it corrects. However one disadvantage of BCH codes is that the codeword is always of length 2^l, and so has a higher redundancy for some message size - specifically for when we want to encode a small number of bits and only want to correct a small number of errors. BCH codes can be made more space efficient by puncturing them. Puncturing, in the context of error correcting codes, is where some bits of the codeword are removed to give a code with lower redundancy. Typically this is performed in such a way that the decoder for the unpunctured codeword can be used for the punctured codeword. BCH codes are typically implemented using the systematic encoding, where the original message appears at the start of the codeword. When the systematic encoding is used for BCH code and the message we wish to encode is smaller than required, then there is a trivial puncturing method of omitting the 0 bits that appear before the message rather than padding the message. In order to minimise the amount of redundancy in the codewords, we will always apply this puncturing method.

A further method of reducing redundancy is to split the message into two chunks and encode them both separately. However whilst this can be done in such a way that the same number of errors can be corrected, it can make the message more

vulnerable to burst errors. Burst errors are not typical of normal decryption failures for KEMs, however could occur during failure boosting. Often only a small space gain can be made by applying this technique, but it can lead to faster decoding, and so we consider this technique later.

We plot the runtime of different BCH codes that use the first order masking scheme presented in Chapter 5 against the error correcting capabilities of the code in Fig. 7.1.

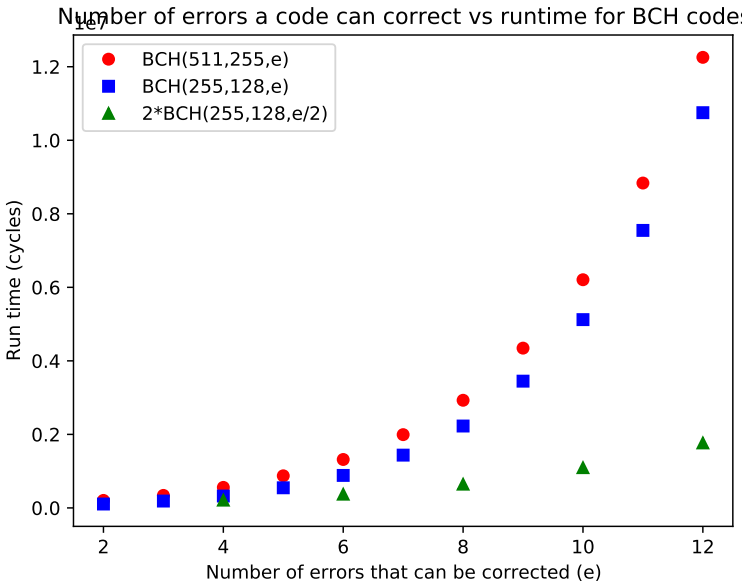

Figure 7.1: Runtime of different masked BCH codes against the number of errors they can correct.

7.3 Using polar codes

7.3.1 Reliability

In order to calculate the probability of successfully decoding a polar code for the AWGN channel, we apply the techniques from [TR17, WLS14], which we briefly outline. First we have to calculate the probability that individual bits are decoded correctly. We denote the probability that the ith bit is decoded incorrectly after $\log(N)$ rounds as $P(b_N^{(i)})$. This probability is calculated recursively using the follow-

ing relation:

$$P(b_N^i) = \begin{cases} 2P(b_N^{i-1}/2)(1 - P(b_N^{i-1}/2)), & i \text{ is even} \\ Q(\sqrt{2}Q^{-1}(P(b_N^{i-1}))), & i \text{ is odd} \end{cases}$$

where $Q(x)$ is the $Q-$function, $Q(x) := \frac{1}{\sqrt{2\pi}} \int_x^\infty e^{-\frac{u^2}{2}} du$ and $Q^{-1}(x)$ is its inverse. The K bits with the lowest error rate are chosen to be data bits and the remaining $N - K$ bits are frozen. The probability that the block is decoded correctly, $P(b_N)$, is calculated as

$$P(b_N) = \prod_{i \in \mathcal{D}} (1 - P(b_N^i)).$$

Where $\mathcal{D}$ represents the set of indices of data bits within the codeword.

7.3.2 Puncturing polar codes

Similarly to BCH codes, there is a trivial puncturing scheme for polar codes, however it does come at the cost of reliability. We briefly outline the algorithm for puncturing p bits out of N, where $p < \frac{N}{2}$. Before selecting which bits are frozen bits or data bits, the last p bits are frozen to 0 and only the remaining bits are allocated to be data bits or frozen bits. Since the last bits of the codeword typically have the lowest bit error rate, this increases the block error rate.

Other methods for puncturing polar codes exist, such as [WL14], however this is at an added runtime complexity for decoding the code. We therefore only consider the trivial method for puncturing polar codes.

Rather than puncturing the codes, we can also split the message into smaller blocks, and apply a smaller polar code to each block. As with BCH codes, this doesn't reduce the size of the codes, however it does reduce the run time of the decoding algorithm.

7.4 Cost of using error correcting codes

We now apply the methodology from Chapter 3, but focus on the time it takes to decrypt the ciphertext rather that the decryption failure rate. We give plots below,

where for each targeted failure rate, we show the trade off between the size of the ciphertext and the decryption time. In order to accurately compare the cost of using masked error correcting codes, we use the timing information from [BGR+21] for masked Kyber-768, however as their code is not available we have assumed that the $2.2x$ overheard for masking Kyber-768 also extends to Kyber-512 and Kyber-1024. The tradeoff is presented in Fig. 7.2 giving the total run time in cycles and is normalised using the initial Kyber ciphertext size and the runtime of the masked version of Kyber in Fig. 7.3. In order to produce the graph we have made the assumption that the only change to the decryption algorithms runtime is that the received message needs to be decoded, ignoring extra time spent on converting the coefficients into bits where multiple bits are encoded in a single coefficient. From the plots we can see that a large reduction in the size of the ciphertext can be achieved by using fairly efficient error correcting codes, however more drastic reductions in the size of the ciphertext comes at a very high cost in terms of the run time.

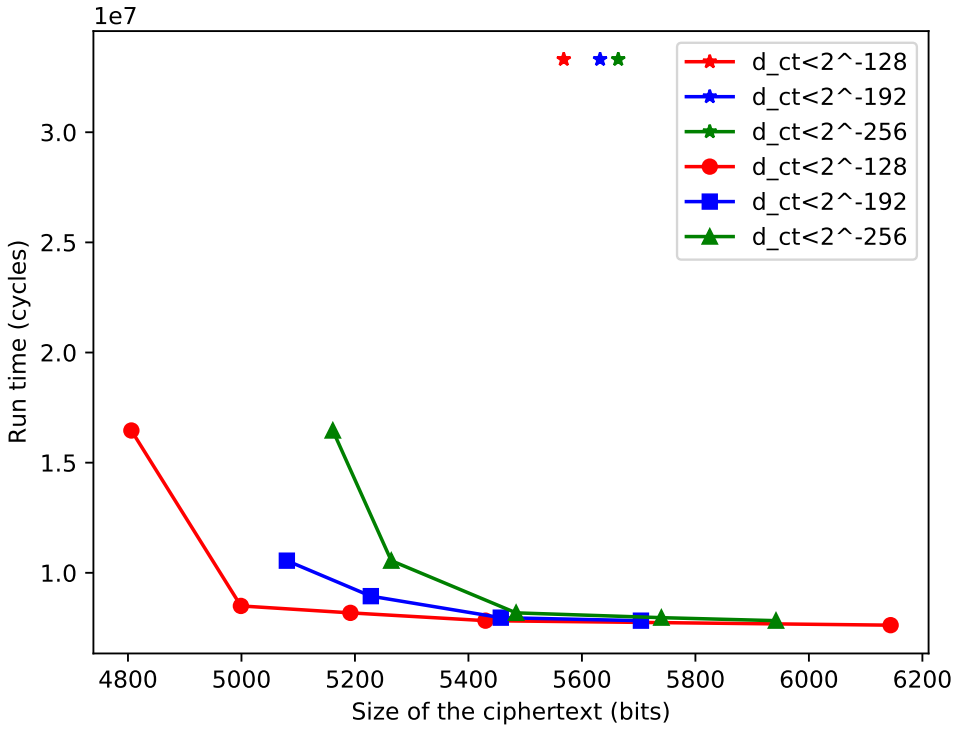

Figure 7.2: The time cost of reducing the size of ciphertexts for Kyber-512.

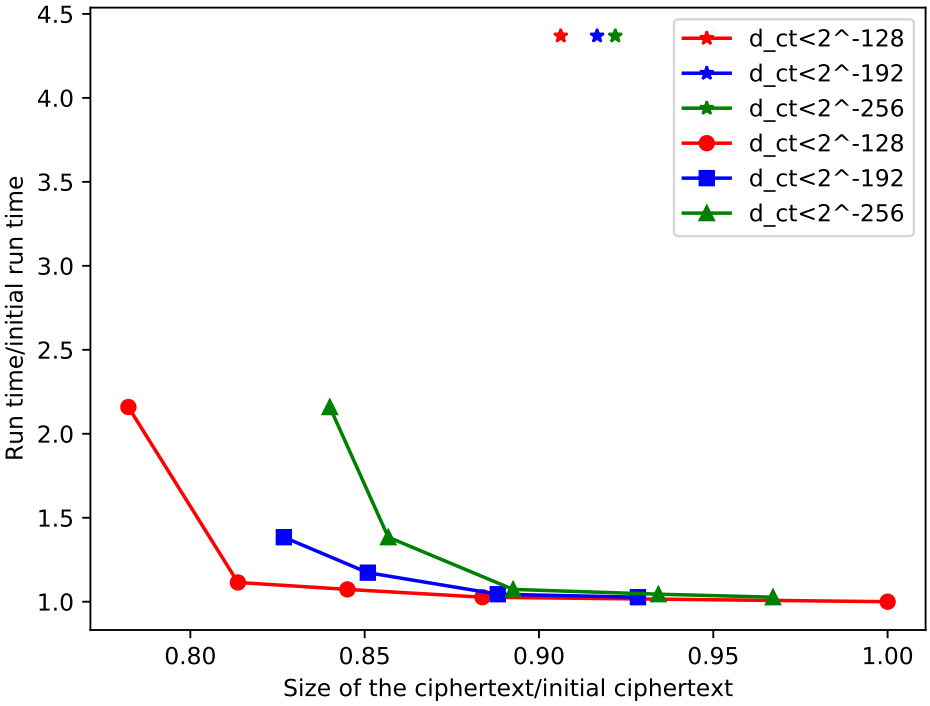

Figure 7.3: The time cost of reducing the size of ciphertexts, normalised by the initial Kyber-512 parameters.

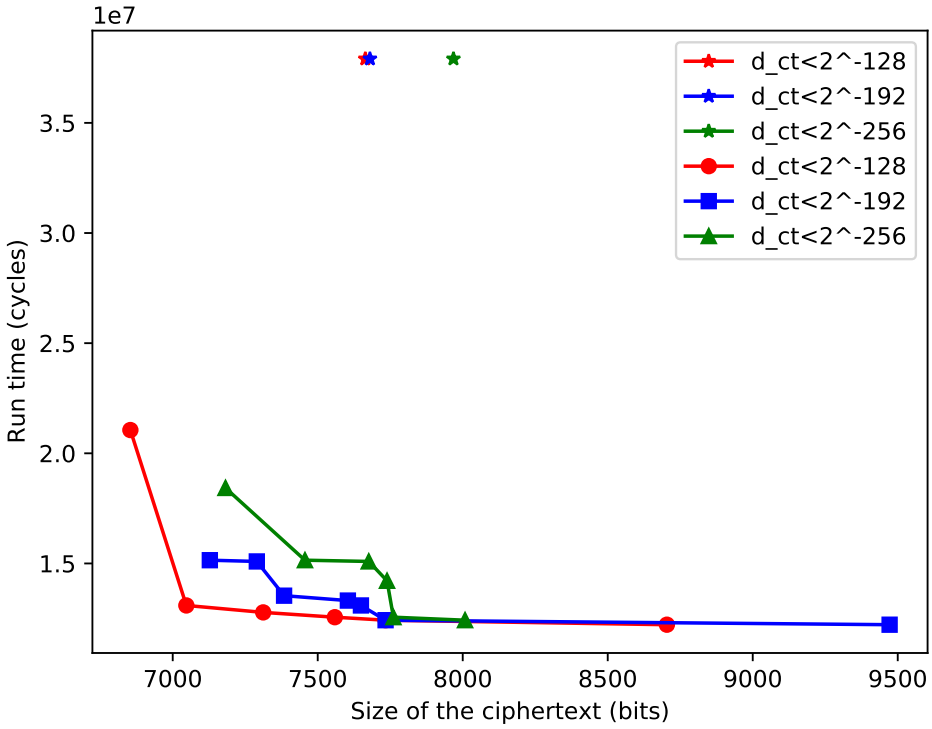

Figure 7.4: The time cost of reducing the size of ciphertexts for Kyber-768.

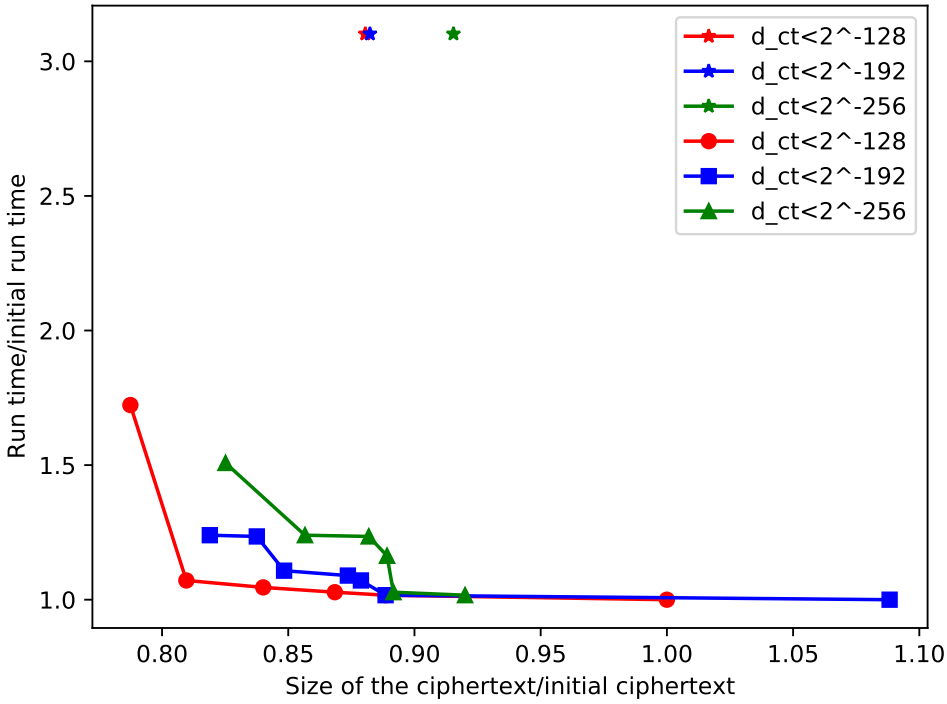

Figure 7.5: The time cost of reducing the size of ciphertexts, normalised by the initial Kyber-768 parameters.

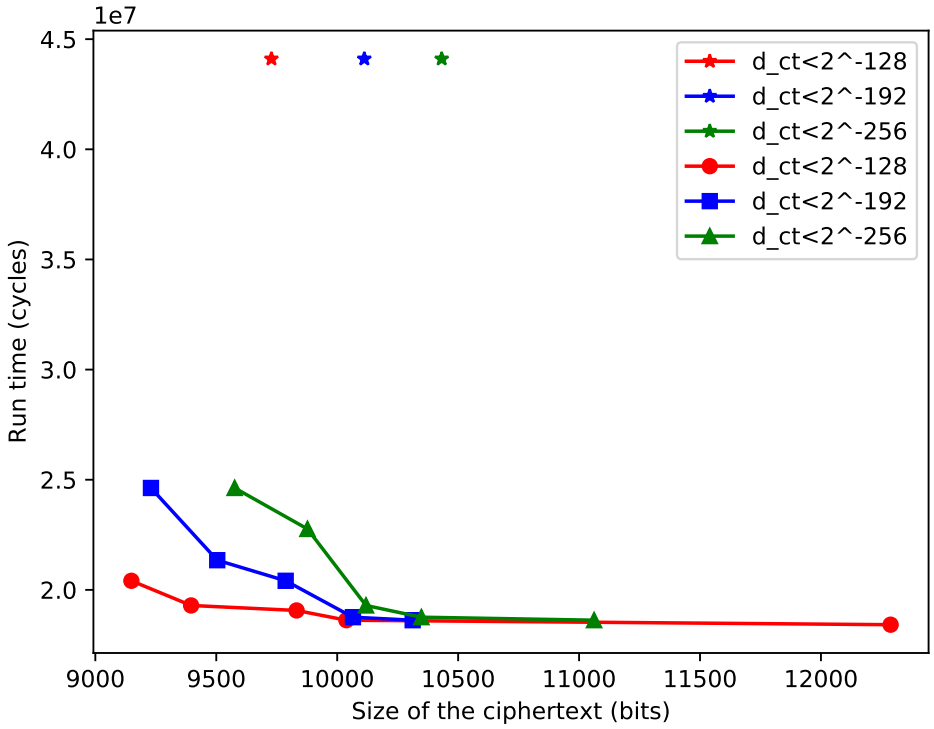

Figure 7.6: The time cost of reducing the size of ciphertexts for Kyber-1024.

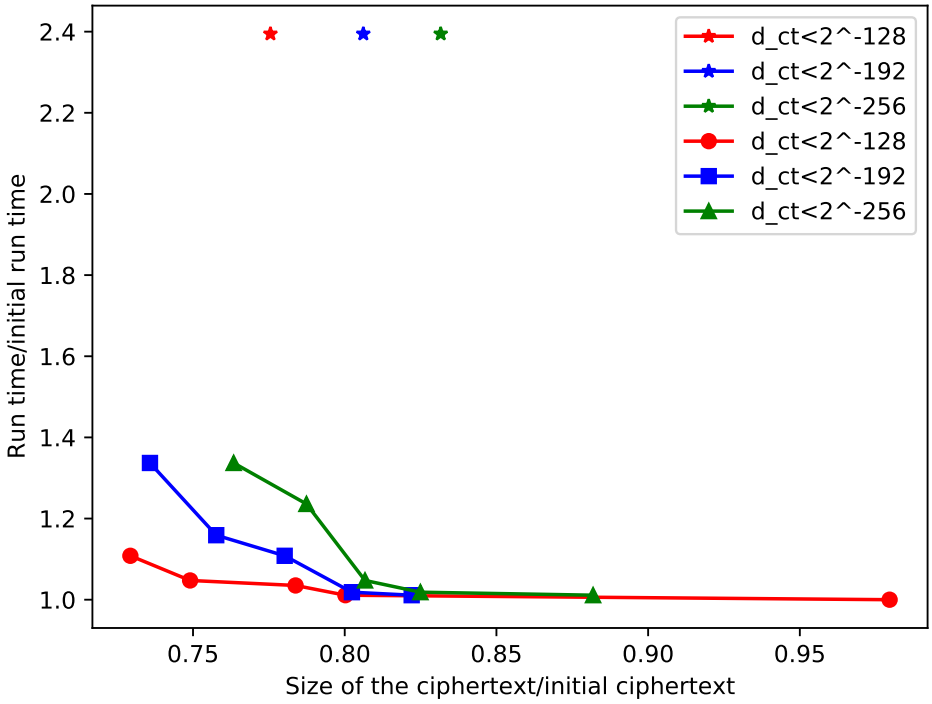

Figure 7.7: The time cost of reducing the size of ciphertexts, normalised by the initial Kyber-1024 parameters.

From these charts we propose the following parameter sets for Kyber, which give varying tradeoffs between a reduction in ciphertext size and the increased cost of decapsulation. We give parameter sets for different decryption failure rates, with reasonable optimisations being possible even for the most conservative of decryption failure rates. Whilst at the moment we do not propose using polar codes for Kyber, we believe that with further optimisations it will become more applicable. We note that schemes that require even lower failure rates might find that polar codes become more advantageous, especially for smaller message lengths.

n	k	p	q	t	b	Code	|pt| (Bytes)	|ct| (Bytes)	δ_{ct}	runtime (1000 cycles)
256	2	3329	2^8	2^2	1	bch	256	4824	2^{-144}	19879
256	2	3329	2^8	2^3	1	bch	256	5026	2^{-159}	8940
256	2	3329	2^8	2^3	1	dblbch	256	5134	2^{-139}	8718
256	2	3329	2^9	2^2	1	bch	256	5192	2^{-143}	8182
256	2	3329	2^9	2^3	1	bch	256	5430	2^{-181}	7827
256	2	3329	2^{11}	2^4	1	bch	256	6656	2^{-163}	7623
256	2	3329	2^{11}	2^4	1	dblbch	256	6656	2^{-163}	7623
256	2	3329	2^8	2^3	1	bch	256	5080	2^{-206}	10551
256	2	3329	2^9	2^2	1	bch	256	5228	2^{-203}	8940
256	2	3329	2^9	2^3	1	bch	256	5457	2^{-242}	7964
256	2	3329	2^9	2^4	1	bch	256	5704	2^{-240}	7827
256	2	3329	2^8	2^3	1	bch	256	5161	2^{-278}	16459
256	2	3329	2^9	2^2	1	bch	256	5264	2^{-263}	10551
256	2	3329	2^9	2^3	1	bch	256	5484	2^{-304}	8182
256	2	3329	2^9	2^4	1	bch	256	5740	2^{-322}	7964
256	2	3329	2^{10}	2^3	1	bch	256	5942	2^{-316}	7827

Table 7.1: Some fine-tuned Kyber-512 parameter sets, for 256 bit plaintexts with the cost of a fully masked implementation.

n	k	p	q	t	b	Code	|pt| (Bytes)	|ct| (Bytes)	δ_{ct}	runtime (1000 cycles)
256	3	3329	2^8	2^3	1	bch	256	7101	2^{-189}	9616
256	3	3329	2^8	2^3	1	dblbch	256	7236	2^{-169}	9391
256	3	3329	2^8	2^4	1	bch	256	7348	2^{-186}	8497
256	3	3329	2^8	2^5	1	bch	256	7604	2^{-172}	8182
256	3	3329	2^9	2^3	1	bch	256	7734	2^{-198}	7827
256	3	3329	2^{11}	2^4	1	bch	256	9472	2^{-201}	7623
256	3	3329	2^{11}	2^4	1	dblbch	256	9472	2^{-201}	7623
256	3	3329	2^8	2^3	1	bch	256	7128	2^{-214}	10551
256	3	3329	2^8	2^3	1	dblbch	256	7290	2^{-195}	10495
256	3	3329	2^8	2^4	1	bch	256	7384	2^{-217}	8940
256	3	3329	2^9	2^2	1	dblbch	256	7604	2^{-193}	8718
256	3	3329	2^8	2^5	1	bch	256	7649	2^{-208}	8497
256	3	3329	2^9	2^3	1	bch	256	7734	2^{-198}	7827
256	3	3329	2^{11}	2^4	1	bch	256	9472	2^{-201}	7623
256	3	3329	2^{11}	2^4	1	dblbch	256	9472	2^{-201}	7623
256	3	3329	2^8	2^3	1	bch	256	7182	2^{-263}	13830
256	3	3329	2^8	2^4	1	bch	256	7456	2^{-282}	10551
256	3	3329	2^9	2^2	1	dblbch	256	7676	2^{-259}	10495
256	3	3329	2^8	2^5	1	bch	256	7739	2^{-279}	9616
256	3	3329	2^9	2^3	1	bch	256	7761	2^{-266}	7964
256	3	3329	2^9	2^4	1	bch	256	8008	2^{-265}	7827

Table 7.2: Some fine-tuned Kyber-768 parameter sets, for 256 bit plaintexts with the cost of a fully masked implementation.

n	k	p	q	t	b	Code	|pt| (Bytes)	|ct| (Bytes)	δ_{ct}	runtime (1000 cycles)
256	4	3329	2^8	2^3	1	bch	256	9203	2^{-176}	11969
256	4	3329	2^8	2^4	1	bch	256	9468	2^{-184}	9616
256	4	3329	2^8	2^5	1	bch	256	9742	2^{-178}	8940
256	4	3329	2^9	2^3	1	bch	256	10065	2^{-201}	7964
256	4	3329	2^9	2^4	1	bch	256	10312	2^{-200}	7827
256	4	3329	2^{12}	2^4	1	bch	256	13312	2^{-183}	7623
256	4	3329	2^{12}	2^4	1	dblbch	256	13312	2^{-183}	7623
256	4	3329	2^8	2^3	1	bch	256	9230	2^{-195}	13830
256	4	3329	2^8	2^4	1	bch	256	9504	2^{-208}	10551
256	4	3329	2^8	2^5	1	bch	256	9787	2^{-205}	9616
256	4	3329	2^9	2^3	1	bch	256	10065	2^{-201}	7964
256	4	3329	2^9	2^4	1	bch	256	10312	2^{-200}	7827
256	4	3329	2^8	2^4	1	bch	256	9576	2^{-256}	13830
256	4	3329	2^8	2^5	1	bch	256	9877	2^{-258}	11969
256	4	3329	2^9	2^3	1	bch	256	10119	2^{-304}	8497
256	4	3329	2^9	2^4	1	bch	256	10348	2^{-267}	7964
256	4	3329	2^{10}	2^3	1	bch	256	11062	2^{-285}	7827

Table 7.3: Some fine-tuned Kyber-1024 parameter sets, for 256 bit plaintexts with the cost of a fully masked implementation.

CHAPTER 8

Conclusion

We conclude the thesis by giving a brief overview of each chapter and finally discuss potential future directions for building on this work.

8.1 Contribution

Error correction for post quantum cryptography. In this chapter we presented an analysis of how different techniques for reducing the size of ciphertexts apply to specific KEMs of interest, including highlighting new parameter sets that minimise the size of the ciphertexts produced. One key issue that was presented was that whilst error correcting codes could dramatically reduce the size of ciphertexts, no side-channel resistant implementations existed.

Secure linear algebra. Here we provided masked implementations of the LUP decomposition algorithm, a masked algorithm for calculating the determinant of a matrix, and a masked algorithm for solving a system of linear equations. As part of this we also created a masked version of the magnitude comparator algorithm. For all of these algorithms we provided proofs in the probing model to show that

they are secure against side-channel attacks. These algorithms form the backbone of error correcting code decoding algorithms, and so are essential for providing masked implementations of the decoding algorithms.

Secure BCH codes. In this chapter we showed how the masked linear algebra algorithms could be used for masking the decoding algorithm for BCH codes. As part of this we masked the syndrome calculation algorithm and the Chien search algorithm, which are used by most decoding algorithms. For all algorithms we presented proofs in the probing model to show security and showed experimental results for security as well.

Masked polar codes. Here we masked the polar code decoding algorithm. As part of this we created new gadgets for masking operations over $\mathbb{Q}_1$, rather than $\mathbb{Z}_q$. For all algorithms we presented proofs in the probing model to show security and showed experimental results for security as well.

The cost of secure error correction for post quantum cryptography. In the final chapter we gave an analysis of how secure error correcting codes would reduce the size of ciphertexts and what the cost of this would be in terms of run time.

8.2 Future Directions

Further benchmarking and model checking. In order to increase confidence in the masked algorithms that we have proposed, we suggest two further improvements. The first would be to run tests and provide benchmarks for the algorithms using other hardware, e.g. the Cortex-M4. This would verify that the masking scheme is secure for other microcontrollers and would provide more information on the efficiency of the masking under different memory models. The second improvement would be to use model checking, such as EasyCrypt, to add further verification of the security of the smaller gadgets.

Optimised BCH masking. We propose three methods for improving the masked schemes. Firstly rewriting the linear algebra algorithms so that they are vectorised. This would make it possible to speed up the gadgets by making use of SIMD instruction sets on e.g. the ARM Cortex-M4. This could also be achieved by rewriting the gadgets so that they could use optimised libraries such as BLAS. The second improvement would be in mask the Berlekamp-Massey algorithm rather than the Peterson algorithm, giving - hopefully - an asymptotic speedup. However, as discussed in Chapter 5, there are several challenges with masking this. The final future direction to improve the masked algorithms is to look at methods to reduce the number of calls to SecRef. Ideally this would be done generally enough that it can provide a minimum amount of randomness regardless of the value of t rather than only being optimal for small values of t.

Optimised polar code masking. We propose two methods for improving the masked polar code schemes. The first is to apply the memoisation techniques mentioned in Chapter 6 to reduce the time complexity, however due to the reuse of shares this may lead to an increased randomness complexity. The second method is to convert the decoding algorithm from a recursive algorithm to an iterative algorithm, as iterative algorithms often perform better on microcontrollers.

Masking code based KEMs. Finally the main future direction would be to use the tools that we've built to mask the BCH code decoding algorithm and apply them to give masked versions of the code based KEMs. Despite code based KEMs being considered for standardisation, there has been very little work on side channel resistant implementations of any of the KEMs. Two of the code based KEMs that are currently being considered for standardisation make use of BCH codes or similar linear codes, and so our masked algorithms could be used directly.

Bibliography

[AAB+22] Carlos Aguilar Melchor, Nicolas Aragon, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Edoardo Persichetti, Gilles Zémor, Jurjen Bos, Arnaud Dion, Jerome Lacan, Jean-Marc Robert, and Pascal Veron. HQC. Technical report, National Institute of Standards and Technology, 2022. available at `https://csrc.nist.gov/Projects/post-quantum-cryptography/round-4-submissions`. (document), 3.5, 4

[AAC+22] Gorjan Alagic, Daniel Apon, David Cooper, Quynh Dang, Thinh Dang, John Kelsey, Jacob Lichtinger, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, Daniel Smith-Tone, and Yi-Kai Liu. Status Report on the Third Round of the NIST Post-Quantum Cryptography Standardization Process. Technical report, 2022. 3

[AASA+19] Gorjan Alagic, Jacob Alperin-Sheriff, Daniel Apon, David Cooper, Quynh Dang, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, and Daniel Smith-Tone. Status Report on the First Round of the NIST Post-Quantum Cryptography Standardization Process. NISTIR, 8240, 2019. 3

[AASA+20] Gorjan Alagic, Jacob Alperin-Sheriff, Daniel Apon, David Cooper, Quynh Dang, John Kelsey, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, and Daniel Smith-Tone. Status Report on the Second Round of the NIST Post-Quantum Cryptography Standardization Process. NISTIR, 8309, 2020. `https://csrc.nist.gov/publications/detail/nistir/8309/final`. 3

[ABB+22] Nicolas Aragon, Paulo Barreto, Slim Bettaieb, Loic Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Phillipe Gaborit, Shay Gueron, Tim Guneysu, Carlos Aguilar Melchor, Rafael Misoczki, Edoardo Persichetti, Nicolas Sendrier, Jean-Pierre Tillich, Gilles Zémor, Valentin Vasseur, Santosh Ghosh, and Jan Richter-Brokmann.

BIKE. Technical report, National Institute of Standards and Technology, 2022. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/round-4-submissions. (document), 3.5

[ABC+22] Martin R. Albrecht, Daniel J. Bernstein, Tung Chou, Carlos Cid, Jan Gilcher, Tanja Lange, Varun Maram, Ingo von Maurich, Rafael Misoczki, Ruben Niederhagen, Kenneth G. Paterson, Edoardo Persichetti, Christiane Peters, Peter Schwabe, Nicolas Sendrier, Jakub Szefer, Cen Jung Tjhai, Martin Tomlinson, and Wen Wang. Classic McEliece. Technical report, National Institute of Standards and Technology, 2022. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-4-submissions. (document), 3.5

[ACD+18] Martin R. Albrecht, Benjamin R. Curtis, Amit Deo, Alex Davidson, Rachel Player, Eamonn W. Postlethwaite, Fernando Virdia, and Thomas Wunderer. Estimate all the LWE, NTRU schemes! Cryptology ePrint Archive, Report 2018/331, 2018. https://eprint.iacr.org/2018/331. 3.2.3, 3.4.1

[APS15] Martin R Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. Journal of Mathematical Cryptology, 9(3):169–203, 2015. 3.2.3

[Ari09] Erdal Arikan. Channel polarization: A method for constructing capacity-achieving codes for symmetric binary-input memoryless channels. IEEE Trans. Inf. Theor., 55(7):3051–3073, Jul 2009. 2.5.2, 7, 6.5

[Bar68] Erwin H. Bareiss. Sylvester's identity and multistep integer-preserving gaussian elimination. Mathematics of Computation, 22(103):565–578, 1968. 5.1

[BBD+16] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, Pierre-Yves Strub, and Rébecca Zucchini. Strong non-interference and type-directed higher-order masking. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, ACM CCS 2016: 23rd Conference on Computer and Communications Security, pages 116–129, Vienna, Austria, October 24–28, 2016. ACM Press. 2.4, 1, 2, 1, 2.4, 4, 5, 6

[BBP+16] Sonia Belaïd, Fabrice Benhamouda, Alain Passelègue, Emmanuel Prouff, Adrian Thillard, and Damien Vergnaud. Randomness complexity of private circuits for multiplication. In Marc Fischlin and Jean-Sébastien Coron, editors, Advances in Cryptology – EUROCRYPT 2016, Part II, volume 9666 of Lecture Notes in Computer Science, pages 616–648, Vienna, Austria, May 8–12, 2016. Springer, Heidelberg, Germany. 2.4, 11, 4, 5, 6

[BDK+20] Michiel Van Beirendonck, Jan-Pieter D'Anvers, Angshuman Karmakar, Josep Balasch, and Ingrid Verbauwhede. A side-channel resistant implementation of SABER. Cryptology ePrint Archive, Report 2020/733, 2020. https://eprint.iacr.org/2020/733. 5

[BGPT19] Sauvik Bhattacharya, Oscar Garcia-Morchon, Rachel Player, and Ludo Tolhuizen. Achieving secure and efficient lattice-based public-key encryption: the impact of the secret-key distribution. Cryptology ePrint Archive, Report 2019/389, 2019. https://eprint.iacr.org/2019/389. 3.1

[BGR+21] Joppe W. Bos, Marc Gourjon, Joost Renes, Tobias Schneider, and Christine van Vredendaal. Masking kyber: First- and higher-order implementations. IACR Transactions on Cryptographic Hardware and Embedded Systems, 2021(4):173–214, 2021. https://tches.iacr.org/index.php/TCHES/article/view/9064. 1, 2, 5, 7.4

[Ble98] Daniel Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In Hugo Krawczyk, editor, Advances in Cryptology – CRYPTO'98, volume 1462 of Lecture Notes in Computer Science, pages 1–12, Santa Barbara, CA, USA, August 23–27, 1998. Springer, Heidelberg, Germany. 2.2.1

[BPR12] Abhishek Banerjee, Chris Peikert, and Alon Rosen. Pseudorandom functions and lattices. In David Pointcheval and Thomas Johansson, editors, Advances in Cryptology – EUROCRYPT 2012, volume 7237 of Lecture Notes in Computer Science, pages 719–737, Cambridge, UK, April 15–19, 2012. Springer, Heidelberg, Germany. 2.3.3

[BRC60] R.C. Bose and D.K. Ray-Chaudhuri. On a class of error correcting binary group codes. Information and Control, 3(1):68 – 79, 1960. 3.3.1

[BS91] Eli Biham and Adi Shamir. Differential cryptanalysis of DES-like cryptosystems. In Alfred J. Menezes and Scott A. Vanstone, editors, Advances in Cryptology – CRYPTO'90, volume 537 of Lecture Notes in Computer Science, pages 2–21, Santa Barbara, CA, USA, August 11–15, 1991. Springer, Heidelberg, Germany. 2.2.2

[Chi64] R. Chien. Cyclic decoding procedures for Bose-Chaudhuri-Hocquenghem codes. IEEE Transactions on Information Theory, 10(4):357–363, 1964. 2.5.1

[CJNP00] Jean-Sébastien Coron, Marc Joye, David Naccache, and Pascal Paillier. New attacks on PKCS#1 v1.5 encryption. In Bart Preneel, editor, Advances in Cryptology – EUROCRYPT 2000, volume 1807 of Lecture Notes in Computer Science, pages 369–381, Bruges, Belgium, May 14–18, 2000. Springer, Heidelberg, Germany. 2.2.1

[CJRR99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In

Michael J. Wiener, editor, Advances in Cryptology – CRYPTO'99, volume 1666 of Lecture Notes in Computer Science, pages 398–412, Santa Barbara, CA, USA, August 15–19, 1999. Springer, Heidelberg, Germany. 5

[CM07] Michael D Ciletti and M Morris Mano. Digital design. Prentice-Hall, 2007. 4.1.1, 4.2

[Cor14] Jean-Sébastien Coron. Higher order masking of look-up tables. In Phong Q. Nguyen and Elisabeth Oswald, editors, Advances in Cryptology – EUROCRYPT 2014, volume 8441 of Lecture Notes in Computer Science, pages 441–458, Copenhagen, Denmark, May 11–15, 2014. Springer, Heidelberg, Germany. 2.4

[CPL+17] Jung Hee Cheon, Sangjoon Park, Joohee Lee, Duhyeong Kim, Yongsoo Song, Seungwan Hong, Dongwoo Kim, Jinsu Kim, Seong-Min Hong, Aaram Yun, Jeongsu Kim, Haeryong Park, Eunyoung Choi, Kimoon kim, Jun-Sub Kim, and Jieun Lee. Lizard. Technical report, National Institute of Standards and Technology, 2017. available at `https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-1-submissions`. 3.1

[CS18] Gaëtan Cassiers and François-Xavier Standaert. Improved bitslice masking: from optimized non-interference to probe isolation. Cryptology ePrint Archive, Report 2018/438, 2018. `https://eprint.iacr.org/2018/438`. 11

[DDGR20] Dana Dachman-Soled, Léo Ducas, Huijing Gong, and Mélissa Rossi. LWE with side information: Attacks and concrete security estimation. In Daniele Micciancio and Thomas Ristenpart, editors, Advances in Cryptology – CRYPTO 2020, Part II, volume 12171 of Lecture Notes in Computer Science, pages 329–358, Santa Barbara, CA, USA, August 17–21, 2020. Springer, Heidelberg, Germany. 3.2.3

[DGJ+19] Jan-Pieter D'Anvers, Qian Guo, Thomas Johansson, Alexander Nilsson, Frederik Vercauteren, and Ingrid Verbauwhede. Decryption failure attacks on IND-CCA secure lattice-based schemes. In Dongdai Lin and Kazue Sako, editors, PKC 2019: 22nd International Conference on Theory and Practice of Public Key Cryptography, Part II, volume 11443 of Lecture Notes in Computer Science, pages 565–598, Beijing, China, April 14–17, 2019. Springer, Heidelberg, Germany. 3.2.2, 5

[DRV20] Jan-Pieter D'Anvers, Mélissa Rossi, and Fernando Virdia. (One) failure is not an option: Bootstrapping the search for failures in lattice-based encryption schemes. In Anne Canteaut and Yuval Ishai, editors, Advances in Cryptology – EUROCRYPT 2020, Part III, volume 12107 of Lecture Notes in Computer Science, pages 3–33, Zagreb, Croatia, May 10–14, 2020. Springer, Heidelberg, Germany. 3.2.2, 3.2.3, 5

[DTVV19] Jan-Pieter D'Anvers, Marcel Tiepelt, Frederik Vercauteren, and Ingrid Verbauwhede. Timing attacks on error correcting codes in post-quantum schemes. Cryptology ePrint Archive, Report 2019/292, 2019. https://eprint.iacr.org/2019/292. 5

[DVV18] Jan-Pieter D'Anvers, Frederik Vercauteren, and Ingrid Verbauwhede. On the impact of decryption failures on the security of LWE/LWR based schemes. Cryptology ePrint Archive, Report 2018/1089, 2018. https://eprint.iacr.org/2018/1089. 3.2.2

[DVV19] Jan-Pieter D'Anvers, Frederik Vercauteren, and Ingrid Verbauwhede. The impact of error dependencies on ring/mod-LWE/LWR based schemes. In Jintai Ding and Rainer Steinwandt, editors, Post-Quantum Cryptography - 10th International Conference, PQCrypto 2019, pages 103–115, Chongqing, China, May 8–10, 2019. Springer, Heidelberg, Germany. 5

[DZD+18] A. Adam Ding, Liwei Zhang, Francois Durvaux, Francois-Xavier Standaert, and Yunsi Fei. Towards sound and optimal leakage detection procedure. In Thomas Eisenbarth and Yannick Teglia, editors, Smart Card Research and Advanced Applications, pages 105–122, Cham, 2018. Springer International Publishing. 5.2.5, 6.5

[FO99a] Eiichiro Fujisaki and Tatsuaki Okamoto. How to enhance the security of public-key encryption at minimum cost. In Hideki Imai and Yuliang Zheng, editors, PKC'99: 2nd International Workshop on Theory and Practice in Public Key Cryptography, volume 1560 of Lecture Notes in Computer Science, pages 53–68, Kamakura, Japan, March 1–3, 1999. Springer, Heidelberg, Germany. 2.2.2

[FO99b] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In Michael J. Wiener, editor, Advances in Cryptology – CRYPTO'99, volume 1666 of Lecture Notes in Computer Science, pages 537–554, Santa Barbara, CA, USA, August 15–19, 1999. Springer, Heidelberg, Germany. 2.2.2

[For65] G David Forney. Concatenated codes. 1965. 7.1.1

[FPS19] Tim Fritzmann, Thomas Pöppelmann, and Johanna Sepúlveda. Analysis of error-correcting codes for lattice-based key exchange. In Carlos Cid and Michael J. Jacobson Jr:, editors, SAC 2018: 25th Annual International Workshop on Selected Areas in Cryptography, volume 11349 of Lecture Notes in Computer Science, pages 369–390, Calgary, AB, Canada, August 15–17, 2019. Springer, Heidelberg, Germany. 3.3.1, 5

[GGJR+11] Benjamin Jun Gilbert Goodwill, Josh Jaffe, Pankaj Rohatgi, et al. A testing methodology for side-channel resistance validation. pages 115–136, 2011. 11, 5, 6

[GM84] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. Journal of Computer and System Sciences, 28(2):270 – 299, 1984. 2.2.2

[GMK16] Hannes Gross, Stefan Mangard, and Thomas Korak. Domain-oriented masking: Compact masked hardware implementations with arbitrary protection order. Cryptology ePrint Archive, Report 2016/486, 2016. https://eprint.iacr.org/2016/486. 11

[Gra] Frank Gray. Pulse code communication. 3.3.2

[GVL89] Gene H. Golub and Charles F. Van Loan. Matrix Computations. The Johns Hopkins University Press, second edition, 1989. 4.1

[GZB+19] Oscar Garcia-Morchon, Zhenfei Zhang, Sauvik Bhattacharya, Ronald Rietman, Ludo Tolhuizen, Jose-Luis Torre-Arce, Hayo Baan, Markku-Juhani O. Saarinen, Scott Fluhrer, Thijs Laarhoven, and Rachel Player. Round5. Technical report, National Institute of Standards and Technology, 2019. available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-2-submissions. 3.3.1, 5

[Ham19] Mike Hamburg. Three Bears. Technical report, National Institute of Standards and Technology, 2019. available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-2-submissions. 3.3.1, 5

[Han13] Yunghsiang S. Han. BCH Codes, 2013. https://web.ntpu.edu.tw/~yshan/BCH_code.pdf. 2.5.1

[HHK17] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. A modular analysis of the Fujisaki-Okamoto transformation. In Yael Kalai and Leonid Reyzin, editors, TCC 2017: 15th Theory of Cryptography Conference, Part I, volume 10677 of Lecture Notes in Computer Science, pages 341–371, Baltimore, MD, USA, November 12–15, 2017. Springer, Heidelberg, Germany. 2.2.2

[ISW03] Yuval Ishai, Amit Sahai, and David Wagner. Private circuits: Securing hardware against probing attacks. In Dan Boneh, editor, Advances in Cryptology – CRYPTO 2003, volume 2729 of Lecture Notes in Computer Science, pages 463–481, Santa Barbara, CA, USA, August 17–21, 2003. Springer, Heidelberg, Germany. 2.4

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael J. Wiener, editor, Advances in Cryptology – CRYPTO'99, volume 1666 of Lecture Notes in Computer Science, pages 388–397, Santa Barbara, CA, USA, August 15–19, 1999. Springer, Heidelberg, Germany. 2.4

[Koc96] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In Neal Koblitz, editor, Advances in Cryptology – CRYPTO'96, volume 1109 of Lecture Notes in Computer Science, pages 104–113, Santa Barbara, CA, USA, August 18–22, 1996. Springer, Heidelberg, Germany. (document), 1.1, 2.4, 5

[LLJ+19] Xianhui Lu, Yamin Liu, Dingding Jia, Haiyang Xue, Jingnan He, Zhenfei Zhang, Zhe Liu, Hao Yang, Bao Li, and Kunpeng Wang. LAC. Technical report, National Institute of Standards and Technology, 2019. available at `https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-2-submissions`. 2.3.2, 3.3.1, 5

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert, editor, Advances in Cryptology – EUROCRYPT 2010, volume 6110 of Lecture Notes in Computer Science, pages 1–23, French Riviera, May 30 – June 3, 2010. Springer, Heidelberg, Germany. 2.3.2, 3

[LPR12] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. Cryptology ePrint Archive, Report 2012/230, 2012. `https://eprint.iacr.org/2012/230`. 2.3.2

[LS12] Adeline Langlois and Damien Stehlé. Worst-case to average-case reductions for module lattices. Cryptology ePrint Archive, Report 2012/090, 2012. `https://eprint.iacr.org/2012/090`. 3

[Mar90] Marvin Marcus. Determinants of sums. The College Mathematics Journal, 21(2):130–135, 1990. 5.1

[Mat94] Mitsuru Matsui. Linear cryptanalysis method for DES cipher. In Tor Helleseth, editor, Advances in Cryptology – EUROCRYPT'93, volume 765 of Lecture Notes in Computer Science, pages 386–397, Lofthus, Norway, May 23–27, 1994. Springer, Heidelberg, Germany. 2.2.2

[MFS20] Georg Maringer, Tim Fritzmann, and Johanna Sepúlveda. The influence of LWE/RLWE parameters on the stochastic dependence of decryption failures. In Weizhi Meng, Dieter Gollmann, Christian Damsgaard Jensen, and Jianying Zhou, editors, ICICS 20: 22nd International Conference on Information and Communication Security, volume 11999 of Lecture Notes in Computer Science, pages 331–349, Copenhagen, Denmark, August 24–26, 2020. Springer, Heidelberg, Germany. 3.2, 7.1

[MOW17] David McCann, Elisabeth Oswald, and Carolyn Whitnall. Towards practical tools for side channel aware software engineering: 'grey box' modelling for instruction leakages. In Engin Kirda and Thomas Ristenpart, editors, USENIX Security 2017: 26th USENIX Security Symposium, pages 199–216, Vancouver, BC, Canada, August 16–18, 2017. USENIX Association. 11, 5, 5.2.5, 6, 6.5

[MPWZ23] Georg Maringer, Sven Puchinger, and Antonia Wachter-Zeh. Information- and Coding-Theoretic Analysis of the RLWE/MLWE channel. IEEE Transactions on Information Forensics and Security, 18:549–564, 2023. 7.1

[NAB+20] Michael Naehrig, Erdem Alkim, Joppe Bos, Léo Ducas, Karen Easterbrook, Brian LaMacchia, Patrick Longa, Ilya Mironov, Valeria Nikolaenko, Christopher Peikert, Ananth Raghunathan, and Douglas Stebila. FrodoKEM. Technical report, National Institute of Standards and Technology, 2020. available at `https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions`. 3.3.2

[PAA+19] Thomas Poppelmann, Erdem Alkim, Roberto Avanzi, Joppe Bos, Léo Ducas, Antonio de la Piedra, Peter Schwabe, Douglas Stebila, Martin R. Albrecht, Emmanuela Orsini, Valery Osheter, Kenneth G. Paterson, Guy Peer, and Nigel P. Smart. NewHope. Technical report, National Institute of Standards and Technology, 2019. available at `https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-2-submissions`. 2.3.2

[Pet60] W. W. Peterson. Encoding and error-correction procedures for the bose-chaudhuri codes. IRE Trans. Inf. Theory, 6:459–470, 1960. 2.5.1

[PKC91] PKCS #1: RSA cryptography standard. RSA Data Security, Inc., June 1991. 2.2.1

[PQC16a] Federal register. 81:92787–92788, 2016. 2.2.2

[PQC16b] Submission requirements and evaluation criteria for the post-quantum cryptography standardization process. 2016. `https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf`. 2.2.2, 11

[Reg05] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In Harold N. Gabow and Ronald Fagin, editors, 37th Annual ACM Symposium on Theory of Computing, pages 84–93, Baltimore, MA, USA, May 22–24, 2005. ACM Press. 2.3.1, 2.3.1

[RRCB20] Prasanna Ravi, Sujoy Sinha Roy, Anupam Chattopadhyay, and Shivam Bhasin. Generic side-channel attacks on CCA-secure lattice-based PKE and KEMs. IACR Transactions on Cryptographic Hardware and Embedded Systems, 2020(3):307–335, 2020. `https://tches.iacr.org/index.php/TCHES/article/view/8592`. 11, 5, 5.2.1

[RS92] Charles Rackoff and Daniel R. Simon. Non-interactive zero-knowledge proof of knowledge and chosen ciphertext attack. In Joan Feigenbaum,

editor, Advances in Cryptology – CRYPTO'91, volume 576 of Lecture Notes in Computer Science, pages 433–444, Santa Barbara, CA, USA, August 11–15, 1992. Springer, Heidelberg, Germany. 2.2.2

[Saa17] Markku-Juhani O. Saarinen. On reliability, reconciliation, and error correction in ring-LWE encryption. Cryptology ePrint Archive, Report 2017/424, 2017. https://eprint.iacr.org/2017/424. 3.3.1

[SAB+19] Peter Schwabe, Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Gregor Seiler, and Damien Stehlé. CRYSTALS-KYBER. Technical report, National Institute of Standards and Technology, 2019. available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-2-submissions. 2.3.1

[SAB+22] Peter Schwabe, Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Gregor Seiler, Damien Stehlé, and Jintai Ding. CRYSTALS-KYBER. Technical report, National Institute of Standards and Technology, 2022. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022. 2.2.2

[SHR+22] Thomas Schamberger, Lukas Holzbaur, Julian Renner, Antonia Wachter-Zeh, and Georg Sigl. A power side-channel attack on the reed-muller reed-solomon version of the HQC cryptosystem. Cryptology ePrint Archive, Report 2022/724, 2022. https://eprint.iacr.org/2022/724. 5

[TR17] Bashar Tahir and Markus Rupp. New construction and performance analysis of polar codes over awgn channels. In 2017 24th International Conference on Telecommunications (ICT), pages 1–4, 2017. 7.3.1

[TU16] Ehsan Ebrahimi Targhi and Dominique Unruh. Post-quantum security of the Fujisaki-Okamoto and OAEP transforms. In Martin Hirt and Adam D. Smith, editors, TCC 2016-B: 14th Theory of Cryptography Conference, Part II, volume 9986 of Lecture Notes in Computer Science, pages 192–216, Beijing, China, October 31 – November 3, 2016. Springer, Heidelberg, Germany. 2.2.2

[Wic94] Stephen B. Wicker. Error Control Systems for Digital Communication and Storage. Prentice-Hall, Inc., USA, 1994. 2.5.1

[WL14] Runxin Wang and Rongke Liu. A novel puncturing scheme for polar codes. IEEE Communications Letters, 18(12):2081–2084, 2014. 7.3.2

[WL21] Jiabo Wang and Cong Ling. Polar coding for ring-LWE-based public key encryption. Cryptology ePrint Archive, Report 2021/619, 2021. https://eprint.iacr.org/2021/619. 7, 3.3.1, 5, 7.1

[WLS14] Daolong Wu, Ying Li, and Yue Sun. Construction and block error rate analysis of polar codes over awgn channel based on gaussian approximation. IEEE Communications Letters, 18(7):1099–1102, 2014. 7.3.1

[WPH+22] Yingchen Wang, Riccardo Paccagnella, Elizabeth Tang He, Hovav Shacham, Christopher W. Fletcher, and David Kohlbrenner. Hertzbleed: Turning power side-channel attacks into remote timing attacks on x86. In Kevin R. B. Butler and Kurt Thomas, editors, USENIX Security 2022: 31st USENIX Security Symposium, pages 679–697, Boston, MA, USA, August 10–12, 2022. USENIX Association. 2.4

[WR19] Matthew Walters and Sujoy Sinha Roy. Constant-time BCH error-correcting code. Cryptology ePrint Archive, Report 2019/155, 2019. https://eprint.iacr.org/2019/155. 3.3.1, 5

[XD19] Hong Xiang and Jintai Ding. The latest Progress of PQC Competition in China. ETSI/IQC Quantum Safe Cryptography Workshop 2019, 2019. 3.3.1