Universidade Estadual de Campinas
Instituto de Computação

Antonio Carlos Guimarães Junior

Secure and efficient software implementation of QC-MDPC code-based cryptography

Implementação segura e eficiente em software de criptografia baseada em códigos QC-MDPC

CAMPINAS
2019

Antonio Carlos Guimarães Junior

Secure and efficient software implementation of QC-MDPC code-based cryptography

Implementação segura e eficiente em software de criptografia baseada em códigos QC-MDPC

Dissertação apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Thesis presented to the Institute of Computing of the University of Campinas in partial fulfillment of the requirements for the degree of Master in Computer Science.

Supervisor/Orientador: Prof. Dr. Diego de Freitas Aranha
Co-supervisor/Coorientador: Prof. Dr. Edson Borin

Este exemplar corresponde à versão final da Dissertação defendida por Antonio Carlos Guimarães Junior e orientada pelo Prof. Dr. Diego de Freitas Aranha.

CAMPINAS
2019

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Ana Regina Machado - CRB 8/5467

G947s

Guimarães Junior, Antonio Carlos, 1994-
Secure and efficient software implementation of QC-MDPC code-based cryptography / Antonio Carlos Guimarães Junior. – Campinas, SP : [s.n.], 2019.

Orientador: Diego de Freitas Aranha.
Coorientador: Edson Borin.
Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Criptografia. 2. Criptografia pós-quântica. 3. Criptografia de chaves públicas. 4. Códigos corretores de erros (Teoria da informação). I. Aranha, Diego de Freitas, 1982-. II. Borin, Edson, 1979-. III. Universidade Estadual de Campinas. Instituto de Computação. IV. Título.

Informações para Biblioteca Digital

Título em outro idioma: Implementação segura e eficiente em software de criptografia baseada em códigos QC-MDPC
Palavras-chave em inglês:
Cryptography
Post-quantum cryptography
Public key cryptography
Error-correcting codes (Information theory)
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Edson Borin [Coorientador]
Marcos Antonio Simplicio Junior
Julio César López Hernández
Data de defesa: 19-03-2019
Programa de Pós-Graduação: Ciência da Computação

Identificação e informações acadêmicas do(a) aluno(a)
- ORCID do autor: https://orcid.org/0000-0001-5110-6639
- Currículo Lattes do autor: http://lattes.cnpq.br/3952604251815458

Universidade Estadual de Campinas
Instituto de Computação

Antonio Carlos Guimarães Junior

Secure and efficient software implementation of QC-MDPC code-based cryptography

Implementação segura e eficiente em software de criptografia baseada em códigos QC-MDPC

Banca Examinadora:

- Prof. Dr. Edson Borin
  IC/UNICAMP
- Prof. Dr. Marcos Antonio Simplicio Junior
  POLI/USP
- Prof. Dr. Julio Cesar López Hernández
  IC/UNICAMP

A ata da defesa, assinada pelos membros da Comissão Examinadora, consta no SIGA/Sistema de Fluxo de Dissertação/Tese e na Secretaria do Programa da Unidade.

Campinas, 19 de março de 2019

Agradecimentos

Gostaria de agradecer a todos que de alguma forma contribuíram para realização deste trabalho. As inestimáveis contribuições que recebi foram fundamentais para obtenção dos resultados alcançados e consequente conclusão do curso de mestrado. Dentre as muitas pessoas e instituições que merecem ser aqui mencionadas, eu gostaria de agradecer, em especial:

- Aos meus pais, Antônio Carlos Guimarães e Evani Moreira Cézar Guimarães.
- Ao meu orientador, Professor Diego Aranha e ao meu coorientador, Professor Edson Borin.
- À Intel e à Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP) por patrocinarem o projeto *Execução segura de algoritmos criptográficos*, processo nº 2014/50704-7, do qual minha pesquisa fez parte.
- Aos professores, pesquisadores e colegas que participaram do projeto acima mencionado. Em especial ao Professor Julio López e ao Professor Ricardo Dahab; e aos pesquisadores da Intel, Marcio Juliato e Rafael Misoczki.
- Ao Professor Marcos Simplicio e ao Professor Sandro Rigo, que participaram de minhas bancas de defesa e exame de qualificação.
- Aos demais professores do Instituto de Computação da Unicamp, em especial à Professora Islene Garcia.
- Aos funcionários do Instituto de Computação da Unicamp.
- Aos laboratórios LASCA e LMCAD; e aos colegas que deles participam.
- À Microsoft por fornecer a infraestrutura de nuvem computacional necessária a diversos dos experimentos realizados.
- À minha família e amigos.

Resumo

A expectativa do surgimento de computadores quânticos impulsiona uma transição sem precedentes na área de criptografia de chave pública. Algoritmos convencionais, representados principalmente por criptografia baseada em curvas elípticas [41] e pelo RSA [59], são vulneráveis a ataques utilizando computadores quânticos e, portanto, precisarão ser substituídos. Criptosistemas baseados em códigos corretores de erros são considerados alguns dos candidatos mais promissores para substituí-los em esquemas de encriptação. Entre as famílias de códigos, os códigos QC-MDPC [51] alcançam os menores tamanhos de chave, enquanto mantêm as propriedades de segurança desejadas. Seu desempenho, no entanto, ainda precisa ser melhorado para atingir um nível competitivo.

Este trabalho tem ênfase na otimização do desempenho dos criptosistemas baseados em código QC-MDPC através de melhorias em suas implementações e algoritmos. Primeiramente, é apresentada uma nova versão aprimorada do mecanismo de encapsulamento de chaves da QcBits [16], uma implementação em tempo constante do Criptosistema Niederreiter [56] utilizando códigos QC-MDPC. Nesta versão, os parâmetros da implementação foram atualizados para atender ao nível de segurança quântica de 128 bits, alguns dos principais algoritmos foram substituídos para evitar o uso de instruções mais lentas, o código foi inteiramente vetorizado utilizando o conjunto de instruções AVX 512 e outras pequenas melhorias foram introduzidas. Comparando com o atual estado-da-arte para códigos QC-MDPC, a implementação BIKE [2], a implementação apresentada neste trabalho executa 1,9 vezes mais rápido ao decriptar mensagens.

Em seguida, foca-se na otimização de desempenho dos sistemas criptográficos baseados em códigos QC-MDPC por meio da inserção de uma taxa de falhas configurável em seus procedimentos aritméticos. São apresentados algoritmos com execução em tempo constante que aceitam uma taxa de falhas configurável para multiplicação e inversão sobre polinômios binários, as duas sub-rotinas mais caras utilizadas nas implementações QC-MDPC. Usando uma taxa de falhas negligível comparada ao nível de segurança (2^{-128}), a multiplicação é 2 vezes mais rápida que a multiplicação utilizada pela biblioteca NTL [63] em polinômios esparsos e 1,6 vezes mais rápida que uma multiplicação polinomial esparsa ingênua em tempo constante. O algoritmo de inversão, baseado no algoritmo de Wu *et al.* [68], é 2 vezes mais rápido que o original e 12 vezes mais rápido que o algoritmo de inversão de Itoh e Tsujii [40] utilizando o mesmo polinômio de módulo ($x^{32749} - 1$). Ao inserir esses algoritmos na versão aprimorada da QcBits, atingiu-se uma aceleração de 1,9 na geração de chaves e de até 1,4 na decriptação.

Comparando com a BIKE, a versão final da QcBits apresentada neste trabalho executa a decriptação uniforme 2,7 vezes mais rápida. Além disso, as técnicas aqui apresentadas também podem ser aplicadas à BIKE, abrindo novas possibilidades de melhorias para criptosistemas QC-MDPC.

Abstract

The emergence of quantum computers is pushing an unprecedented transition in the public key cryptography field. Conventional algorithms, mostly represented by elliptic curves [41] and RSA [59], are vulnerable to attacks using quantum computers and need, therefore, to be replaced. Cryptosystems based on error-correcting codes are considered some of the most promising candidates to replace them for encryption schemes. Among the code families, QC-MDPC codes [51] achieve the smallest key sizes while maintaining the desired security properties. Their performance, however, still needs to be greatly improved to reach a competitive level.

In this work, we focus on optimizing the performance of QC-MDPC code-based cryptosystems through improvements concerning both their implementations and algorithms. We first present a new enhanced version of QcBits' key encapsulation mechanism [16], which is a constant time implementation of the Niederreiter cryptosystem [56] using QC-MDPC codes. In this version, we updated the implementation parameters to meet the 128-bit quantum security level, replaced some of the core algorithms avoiding slower instructions, vectorized the entire code using the AVX 512 instruction set extension and introduced some other minor improvements. Comparing with the current state-of-the-art implementation for QC-MDPC codes, the BIKE implementation [2], our code performs 1.9 times faster when decrypting messages.

We then optimize the performance of QC-MDPC code-based cryptosystems through the insertion of a configurable failure rate in their arithmetic procedures. We present constant time algorithms with a configurable failure rate for multiplication and inversion over binary polynomials, the two most expensive subroutines used in QC-MDPC implementations. Using a failure rate negligible compared to the security level (2^{-128}), our multiplication is 2 times faster than the one used in the NTL library [63] on sparse polynomials and 1.6 times faster than a naive constant-time sparse polynomial multiplication. Our inversion algorithm, based on the inversion algorithm of Wu *et al.* [68], is 2 times faster than the original and 12 times faster than the inversion algorithm of Itoh and Tsujii [40] using the same modulus polynomial ($x^{32749} - 1$). By inserting these algorithms in our enhanced version of QcBits, we were able to achieve a speedup of 1.9 on the key generation and up to 1.4 on the decryption time.

Comparing with BIKE, our final version of QcBits performs the uniform decryption 2.7 times faster. Moreover, the techniques presented in this work can also be applied to BIKE, opening new possibilities for further improvements.

Contents

Chapter 1

Introduction

Transmitting information has been a major necessity for society throughout history. While the basic concept is universally known, the transmission channels have greatly evolved over time. The technological development enabled much faster transmission channels, facilitating communication and enabling new possibilities of use. It also introduced new challenges to achieve some important and common requirements. Among the many research fields raised from the necessities and particularities of transmitting information, two of them are of our particular interest: the fields of error correcting codes and cryptography. The first emerged from the fact that transmitting information is an imperfect process. Physical transmission channels invariably introduce errors in the information and they need to be removed. The second is responsible to achieve or verify properties such as confidentiality, authentication, and integrity.

In this work, we provide a brief background about these two fields, but our major interest is in a specific intersection between them: the code-based cryptography field. This field started with Robert McEliece's discovery that it was possible to achieve some cryptographic properties using error-correcting codes. In 1978, he presented the McEliece Cryptosystem [47], the first code-based public-key encryption scheme. At the time, the public-key cryptography field as a whole was still a novelty. It had been only two years since Diffie and Hellman published the famous "New Directions in Cryptography" [20], marking the public discovery of the field. Many public-key cryptosystems were derived from their work. Notably, also in 1978, Rivest, Shamir, and Adleman presented the RSA algorithm [59], a number-theoretic public-key cryptosystem which would later become a standard in public-key cryptography.

In its original form, McEliece's cryptosystem has great performance and is more efficient than the RSA, but it relies on very large keys. For example, it requires keys with 460Kb (kilo-bits) to achieve an 80-bit classical security level. In a time when computers used to have just a few tens of kilobytes of RAM, this was a major drawback for the cryptosystem. Even the performance advantage was lost when, in 1985, Koblitz and Miller independently presented the elliptic curve cryptography (ECC) [41, 50]. ECC cryptosystems not only enabled the use of much smaller keys but also present better performance than both the RSA and McEliece cryptosystem.

RSA and ECC are the current standards of public-key cryptography. They are considered secure techniques and they suffice the present needs. However, the computing field

might be on the verge of a new technological breakthrough: the creation of large scalable quantum computers. Developing applications which could benefit from them is mostly an open field of study, but their impact on the current public-key cryptography standard has long been known. In 1994, Peter Shor formulated an algorithm that can solve integer factorization in polynomial time using a quantum computer [61, 62]. This is the problem in which the RSA is based, and a polynomial-time solution for it entirely undermines the security. The same occurs with the discrete logarithm problem, the base of the ECC and Diffie-Hellman key exchange.

While the community diverges over predictions, some specialists foresee quantum computers capable of breaking the 2048-bit RSA in the next few decades [54]. Therefore, a secure and efficient replacement for the current standard of public-key cryptography is necessary. This scenario creates a new opportunity for the McEliece cryptosystem, 40 years after its creation. Based on a known NP-complete problem (the decoding of general linear codes), the cryptosystem has so far shown to be resistant against attacks using quantum computers and is one of the promising candidates to become the next standard for encryption in public-key cryptography.

Overshadowed by the RSA and ECC, the McEliece cryptosystem and, more generally, the code-based cryptography field had diminished progress until the 2000s. Most of its original problems are still present in many of current implementations. Derivatives using smaller keys have been proposed, but they usually result in the introduction of vulnerabilities. An exception to that is the implementation of McEliece using QC-MDPC codes, which is believed to be secure. Presented in 2013 by Misoczki *et al.* [51], the cryptosystem provides keys about 100 times smaller than the original McEliece, but it comes at the cost of deteriorating the performance and introducing perceptible failure rates to the decryption process. In this context, a performance improvement to QC-MDPC code-based cryptosystems is necessary and, in this work, we present some contributions toward this goal.

1.1 Objective

Our general objective in this work is to contribute to the implementation of secure and efficient cryptosystems based on error-correcting codes. To do this, we consider three main aspects:

- **Key-size**: A 2016 NIST report [15] highlighted the key size as a primordial factor to be considered on post-quantum cryptosystems. Thus, we opt for the use of QC-MDPC codes, since, for cryptographic purposes, they seem to be the most reliable code family featuring compact keys.

- **Efficiency**: Code-based cryptosystems using QC-MDPC codes are significantly slower than the original McEliece, which uses Goppa codes [31]. In this work, we pursue performance enhancement for these cryptosystems contributing not only to the development of implementation techniques but also to the improvement of their algorithms.

- **Side-channel protection**: Side channel protection is an important requirement for any modern cryptographic implementation. It is not possible to guarantee side-channel protection for an algorithm or implementation without deep knowledge of the machine executing them. However, taking as bases broadly used architectures such as Intel x86 and its extensions, we design all contributions presented in this work to avoid the most common side-channel vulnerabilities. In specific, all implementations presented feature constant-time or uniform execution.

1.2 Contributions

The contributions presented in this work are divided into three sets. The first two concern the development of new implementation techniques aiming at optimizing the performance of QcBits, an implementation of QC-MDPC code-based cryptography. The last one concerns the presentation of improvements in the basic arithmetic algorithms necessary to implement a QC-MDPC code-based cryptosystem. This set presents contributions much more generic and that can be explored in other fields of cryptography, even though they were planned in a specific context. The contributions are summarized below.

- An optimization of the decoding process of the original QcBits implementation.
  - We achieve a speedup of up to 4.8 times over the original implementation through the use of techniques such as vectorization, loop unrolling, and precalculation.
  - We estimate that gains could be as high as 5.06 times considering the introduction of simple and generic extensions to the Intel x86 architecture.
  - We mitigate of all known power vulnerabilities found in the original implementation with an almost negligible ($< 1\%$) impact on the overall performance.
- A new enhanced version of QcBits.
  - We update the security level from 80-bit classical security level to 128-bit quantum security level.
  - We vectorize the entire implementation using the AVX512 instruction set extension.
  - We replace some of the core algorithms with others that have a better performance in face of the new AVX512 instructions.
  - Comparing to BIKE, the current state-of-the-art of QC-MDPC code-based cryptography, this implementation decrypts messages 1.9 times faster.
- A method to accelerate the arithmetic algorithms used to implement QC-MDPC code-based cryptosystems.
  - We introduce the concept of using arithmetic subroutines with a controlled failure rate to accelerate QC-MDPC code-based cryptosystems.

- We present constant-time algorithms for multiplication and inversion over binary polynomials that operate with configurable failure rates.
- We define methods to obtain a correlation between failure rate and performance improvement for each algorithm.
- We show that these algorithms provide a significant performance improvement while introducing an arithmetic failure rate that is negligible compared to the security level of the cryptosystem.
- By introducing these algorithms in our enhanced version of QcBits, we achieve a speedup of 1.9 times on the key generation and 1.4 times on the decryption process. Comparing with BIKE, our final version of QcBits performs the uniform decryption 2.7 times faster.

The first two sets of contributions were published at the *Brazilian Symposium on High-Performance Computational Systems* (WSCAD-2017) [35] and at Wiley's *Concurrency and Computation: Practice and Experience* (CCPE) journal [36]. The last set of contributions is currently under submission.

1.3 Structure

Chapter 2 presents the basic theoretical background necessary in this work. It aims at being as self-contained as possible. Therefore, most of the chapter presents very basic concepts about cryptography and error-correcting codes. If the reader is familiar with these topics, the reading of Subsection 2.4.3 should be enough for understanding this work.

Chapters 3 and 4 present our contributions towards the performance improvement of QC-MDPC code-based cryptosystems. Each chapter starts with a brief summary of contributions and ends with a small discussion about the achieved results. An overall discussion is provided in Chapter 5, as well as some possible future work.

Chapter 2

Theoretical Basis

The understanding of code-based cryptography requires knowledge of two distinct fields: cryptography and error-correcting codes. The former defines the goals and properties to be achieved, whereas the latter provides the tools to achieve them. While these are extensive fields, the broadness of concepts presented in this chapter is mostly restricted to the information necessary for the understanding of the basic functioning of QC-MDPC code-based cryptosystems and, furthermore, the proposals and contributions presented in this work.

2.1 Cryptography

As defined by Menezes *et al.* in the Handbook of Applied Cryptography [48], cryptography is *"the study of mathematical techniques related to aspects of information security such as confidentiality, data integrity, entity identification, and data origin authentication"*. Figure 2.1 illustrates these aspects. Two parties, **A** and **B**, want to communicate through an arbitrary channel. This channel is open, in the sense that a third party, **C**, can read and modify freely the information being transmitted.

The *confidentiality* aspect enables **A** to communicate with **B** through the channel without **C** understanding the message contents. In fact, **C** should not be able to differentiate the encrypted information from random noise. The *data integrity* enables **A** and **B**, upon receiving a message, to verify if its content was modified since it was sent by the trusted party. This aspect should enable the detection of modifications caused not only by transmission errors in the channel but also by deliberated attempts of data tempering by **C**. *Entity identification* enables **A** and **B** to unequivocally identify themselves to other parties, and *data origin authentication* ensures whether or not a certain message was written by the party who claims it.

In this section, we will present the basic concepts of symmetric and public-key cryptography, as well as an introduction to the post-quantum cryptography.

2.1.1 Symmetric Cryptography

In a symmetric cryptosystem, **A** and **B** own a common piece of secret information, called shared secret, and use it to encrypt and decrypt messages. The communication can be

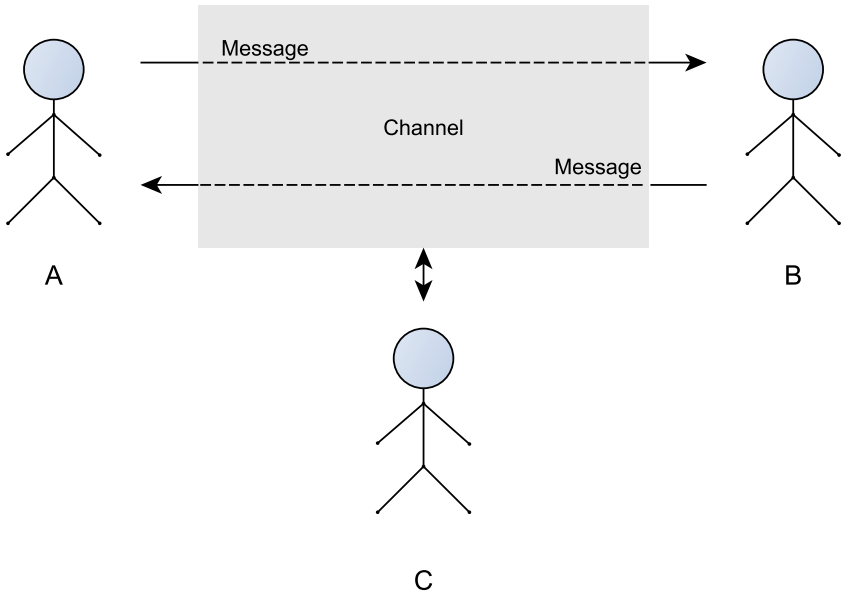


Figure 2.1: Basic communication illustration.

performed symmetrically in both directions. The AES [19] is the current standard and the most used algorithm of symmetric cryptography. It was originally presented as Rijndael and became the standard for symmetric encryption in 2001. The basic algorithm only provides confidentiality, but it can be easily extended to provide data integrity, entity identification, and data origin authentication with the help of hash functions. Currently, most of the computer architectures feature hardware implementation of the AES or extensions to accelerate its software implementation, making it a very fast algorithm.

Even without hardware support, symmetric cryptosystems usually are very efficient and have their security level directly defined as the size of the secret key. A major problem, however, is their dependency on the shared secret. In order to establish a secure communication channel, the parties must first combine the shared secret. Using only symmetric cryptosystems, however, to securely combine the shared secret, it is necessary to have a prior secure communication channel. Solving this deadlock is one of the main purposes of public-key cryptography.

2.1.2 Public-key Cryptography

Before the public-key cryptography discovery [20], in order to establish an encrypted communication channel, it was necessary to have a prior secure channel to combine a shared secret. Historically, this secure channel used to be a trusted courier or face-to-face meetings, which would be a great drawback in the modern computing age. The public-key cryptography enables two new possibilities: to define encryption schemes that work without a shared secret; and to securely combine a shared secret without a prior secure channel.

Public-key encryption schemes function through the use of different keys for encryption and decryption. They usually operate with two keys: the public key, used in message

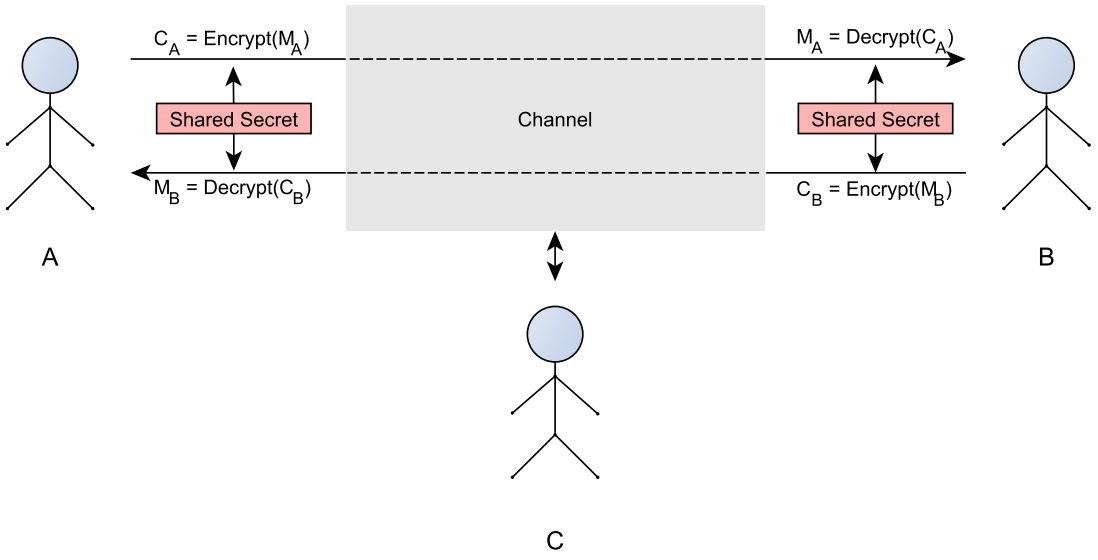


Figure 2.2: Symmetric cryptography illustration.

encryption and signature verification; and the private key, used in message decryption and signature process. In general, the public-key cryptography capabilities are based on the hardness of obtaining private information from the knowledge of public information, which is granted by one-way functions, such as integer multiplication and modular exponentiation. Figure 2.3 illustrates an encryption and signature scheme using public-key cryptography. The public-key is publicly distributed and anyone can encrypt a message or verify whether a signature is valid. Once encrypted, only those who know the private key (**A**) are able to decrypt the ciphertext (encrypted message).

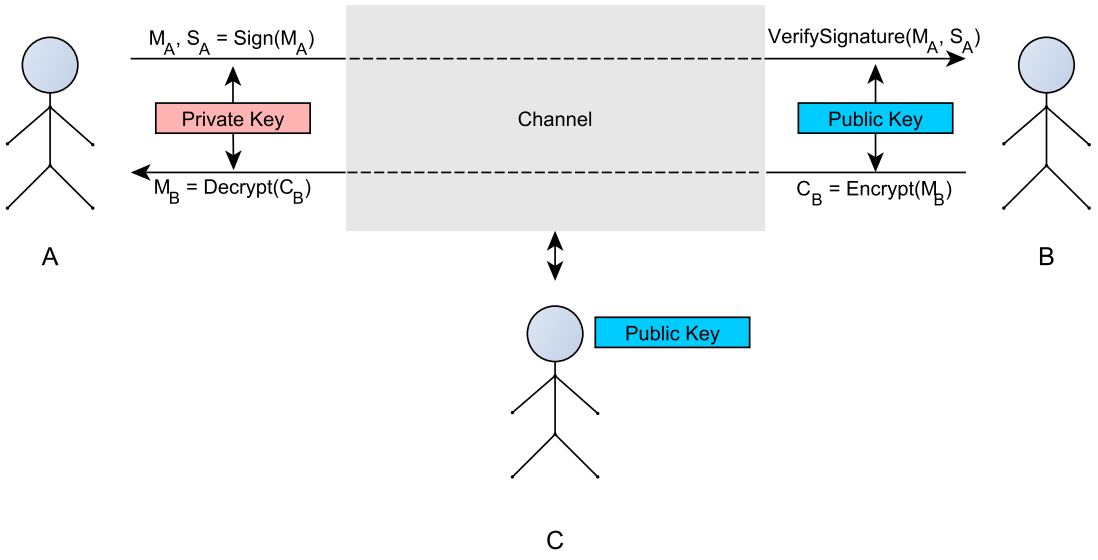


Figure 2.3: Public-key cryptography illustration.

The Diffie-Hellman (DH) key exchange protocol [20] and the RSA [59] are some of the

main representatives of public-key cryptography. The first was published 1976, marking the discovery of public key cryptography. It exploits the hardness of discrete logarithm to combine a shared secret securely using an insecure channel. The second was presented in 1978 and consists of an encryption and signature scheme exploiting the difficulty of factorizing the product of prime integers.

Encryption schemes, such as RSA, can also be used in a key encapsulation mechanism (KEM) to define a shared secret. The basic difference is that, instead of a message, the cryptosystem is used to encrypt the shared secret which will be then transmitted to the other party.

2.1.3 Post-Quantum Cryptography

No algorithm to solve integer factorization or discrete logarithm in polynomial time using a conventional computer is known. The best methods proposed are sub-exponential [1], but there are no proofs that assure the exact hardness of the problems. In a quantum computer, however, polynomial time algorithms that can solve them were already presented [61, 62]. Although the development of quantum computers could be considered as still in its beginning, it is necessary to have secure and efficient public key cryptography algorithms that are resistant to attacks by a quantum computer.

Algorithms that do not rely on the aforementioned problems are known since the late 70s and today constitute the post-quantum cryptography field. In December 2016, the USA's National Institute of Standards and Technology (NIST) started the standardization process for post-quantum public key cryptography by publishing a Call for Proposals [57]. A previous report [15] of the Institute presented some of the promising areas for the concourse, among them: Lattice-based cryptography, Code-based cryptography, Multivariate polynomial cryptography, and Hash-based signatures.

Impact on symmetric encryption

The security of symmetric cryptosystems is also affected by attacks using quantum computers. Hence, the definition of security level needs to be changed to specify the computing model considered (classical or quantum). An N-bit *classical security level* is defined as the computational effort necessary to perform an exhaustive search on 2^N keys to break, for example, the N-bit AES. Grover's Algorithm [32] on a quantum computer is capable of recovering the key through an exhaustive search in $O(\sqrt{2^N})$ time. In this way, *the quantum security level* is related to the classical one by a square root in the complexity of an exhaustive search algorithm. For example, the 80-bit classical security level corresponds to the 40-bit quantum security level.

2.2 Error-Correcting Codes

Transmitting information is an imperfect process. During the transmission process, many physical factors can lead to information loss and noise insertion, usually in a probabilistic

way. The error correcting codes field dedicates to develop methods to encode and decode information in a way that errors can be detected and, if possible, corrected.

Redundancy is the basic principle that enables error detection and correction capabilities. A very primitive way of correcting, for example, is to just repeat the message an odd number of times and take a majority vote on each bit. This method is functional and the processing time to encode and decode the information is minimal. However, the overhead added to the information is too high compared with the correction capabilities of the method. Moreover, the costs of transmitting the redundant information greatly surpass the processing cost. Thus, it is necessary to have methods presenting smaller overheads and better error correcting capabilities while providing efficient encoding and decoding algorithms.

Better error correcting methods can be designed with the use of parity checks. To illustrate this concept, we first define the *Hamming Weight* of a binary vector as the number of one-value positions in it. In a binary vector A of length n, the parity check bit of the entire vector is the Hamming Weight of A ($HW(A)$) modulus 2, i.e. if $HW(A)$ is even, the parity is 0, otherwise, it is 1. The simplest encoding for error detection is done using just one parity bit for the whole vector, as illustrated in Equation 2.1. The overhead is just one bit and this method is capable of detecting the occurrence of an odd number of errors. The error correction, on the other hand, can only be performed if $n = 1$. Notice that the parity bit is concatenated to the end of the message before the transmission since it must be equally protected. In this way, the parity of the whole block (message + parity check bits) will be 0 if no error occurs.

$$\begin{aligned} &\textit{Message}: && 101001010 \\ &\textit{Parity Check}: && 0 \\ &\textit{Block}: && 101001010\mathbf{0} \end{aligned} \tag{2.1}$$

A little more advanced use of parity check is in the rectangular codes. In this case, the data is organized in a rectangular pattern, as shown by Equation 2.2, where bolded numbers represent the parity check bits. One parity check is calculated for each row and for each column. In the example, the overhead is 6 bits. This code is capable of detecting any two 2 errors and correcting one of them. Depending on where the error occurs, more of them can be detected and corrected. A slightly improved version of rectangular codes are the triangular ones, exemplified in Equation 2.3, where underlined numbers represent padding bits. This code does not present double error detection, but it presents less redundancy than rectangular codes while maintaining the single error correction.

$$\begin{array}{cccc} 1 & 0 & 1 & \mathbf{0} \\ 0 & 0 & 1 & \mathbf{1} \\ 0 & 1 & 0 & \mathbf{1} \\ \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} \end{array} \tag{2.2}$$

$$\begin{array}{ccccc} 1 & 0 & 1 & 0 & \mathbf{0} \\ 0 & 1 & 0 & \mathbf{1} & \\ 1 & 0 & \mathbf{0} & & \\ \underline{0} & \mathbf{1} & & & \\ \mathbf{0} & & & & \end{array} \tag{2.3}$$

A useful metric to evaluate a code is the Rate, which is given by the size of the message over the total size transmitted. For example, the single parity check code has Rate $= \frac{n}{n+1}$,

since it only uses 1 parity check; rectangular codes have rate $\frac{(n-1)^2}{n^2}$; and triangular codes have rate $\frac{n-1}{n+1}$, for messages with $\frac{n(n-1)}{2}$ bits.

Although intuitive for explaining basic codes, geometric representations of data are not useful for more advanced schemes. Thus, we use a more algebraic representation by making lists of the positions checked by each parity bit. Equation 2.4 shows the equivalent lists of indexes for the triangular code of Equation 2.3. These lists can be then represented as a system of equations (Equation 2.5), which, in turn, can be represented in a matrix, one per row, as shown in Equation 2.6. With this last representation, the parity check can be calculated by multiplying the *parity check matrix* and the transposed code-word, as also shown in Equation 2.6. The result of this multiplication is called *syndrome* and it should be composed of zeros if there are no errors in the block.

$$\begin{array}{ll} 1^o: & 1,2,3,4 \\ 2^o: & 4,5,6,7 \\ 3^o: & 3,7,8,9 \\ 4^o: & 2,6,9,10 \\ 5^o: & 1,5,8,10 \end{array} \tag{2.4}$$

$$\begin{aligned} m_1 + m_2 + m_3 + m_4 - p_1 &= 0 \\ m_4 + m_5 + m_6 + m_7 - p_2 &= 0 \\ m_3 + m_7 + m_8 + m_9 - p_3 &= 0 \\ m_2 + m_6 + m_9 + m_{10} - p_4 &= 0 \\ m_1 + m_5 + m_8 + m_{10} - p_5 &= 0 \end{aligned} \tag{2.5}$$

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} m_1 \\ m_2 \\ m_3 \\ m_4 \\ m_5 \\ m_6 \\ m_7 \\ m_8 \\ m_9 \\ m_{10} \\ p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \tag{2.6}$$

Note that, since it represents a system of equations, any elementary row operation can be applied over the parity check matrix. The elementary row operations are listed below.

- Swap two rows.
- Multiply a row by a scalar in $\mathbb{R}^*$.
- Add one row to another.

Two matrices are row equivalent if it is possible to obtain one from the other through a sequence of elementary row operations. A particularly useful row-equivalent matrix is the row-reduced echelon form [49]. Equation 2.7 exemplifies it for the parity check matrix of the triangular code example (Equation 2.6). In each row, the first non-zero element is the only non-zero element of its column and it is located at least one column to the right of the first non-zero element of the previous row. In the example, looking only at the first 10 elements (which corresponds to message bits), the last row have value 0 in all elements. This indicates that only three out of four rows are linearly independent, i.e. any one of four rows can be obtained through a sequence of elementary row operations over the other three. For error correcting codes and, more generally, for solving a system of linear equations, only linearly independent rows are relevant. Therefore, in our example, the last row can be removed from the parity check matrix. The number of linearly independent rows in a matrix is called *Rank* and it can be used to determine the error correction capabilities of a code.

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix} \tag{2.7}$$

2.2.1 Hamming Codes

Hamming codes (HC) [38] are a special class of parity check codes which present the best possible Rate for one error correction on a binary channel. One of its most basic instantiations uses 3 parity check bits to provide one error correction and one error detection for a 4-bit message. Equation 2.8 shows the list of indexes of its parity checks and Equation 2.9 shows the equivalent parity check matrix. To facilitate the decoding process, the parity check bits are inserted in the message on the positions 1, 2 and 4.

$$\begin{array}{ll} 1^o: & 1,3,5,7 \\ 2^o: & 2,3,6,7 \\ 3^o: & 4,5,6,7 \end{array} \tag{2.8}$$

$$H = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix} \tag{2.9}$$

Encoding: As exemplified below, to encode a 4-bit message A, we first insert the parity check bits (p_1, p_2 and p_3) at positions 1, 2 and 4, creating A'. Then, we multiply the parity check matrix, H, by the transposed vector representation of A'. Finally, we solve the simple linear system in Equation 2.10 to obtain the values of p_i. In our example the values are $p_1 = 0$, $p_2 = 1$ and $p_3 = 0$, resulting in the code-word $A' = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 1 & 1 \end{pmatrix}$.

$$A = 1011 = \begin{pmatrix} 1 & 0 & 1 & 1 \end{pmatrix} \longrightarrow A' = \begin{pmatrix} p_1 & p_2 & 1 & p_3 & 0 & 1 & 1 \end{pmatrix}$$

$$H \times A'^{T} = \begin{pmatrix} p_3 + 1 + 1 \\ p_2 + 1 + 1 + 1 \\ p_1 + 1 + 1 \end{pmatrix} \equiv \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \pmod 2 \tag{2.10}$$

Decoding: Similarly to the encoding, we multiply the parity check matrix, H, by the transposed vector representation of A', which, in this case, is the received message possibly containing an error. If the multiplication result, the syndrome, is zero, then the codeword is valid and either there is not an error in the message or there are more errors than the code is capable of correcting/detecting. If the syndrome is different from 0, then the number represented by the syndrome in the binary base is the column in which the error occurred (considering that only one error occurred). The coincidence between the syndrome value and the column number happens because the parity bits were inserted in positions such that each column represents numbers from 1 to 7 in the binary base (bottom-up) sorted from left to right in the parity check matrix. This positioning is not necessary to decode and the columns of matrix H can be at any order. However, if they are not ordered, it is necessary to search for the column which matches the number of the syndrome to identify where the error occurred. Equation 2.11 exemplify the decoding process with error in the $(011_2 = 3_{10})$-th column.

$$\begin{array}{ll} \textit{Block (A')}: & \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 1 & 1 \end{pmatrix} \\ \textit{Error}: & \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix} \\ \textit{Block + Error}: & \begin{pmatrix} 0 & 1 & \mathbf{0} & 0 & 0 & 1 & 1 \end{pmatrix} \\ \textit{Syndrome}: & \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix} \end{array} \tag{2.11}$$

Using a Generator matrix: An alternative method to encode data is the use of a generator matrix, G. Equations 2.12, 2.13, and 2.14 show one of the ways to construct it. First, all parity bits are moved to the end of the message, creating a new parity check matrix, H'. Then, the columns corresponding to parity check bits are removed from H', resulting in the matrix G'. Any elementary row operation can then be applied to G'. Finally, G' is transposed and concatenated with the identity matrix, I. The encoding is performed by simply multiplying the value of the message and G.

$$H = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix} \longrightarrow H' = \begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix} \tag{2.12}$$

$$G' = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \end{pmatrix} \tag{2.13}$$

$$G = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix} \qquad (2.14)$$

Notation: The notation *[n, k, d]-code* usually represents a binary linear code with block length n, message length k and minimum Hamming distance d. The use of square brackets ("[ ]") indicates a linear code and the absence of a base after the closing bracket defaults to the binary base. The example of Hamming Codes presented in this section is a [7,4,3]-code. The Hamming distance between two vectors A and B is the Hamming Weight of $(A \oplus B)$, i.e. the number of positions in which they differ. The minimum Hamming distance is the smallest Hamming distance between any two valid code words. This concept comes from a geometric view of error-correcting codes and is useful to determine the error correction and detection capabilities of a code. The minimum Hamming Distance is also the rank (number of linear independent rows) of the parity check matrix.

2.2.2 LDPC Codes

In 1960, Robert Gallager introduced Low-Density Parity Check (LDPC) Codes [28], a family of parity check codes which presents a very good rate and error correction capability. The generator and parity check matrices are similar to Hamming Codes, as well as the encoding process. We define the density of a matrix as the fraction of one-valued bits that it contains. The main particularity of LDPC over Hamming Codes is in the density of the parity check matrix, which is, as the name states, low. The decoding process is more complex and LDPC codes can be instantiated to correct an arbitrary number of errors.

Basic Construction

In general, the parity check matrix, H, for LDPC codes can be generated in any arbitrary way that results in a relatively low density. If the code is *regular*, then all columns and all rows of H present the same Hamming Weight. The generator matrix creation and the encoding process are essentially the same as for Hamming Codes.

Decoding

There are many published algorithm for decoding LDPC codes. Algorithm 1 shows a very simple version of Gallager's bit-flipping decoding algorithm. It is composed of a syndrome calculation (lines 1 and 7), similar to HC, and the decoding algorithm itself (lines 3 to 6), which is applied iteratively. Although significantly more complex, the decoding slightly reassembles the one of Hamming Codes, since the syndrome is compared to each column of the parity check matrix.

Algorithm 1: Bit-flipping decoding algorithm.

Input : H, c and $Threshold$
Output: c

```
1 s ← H × c
2 while s ≠ 0 do
3 |   foreach column h_i in H do
4 |   |   if HammingWeight(h_i ∧ s) > Threshold then
5 |   |   |   FlipBit(c, i)
6 |   end
7 |   s ← H × c
8 end
```

The Hamming Weight of the logical `and` between the syndrome and the i-th column of H (i.e. the number of one-value positions in common between them) determines the probability of the i-th positions of the code-word containing an error. If this number is greater then a certain threshold (Line 4 of Algorithm 1), the i-th bit is considered an error and is, consequently, flipped. The syndrome is then recalculated and the procedure restarts until all errors are corrected (i.e. the syndrome = 0). There are several ways of defining the threshold, but most of them are based on the original Gallager's equations [28]. The syndrome should tend to zero over the iterations, but the decoding is a probabilistic procedure and, hence, it has a probability of failure.

Quasi-cyclic Structure

The parity check matrix of LDPC codes can be generated in any arbitrary way that results in a relatively low density. While the use of random matrices is possible, some structures enable much better decoding performance and require less storage space. The quasi-cyclic structure, exemplified below, is a great example of this. For each circulating block (the example has 2), the i-th row is the $(i-1)$-th row rotated one bit to the right. This structure allows the matrices to be represented by its first row only and to be treated as polynomials over $x^r - 1$, where r is the size of each circulating block.

$$\begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

2.3 Arithmetic

In this section, we present the basic arithmetic concepts necessary in this work.

2.3.1 Basic Definitions

Definition 2.3.1. Set: A basic Algebraic Structure which represents an unordered collection of elements.

Definition 2.3.2. Well-defined operation: An operation $\star$ over elements of an algebraic structure A which satisfies the following properties:

- Closure: For all $a, b \in A, a \star b = c$, such that $c \in A$.
- Associativity: For all $a, b, c \in A, (a \star b) \star c = a \star (b \star c)$.
- Identity: For any $a \in A, \exists b \in A$, such that $a \star b = b \star a = a$
- Inverse: For each $a \in A, \exists b \in A$, such that $a \star b = Identity$.

Definition 2.3.3. Group: A set equipped with one well-defined operation.

Definition 2.3.4. Abelian Group: A group in which the well-defined operation satisfies one additional property:

- Commutativity: For all $a, b \in A, a \star b = b \star a$.

Definition 2.3.5. Ring: An abelian group equipped with an additional operation satisfying Associativity and Identity. The additional operation $\cdot$ is *distributive* over the well-defined operation $\star$.

- Distributivity: For all $a, b, c \in A, a \cdot (b \star c) = a \cdot b \star a \cdot c$.

Definition 2.3.6. Field: An abelian group equipped with an additional well-defined operation. One operation is also distributive over the other. Every field is a ring, but the converse is not true since the additional operation of a ring is not required to be well-defined.

Definition 2.3.7. Finite algebraic structure: An algebraic structure (e.g set, group, field or ring) containing a finite number of elements. A finite field is also called a Galois field and is represented by the notation $GF(p)$, where p is a field characteristic.

Definition 2.3.8. Monomial: A product expression between a constant coefficient c and any number of variables x_i. Each x_i can also present an exponent e_i. This work uses only single variable monomials (Equation 2.16). The degree of a single variable monomial is the value of its exponent.

$$m(x) = c \times \prod_i {x_i}^{e_i} \qquad (2.15) \qquad\qquad m(x) = cx^e \qquad (2.16)$$

Definition 2.3.9. Polynomial: A sum of monomials. This work uses only single variable polynomials. The usual notation and some properties are listed below.

$$p(x) = \sum_i c_i \times x^i$$

- Degree = max(i, such that $c_i \neq 0$)
- Notation: $[x^i](p(x)) = c_i$

Definition 2.3.10. Polynomial Ring: A ring composed of polynomials with coefficients belonging to another ring.

2.3.2 Modular Arithmetic

The use of modular arithmetic is an easy way to define operations in a finite algebraic structure. In this work, we will use a finite polynomial ring with coefficients in the $GF(2)$ (finite field of characteristic 2).

Operations in GF(2)

The finite field of characteristic 2 is composed of two elements $\{0, 1\}$ and is equipped with the operations addition and multiplication, which are executed modulus 2.

Examples:

$$0+0 \equiv 0 \ (mod\ 2) \qquad 1+0 \equiv 1 \ (mod\ 2) \qquad 0 \times 0 \equiv 0 \ (mod\ 2) \qquad 1 \times 0 \equiv 0 \ (mod\ 2)$$
$$0+1 \equiv 1 \ (mod\ 2) \qquad 1+1 \equiv 0 \ (mod\ 2) \qquad 0 \times 1 \equiv 0 \ (mod\ 2) \qquad 1 \times 1 \equiv 1 \ (mod\ 2)$$

Operations in the Polynomial Ring

To define the basic operations, we first select a modulus polynomial P.

Addition: The addition between two polynomials is defined as the addition in $GF(2)$ between the coefficients of their monomials with the same degree. It is equivalent to a bit-wise exclusive or (`XOR`) between their binary representations. Examples:

$$(x^2+x)+(x^3+x) \equiv (x^3+x^2) \ (mod\ P) \qquad (x^2+x)+(x^2+x) \equiv (0) \ (mod\ P)$$
$$(x+1)+(x^4+x+1) \equiv (x^4) \ (mod\ P) \qquad (x^3+x^2)+(x+1) \equiv (x^3+x^2+x+1) \ (mod\ P)$$

Multiplication: The multiplication is executed similarly to regular polynomial multiplication. However, whenever the result is greater than the modulus polynomial, it needs to be reduced. The reduction is done by subtracting the modulus polynomial (or multiples of it) from the result. Example, using $P = x^4 + x + 1$:

$$\begin{aligned}
&\textit{Distributive property:} && (x^2+x) \times (x^3+x) = x \times (x^3+x) + x^2 \times (x^3+x) \\
&\textit{First monomial multiplication:} && x \times (x^3+x) = (x^4+x^2) \\
&\textit{Reduction:} && (x^4+x^2) - (x^4+x+1) = (x^2+x+1) \\
&\textit{Second monomial multiplication:} && x^2 \times (x^3+x) = (x^5+x^3) \\
&\textit{Reduction:} && (x^5+x^3) - (x^4+x+1) \times x = (x^3+x^2+x) \\
&\textit{Result:} && (x^2+x+1) + (x^3+x^2+x) = (x^3+1)
\end{aligned}$$

Multiplicative Inversion: From the definition of the inverse property: *For each* $a \in A, \exists b \in A$, *such that* $a \star b = Identity$. The multiplicative inverse operation consists of finding b for some a when $\star$ is the multiplication and the algebraic structure A is the polynomial ring. There are several ways of calculating it. A simple one is using exponentiation. Fermat's Little Theorem states that, for a prime p, $a^p \equiv a \ (mod\ p)$, thus $a^{p-1} \equiv 1 \ (mod\ p) \longrightarrow a^{p-2} \equiv a^{-1} \equiv b \ (mod\ p)$. In this way, we can obtain the inverse by exponentiating a to $(p-2)$. For polynomials, it functions equivalently by exponentiating to $(2^d - 2)$, where d is the polynomial degree. Itoh and Tsujii defined a method to efficiently calculate such exponents in its inversion algorithm [40].

2.4 Code-based Cryptography

In Section 2.2, we presented the basic structure and decoding methods for Hamming Codes and LDPC codes. In both cases, decoding requires the knowledge of specific information, the parity check matrix. In fact, without the knowledge of some structural information such as the parity check matrix, the problem of decoding a general linear code is NP-complete [6]. Starting from this observation, we can define a simple symmetric cryptosystem. The parity check matrix is the shared key, the encryption process is the encoding followed by a deliberated error insertion to mask the message, and the decryption is the decoding process using the parity check matrix. Those who know the parity check matrix will be able to decode the message. Those who do not would have to guess the error or the parity check matrix.

From this basic symmetric cryptosystem, it is easy to define a public key one. As we mention in Section 2.2, any elementary row operations can be performed over the generator matrix. Thus, we can apply to the generator matrix (G) a one-way operation (composed of elementary row operations) that hides the parity check matrix (H) structure. In this way, G can be used as the public key of the cryptosystem, while H remains as the secret key. Note that, to guarantee the security in this scheme, it is also necessary to consider the possibility of an attacker exploiting the knowledge of G to discover H. Details vary according to the code family used, but this is the main idea behind code-based cryptography and it was first presented in 1978 with the McEliece Cryptosystem [47].

2.4.1 The McEliece Cryptosystem

The McElice cryptosystem [47] was the first code-based encryption scheme ever proposed and still remains as the most relevant one. The original scheme used Goppa Codes, which enabled great performance due to very efficient decoding algorithms, but keys took 460Kb at the 80-bit security [10], making the system not competitive among the alternatives.

Equation 2.17 shows the encryption in the original McEliece Cryptosystem: m is a message of length k; z is an error vector with Hamming Weight t; and G' is a $k \times n$ matrix defined in Equation 2.18, where S is a scrambling matrix, G is the generator matrix for the chosen code (e.g. Goppa Code) and P is a permutation matrix. All these matrices are randomly generated and the last 3 compose the private key of the cryptosystem, while

their product G' is the public key. The decryption is shown in Equation 2.19, where *Decode* is the decoding algorithm for the chosen code.

$$c' = mG' + z \tag{2.17}$$

$$G' = SGP \tag{2.18}$$

$$m = Decode(cP^{-1})S^{-1} \tag{2.19}$$

Using Goppa codes at the 80-bit security level, the parameters k, n, and t are chosen respectively as 1632, 1269 and 34 bits, resulting in the 460Kb public key size. Many techniques were proposed in order to reduce the key size of Goppa codes. Misoczki and Barreto [52] proposed a dyadic structure, but although they successfully presented a viable small-key alternative with just 20Kb, it resulted in structural vulnerabilities [25].

In 2000, Monico *et al.* [53] suggested the use of Low-Density Parity-Check (LDPC) codes [28] in the McEliece cryptosystem. At the time, these codes were considered the state-of-the-art on error correction, providing very good error correction capabilities at a low cost. For cryptographic purposes, the code presented compact keys and a reasonably good performance. However, the low density of the parity check matrix, its private key, enabled structural vulnerabilities which resulted in very efficient attacks against the cryptosystem [58].

There were several attempts to solve the structural vulnerabilities of LDPC codes. Most of them were proven to be insecure or resulted in cryptosystems with impractical performance levels and key sizes [3]. In 2002, the proposal of using a parity check matrix with a Quasi-Cyclic (QC) structure for LDPC codes [45] brought great advantages in terms of performance and key size. It did not intend to solve the structural vulnerability problem of LDPC codes, but it created the necessary basis for the development of new families of codes that would avoid such vulnerabilities.

2.4.2 QC-MDPC codes

In 2013, Misoczki *et al.* [51] proposed the use of Quasi-Cyclic Moderate Density Parity Check (QC-MDPC) codes, a derivative of QC-LDPC using higher density parity check matrices. The cryptosystem kept the compact size of keys from LDPC cryptosystems and avoided its structural vulnerabilities by increasing the density of the parity check matrix. Table 2.1 shows a key length comparison between QC-MDPC codes and some of the previous alternatives.

Table 2.1: Key length in bits for different codes (from [51])

Security Level	QC-MDPC	QD-Goppa	Goppa
80	4,801	20,480	460,647
128	9,857	32,768	1,537,536
256	32,771	65,536	7,667,855

Another advantage of QC-MDPC codes is eliminating the need for scrambling and permutation matrices. In the original proposal, the generator matrix G is the row-reduced

echelon form of the parity check matrix H. In this way, the first bits of G are the identity matrix and there is no need to store the matrix of linear transformation since it is not necessary to reverse the transformation after decoding. Considering this, the decryption process boils down to the plain decoding (Algorithm 1) and the original message can be extracted from the first bits of the decoded code-word.

The QC-MDPC codes were proposed using Gallager's decoding algorithm [28], which was originally developed to decode LDPC codes. Soft-decision algorithms were also proposed later [4]. In both cases, the algorithms are iterative and present a non-negligible failure rate in which they are unable to recover the message. Following the notation of Section 2.2, the first version of QC-MDPC cryptosystem used a [9602, 4801, 90]-code capable of correcting 84 errors to achieve 80-bit classical security level. Table 2.2 shows the parameters for a $[2 \times R, R, W]$-code capable of correcting T errors to achieve the respective quantum security level.

Table 2.2: Suggested parameters (from Aragon *et al.* [2])

Quantum Security Level	R	W	T
64 bits	10,163	142	134
96 bits	19,853	206	199
128 bits	32,749	274	264

Niederreiter Cryptosystem: The Niederreiter Cryptosystem [56] is a simpler variation of McEliece where the message is equal to the first half of the error vector. In this way, for QC-MDPC cryptosystems, the first half of the (invalid) code-word is always zero, reducing the size of the block. The error becomes the only secret information stored in the ciphertext and it can be used, for example, as a key for a symmetric cryptosystem. This cryptosystem is implemented by most of the modern implementations since it enables faster encryption and more compact messages and public keys.

2.4.3 QcBits

In 2016, Chou published QcBits [16], a constant time implementation of the Niederreiter cryptosystem using QC-MDPC codes. It was the first fully constant-time implementation of a QC-MDPC based cryptosystem and the fastest at the time. The speed improvement was achieved mostly through the use of bit-slicing techniques for the polynomial arithmetic. QcBits was presented in two versions: the C-only `ref` version, and the `clmul` version using the `PCLMULQDQ` instruction [34] to accelerate polynomial arithmetic. In both versions, the bit-flipping decoding (Algorithm 1) was implemented using constant-time vector rotations and bit-slicing.

Aside from raw performance, the constant-time execution is an important feature since side-channel attacks against implementations of code-based cryptography have been frequently explored in the literature [23, 60, 66]. The decoding algorithm is the most challenging part of the implementation to protect. As shown in Algorithm 1, the original form of the algorithm is inherently variable time because the decoding only stops when

all errors are corrected. To work around this problem, QcBits determines a maximum number of iterations for the decoding (6 at the 80-bit security level). There is no proof or strict estimate indicating that 6 iterations are enough for practical secure use of the implementation, but empirical tests showed an acceptably low failure rate [16].

Key Generation

Algorithm 2 shows the key generation process. The parameters R, W and T are defined by the target security level (Table 2.2), where R is the degree of the modulus polynomial, W is the Hamming weight of the key, and T is the Hamming weight of the error polynomial. The function *GeneratePolynomial* generates a binary polynomial with the specified Hamming weight and maximum degree. The verification in line 4 is necessary to assure that the generated polynomial belongs to the multiplicative polynomial ring, i.e. the polynomial has no factors in common with $x^R - 1$.

As described in Section 2.4.2, the generator matrix G is the row-reduced echelon form of the parity check matrix, H. The probably most intuitive way of obtaining G is through Gaussian Elimination, but there are more simple and efficient methods to calculate it. The goal is to find a matrix E such that $[E] \times [H_0 : H_1] = [I : G]$. Note that $[E] \times [H_0] = [I]$, therefore $E = H_0^{-1}$ and $H_0^{-1} \times H_1 = G$, which is calculated in line 5 of Algorithm 2. In QcBits, the polynomial inversion of H_0 is calculated using Itoh-Tsujii inversion algorithm [40].

Algorithm 2: Key Generation.

Input : *GeneratePolynomial*, R and W
Output: $PrivateKey$ and $PublicKey$

1 **repeat**
2 $\quad H_0 \leftarrow GeneratePolynomial(MaxDegree = R - 1, HammingWeight = \frac{W}{2})$
3 $\quad H_1 \leftarrow GeneratePolynomial(MaxDegree = R - 1, HammingWeight = \frac{W}{2})$
4 **until** $H_0^{-1} \times H_0 \equiv 1 \ (mod\ x^R - 1)$;
5 $G \leftarrow H_0^{-1} \times H_1$
6 $PublicKey \leftarrow G$
7 $PrivateKey \leftarrow (H_0, H_1)$

Encryption

The encryption process (Algorithm 3) is structured to be part of a Key Encapsulation Mechanism (KEM). The error polynomials e_0 and e_1 are randomly generated and can be used as a key for a symmetric cryptosystem. The input G is the public key and the other inputs are the same of the key generation. The encryption could also be used to encrypt an arbitrary message. In this case, the polynomials e_i would be part of the input and the message would need to be encoded as a binary polynomial complying with the restrictions of maximum degree and Hamming weight.

Algorithm 3: Encryption.

Input : G, R, T and $GeneratePolynomial$
Output: $Ciphertext$ and Key

$e_0 \leftarrow GeneratePolynomial(MaxDegree = R - 1, HammingWeight = \frac{T}{2})$
$e_1 \leftarrow GeneratePolynomial(MaxDegree = R - 1, HammingWeight = \frac{T}{2})$
$Ciphertext \leftarrow e_1 \times G + e_0$
$Key \leftarrow (e_0, e_1)$

Decryption

Algorithm 4 shows the polynomial view of the decryption process. The function *TransposePolynomial* obtains the polynomial representing the column for a Quasi-Cyclic matrix from the polynomial representing the row. The polynomial *sum* is an integer polynomial and the function $IntegerPolynomialAddition$ interprets w as an integer polynomial and adds it to *sum*. The function $CalculateThreshold$ calculates the threshold used to define which bits probably belong to the error polynomials. The method used to determine it varies with the implementation and QcBits uses fixed pre-calculated values.

Algorithm 4: Polynomial view of the decryption using the bit-flipping algorithm.

Input : H and c
Output: e_0 and e_1

$e_0 \leftarrow 0,\ e_1 \leftarrow 0$
$H'_0(x) \leftarrow TransposePolynomial(H_0(x))$, $H'_1(x) \leftarrow TransposePolynomial(H_1(x))$
$s \leftarrow (H_0 \times (e_0 + c)) + (H_1 \times e_1)$
while $s \neq 0$ **do**
  for $j = 0 \rightarrow 1$ **do**
    $sum \leftarrow 0$
    foreach *monomial* $x^i \in H'_j(x)$ **do**
      $w \leftarrow s \times x^i$
      $sum \leftarrow IntegerPolynomialAddition(sum, w)$
    end
    $Threshold \leftarrow CalculateThreshold(s)$
    foreach *monomial* $x^i \in sum(x)$ **do**
      if $[x^i](sum(x)) > Threshold$ **then**
        $e_j \leftarrow e_j + x^i$
      end
    end
  end
  $s \leftarrow (H_0 \times (e_0 + c)) + (H_1 \times e_1)$
end

Algorithm 5 shows the constant-time implementation of each decoding iteration in QcBits. The value TH is the iteration threshold, s is the syndrome, c is the ciphertext and H' the sparse representation of the parity check matrix, which is an array of non-zero indices. The *BitSliceAdder* function consists in adding each bit individually by

positioning and storing each bit of the result in an array position (Algorithm 6), similarly to a half adder circuit. The *BitSliceSubtractor* is implemented in the same way, but with a full adder or subtractor instead.

Algorithm 5: QcBits Bit-flipping implementation logic

Input : H', c, s and TH
Output: c

1 $N \leftarrow 1 + \lceil log_2(|H'|) \rceil$
2 $sum[N] \leftarrow 0's$
3 **foreach** *index* i ***in*** H' **do**
4 $\quad w \leftarrow s \lll i$
5 $\quad sum \leftarrow BitSliceAdder(sum, w)$
6 **end**
7 $sum \leftarrow BitSliceSubtractor(sum, TH)$
8 $c \leftarrow \neg sum[N-1] \oplus c$

Algorithm 6: BitSlice Adder Implementation Logic

Input : N, sum and w
Output: sum

1 **for** $i = 0$ ***to*** N **do**
2 $\quad c_{out} \leftarrow sum[i] \wedge w$
3 $\quad sum[i] \leftarrow sum[i] \oplus w$
4 $\quad w \leftarrow c_{out}$
5 **end**

Line 1 in Algorithm 5 calculates the number of bits necessary to represent the number of elements belonging to H', which is the maximum result that can be stored on the *sum* array by the *BitSliceAdder*. Line 2 initializes *sum* with zeros. The loop on line 3 iterates over the private key indices: for each index, the syndrome is rotated left on the index value (line 4) and the result is added to the *sum* array using the *BitSliceAdder* function. This process is equivalent to calculating the Hamming Weight of the bitwise `AND` between each matrix column and the syndrome. However, for 80-bit security, instead of iterating over the 4801 rows of the parity check matrix, this method just needs to iterate over the 90 indices of the sparse matrix representation. At the end of the loop, the threshold is subtracted from the sum of each bit. If the most significant result bit is one on line 8, it indicates that the threshold is greater than the sum and the corresponding bit must not be flipped. Otherwise, the bit is flipped.

2.4.4 BIKE

In 2017, Aragon *et al.* published the BIKE suite [2] containing 3 key encapsulation schemes using QC-MDPC codes. BIKE is the main representative of QC-MDPC codes in NIST's standardization project [57] and its Variation 2 implements the same cryptosystem as QcBits.

Comparing with the original QcBits, BIKE presents several improvements. It uses updated parameters to achieve up to 128-bit quantum security level; it features an enhanced version of the bit-flipping algorithm (Algorithm 7); it exploits the Montgomery Trick to perform batch key generations (Algorithm 9); it present versions using modern instruction sets, such as Intel AVX-512; among other minor improvements. BIKE, however, does not present a fully constant time version, which could be considered a problem from the side-channel protection perspective.

Algorithm 7 shows the bit-flipping decode variant presented in BIKE. H is the parity check matrix, s is the syndrome, and W is a security parameter(Table 2.2). S, δ and the *threshold* function are additional parameters and function with values depending on the security level and on the BIKE variant being implemented.

2.5 Side-Channel Protection

Side-channel attacks are those that exploit the possible correlation between the cryptosystem secret data (keys and plain-text) and the physical behavior of the hosting machine during its execution. Examples of physical data that can be gathered to perform a side-channel attack are power consumption, execution time, heat emanation, and sound emission. In a side-channel attack, the information leakage occurs accidentally as a characteristic of the implementation [29]. Although most of them require physical access to the host machine, some can be entirely executed through remote connections. Timing side-channel attacks are an example of side-channel attacks that can usually be executed in remote ways [13] and, hence, protection against them became a requirement in modern cryptographic implementations. Protections against more intrusive data collection are also important depending on the context.

Both conventional and post-quantum cryptography algorithms are a target for side-channel attacks. Considering conventional algorithms, there are, for example, attacks against the AES, exploiting the cache memory behavior [7]; time-based attacks against Diffie-Hellman and RSA [43], and fault-based attacks against elliptic curve cryptography [17]. In the post-quantum cryptography field, can be cited, for example, attacks against code-based cryptography [66, 23].

The implementation of QcBits is fully constant time in order to protect the implementation against timing side-channel attacks. However, it was demonstrated that the power consumption of original implementation execution depends on the secret key. Rossi *et al.* [60] presented a power-based side-channel attack against the syndrome calculation of QcBits. The attack exploited a power-leakage at the store of the rotated code-word (line 4 of Algorithm 5). They also provided a simple countermeasure in order to prevent the attack. Another power side-channel vulnerability concerning QcBits is the conditional copy implementation used along the code, which will be further discussed in Section 3.3.

2.5.1 Constant-time implementations

The use of constant-time implementations is the main countermeasure used to avoid timing side-channel attacks. The definition of constant-time implementation, however, varies in the literature. Strictly speaking, a constant-time implementation is one whose execution always take the same amount of time. While sometimes used, this definition could only be achieved considering an execution on an ideal machine. On real machines, small variations in the execution time occur naturally due to the physical properties of hardware. Modern architectures also present features that are out of the application's control and that can significantly impact the execution time, such as the dynamic voltage and frequency scaling [67]. Some less strict but achievable definitions are presented below. In these cases, the variations in the execution time caused by the execution environment are disregarded.

1. The execution time does not depend on any data being processed, except for data that is public by construction (e.g. the public key, the security parameters, and the length of an arbitrary message). This is the definition we adopt in this work.

2. The execution time does not depend on secret data being processed, but it may vary depending on derivatives of secret data, such as data resultant of masking or blinding techniques.

3. The execution time depends on secret data being processed, but there are no known attacks capable of recovering any significant information about secret data through the observation of execution time. While this definition arguably results in similar protection against currently known timing side-channel attacks, it lacks guarantees against possible future attacks.

A more relaxed variation of this concept is a *uniform implementation*: given an iterative algorithm with the number of iterations depending on secret data, a uniform implementation executes each iteration in constant time, but the number of executed iterations is variable. If the number of iterations is not sufficient to obtain any significant information about secret data, then this definition fits the definition of constant-time implementation present on Item 3. The decoding process of BIKE is an example of Uniform implementation that fits the definition on Item 3.

Implementing conditional statements in constant-time

The implementation of conditional statements in constant-time is fundamental to the implementation of algorithms presented in this work. In this section, we present a brief example of how we implemented them. Generally, we execute all possible execution flows and select the correct result using constant-time conditional copies. These conditional copies, in turn, are implemented in constant-time through specialized instructions, such as Intel's `CMOV`, `BLENDV` or AVX-512 masked instruction; or through the explicit use of masks with a bit-wise `AND` operation.

Listing 1 shows an example of a non-constant-time conditional operation. Assuming that `A` and `B` are secret data, this implementation is vulnerable to timing side-channel

attacks. Listing 2 shows the equivalent constant-time implementation (considering that `Function1` and `Function2` do not have side effects). The operations using 64-bit integers (`uint64_t`) had their results conditionally selected through a bit-wise `AND` with the mask `cond`. When using AVX-512 registers, the implementation of conditional operations is significantly simplified. The AVX-512 instruction set extension already provides masked versions for most of its instructions. In this way, we simply use the mask `cond` in the mask field of the intrinsics of these instructions.

```
uint64_t A, B, C, D, cond;
__m512i V1, V2, V3;

[...]

if(A < B){
    C = Function1();
    D += 5;
    V1 -= V2;
}else{
    C = Function2();
    D ^= 0xf;
    V1 &= V3;
}
```

Listing 1: Non-constant-time conditional operations

```
cond = ((int64_t) (A - B)) >> (63);

C = cond & Function1() | ~cond &  Function2();
D = cond & (D + 5) | ~cond & (D ^ 0xf);

V1 = _mm512_mask_sub_epi64(V1, cond, V1, V2);
V1 = _mm512_mask_and_epi64(V1, ~cond, V1, V3);
```

Listing 2: Constant-time conditional operations

2.5.2 Reaction Attack

The current implementations of QC-MDPC code-based cryptography rely on imperfect decoding processes that present a non-negligible failure rate in which they are unable to

remove the errors. The implementations usually deal with this Decoding Failure Rate (DFR) by establishing an upper bound and adjusting the decoding parameters accordingly. It is commonly accepted that the DFR should be at most 10^{-7} [5]. QcBits uses a DFR of 10^{-8} and 10^{-5} for the 40-bit and 64-bit quantum security level, respectively. BIKE defined the upper bound of 10^{-7} for the DFR for 64-bit, 96-bit, and 128-bit quantum security level implementations. Considering QC-LDPC, we can refer to the cryptanalysis work by Fabšič *et al.* [24] that estimated the DFR of recent implementations to be around 10^{-5} for a 40-bit quantum security level.

The DFR used to be an issue only from the usability perspective, but it also became a security issue with the publication of the Reaction Attack [37]. It was first discovered in 2016, when Guo *et al.* [37] presented an attack exploiting the relation between the parity check matrix bits and the decoding failure rate of QC-MDPC codes. Later, Fabšič *et al.* [24] showed that it also works for QC-LDPC codes and even with the use of a soft-decision decoding algorithm [44] instead of bit flipping. The attack was named Reaction Attack and is capable of recovering the entire secret key once provided with a large number of decoding attempts. It works as follows:

1. Given a group ψ_D containing all possible error vectors, such that each element is composed of $\frac{t}{2}$ pairs of bits with a distance D between them:
   (a) Send M messages encrypted with error vectors belonging to ψ_D to the decryption process.
   (b) Based on the number of failures, calculate the failure rate for the distance D.
2. Repeat step 1 for all possible values of D.
3. Based on failure rates, for each D, determine if there are two bits at the distance D of each other in the private key. In a simple way, this relation can be defined as: the higher the failure rate, the lower the number (multiplicity) of pairs of bits with distance D between them in the private key.
4. Reconstruct the private key from this distance spectrum.

The main countermeasure being applied to avoid this attack is the use of ephemeral keys [5]. More recent implementations also featured methods for obtaining a DFR negligible in the security level [23], but they did not achieve practical levels of performance and message length yet.

2.6 Summary

Chapter 2 presented the concepts necessary for the understanding of this work. Most of them regard basic notions of cryptography and error correcting codes, but some are more context-specific, such as in Section 2.4.3, where the QcBits implementation is presented.

The Chapter started by introducing symmetric, public-key, and post-quantum cryptography in Section 2.1. Section 2.2 presented a basic background on error-correcting

codes, focusing specifically on parity check codes. Section 2.3 developed the arithmetic concepts from the definition of *set* up to the definitions of *polynomial, field* and *ring*, which are used in this work. Section 2.4 included an introduction to code-based cryptography, but it focused mainly on presenting QcBits as an example of modern implementation of QC-MDPC code-based cryptosystem. Finally, Section 2.5 first introduced side-channel attacks in a generic way and then presented some more specific cases in which QC-MDPC code-based cryptosystems are affected.

Chapters 3 and 4 will present our contributions to the performance improvement of QC-MDPC cryptosystems. These contributions are built upon QcBits, therefore the understanding of its functioning (Section 2.4) will be necessary, as well as the basic concepts of arithmetic over binary polynomials (Section 2.3).

Algorithm 7: BIKE's One-Round Bit Flipping Algorithm [2]

Input : H, S, W, δ and s
Output: e

$T \leftarrow threshold(s)$
$J \leftarrow [[]]$; // Empty array of arrays
foreach *column* h_i ***in*** H **do**
  $l \leftarrow min(HammingWeight(h_i \wedge s), T)$
  $J_l \leftarrow J_l \cup h_i$
end
$e \leftarrow J_T$
$s' \leftarrow s - eH^T$
while $HammingWeight(s') > S$ **do**
  for $l = 0$ ***to*** δ **do**
    $e' \leftarrow 0$
    for $h_i \in J_{T-l}$ **do**
      if $HammingWeight(h_i \wedge s) \geq W/4$ **then**
        $e'_i \leftarrow 1$
      end
    end
    $(e, s') \leftarrow (e + e', s' - e'H^T)$
  end
end
$e' \leftarrow 0$
for $i = 0$ ***to*** $length(e)$ **do**
  if $e_i = 1$ ***and*** $HammingWeight(h_i \wedge s) \geq W/4$ **then**
    $e'_i \leftarrow 1$
  end
end
$(e, s') \leftarrow (e + e', s' - e'H^T)$
while $HammingWeight(s') > u$ **do**
  $j \leftarrow -1$
  while $j = -1$ **do**
    $r \leftarrow RandomValueFrom(\{x \in [0, length(s)) \mid s_x = 1\})$
    foreach *column* h_i ***in*** $\{H \mid H_{i,j} = 1\}$ **do**
      if $HammingWeight(h_i \wedge s) \geq W/4$ **then**
        $j \leftarrow i$
      end
    end
  end
  $(e, s') \leftarrow (e_j + 1, s' + h_j)$
end
return e

Chapter 3

Accelerating the implementation of QcBits

The original QcBits presents a very good performance level due to the employed techniques and some of the algorithm choices. However, it does not exploit modern instruction set extensions which could improve the performance even further. Moreover, it was published using outdated parameters that achieves only 40-bit quantum security level. Considering this in this chapter, we present several contributions towards a performance improvement of QcBits, as well as an update on its parameters.

First, we focused on improving the performance of the decoding process in 40-bit QcBits and presented the following contributions.

- An optimization of the decoding process for both versions of QcBits, achieving a speedup of 3.6 times over the `clmul` version and 4.8 times over `ref`. The performance improvement came from vectorization using AVX instructions, loop unrolling on hot spots and pre-calculation of vector rotations.
- An estimation that performance gains could be as high as 5.06 times on `clmul` version if new instructions for conditional vectorial moves and 256-bit register shifts were added to the architecture.
- The mitigation of all known power vulnerabilities found in the original implementation with an almost negligible ($< 1\%$) impact on the overall performance.

These contributions were published at WSCAD-2017 [35]. Then, we extended this work by rewriting the entire implementation optimizing it through both algorithmic and implementation techniques. Our newer contributions are listed below.

- The update of the security level from 80-bit classical security level to 128-bit quantum security level, meeting NIST's highest security level requirement for the standardization process.
- The vectorization of the entire implementation using the AVX512 instruction set extension.
- The replacement of some of the core algorithms with others that have a better performance in face of the new AVX512 instructions.

- The implementation of BIKE's batch key generation processes using QcBits' algorithms.

As a result of the above-mentioned contributions, our implementation takes 928, 259 and 9,803 thousand Skylake cycles to respectively perform the batch key generation (cost per key), encryption, and decryption. These times were measured using our constant-time implementation, which prevents time-based side-channel attacks [23]. Using a uniform implementation, our decryption takes 5,008 thousand cycles.

Comparing to BIKE, we have a 1.91-factor speedup on the uniform decryption and a 1.34-factor speedup on the constant-time encryption. Our constant-time key generation is 3 times slower than BIKE's since we choose to uphold the constant-time execution. BIKE does not provide constant time implementations for key generation and decryption [33]. Its decryption is uniform, its encryption is constant time and its key generation relies on the NTL Library [63], which is neither constant-time nor uniform.

3.1 Optimizing the decoding process of QcBits

We began our optimization of the decoding process of the original QcBits by extending the vectorization to the whole code using the SSE4, AVX2 and AVX512 instruction set extensions. Our initial expectation was to obtain a 2, 4 and 8 factor speedup in each of the cases, respectively, since these values correspond to the number of SIMD lanes found on these standards. Most of the code was composed of bitwise operations, such as `XOR` and `AND` of the bit slice adder, and were easily vectorizable, resulting in an immediate gain of 2.6 times when using the AVX2 instruction set, for example. However, the absence of some instructions in the SIMD instruction sets prevented the expectation from materializing.

The main obstacle for vectorization was the implementation of shifts on registers larger than 64 bits. These operations are necessary to perform the vector rotations shown on line 4 of Algorithm 5. For the 80-bit security level, the rotation target has 4,801 bits and it is implemented in two steps using C language: first, the words that compose the vector are permuted following the rotation logic; next, the rotation is done inside each word, shifting its bits and inserting next word bits in the shifted area. For registers with size smaller or equal to 64 bits, there is a single instruction to shift all the register bits, which facilitates the implementation. For larger registers, bit shift instructions operate over the 64-bit lanes only. Hence, we had to perform a custom multi-instruction logic, making the implementation slower and more complex.

Listing 3 and 4 show our implementation of a shift right with carry on AVX2 and AVX512 registers, respectively, used in the vector rotation shown on line 4 of Algorithm 5. The code is composed of 10 intrinsics for vector instructions. It works by permuting 64-bit sets to reduce the shift amount to less than 64, then the Carry In is inserted using the `BLENDV` instruction and the shift is finished using instructions that shift inside the 64-bit lanes. Some of the used instructions are very expensive. For example, the `PERMUTEVAR` instruction on line 12 and 19, which has 3-cycle latency in Skylake, according to Agner Fog's instruction tables [27].

```
word_t bitShiftRight256bitCarry (word_t data, index_t count, word_t * carryOut, word_t carryIn){
  word_t innerCarry, out, countVet;
  word_t idx = _mm256_set_epi32(0x7, 0x6, 0x5, 0x4, 0x3, 0x2, 0x1, 0x0);
  const word_t zeroMask = _mm256_set_epi64x(-1, -1, -1, 0);
  word_t zeroMask2 = _mm256_set_epi8(0x80, 0x80, 0x80, 0x80, 0x80, 0x80, 0x80, 0x80,
                                     0x82, 0x82, 0x82, 0x82, 0x82, 0x82, 0x82, 0x82,
                                     0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x84,
                                     0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x86);

  countVet = _mm256_set1_epi8((count >> 5) & 0xE);
  idx = _mm256_add_epi8(idx, countVet);
  data = _mm256_permutevar8x32_epi32(data, idx); // rotate
  *carryOut = data;
  zeroMask2 = _mm256_sub_epi8(zeroMask2, countVet);
  data = _mm256_blendv_epi8 (carryIn, data, zeroMask2);
  // shift less than 64
  count = (count & 0x3F);
  innerCarry = _mm256_blendv_epi8(carryIn, data, zeroMask);
  innerCarry = _mm256_permute4x64_epi64(innerCarry, 0x39); // >> 64
  innerCarry = _mm256_slli_epi64 (innerCarry, 64 - count);
  out = _mm256_srli_epi64 (data, count);
  out = _mm256_or_si256 (out, innerCarry);
  return out;
}
```

Listing 3: 256-bit register shift implementation

For the `clmul` version vectorized with AVX2 instruction, the syndrome calculation was also a problem. Executed at the beginning of the decoding process, it was originally implemented using the carry-less multiplication instruction which is only available for 128-bit size registers. Therefore, this code snippet, which takes approximately 20% of the code execution time, is stuck at the 128-bit implementation.

3.1.1 Basic Vectorization Results

We compiled the implementations using the three industry-standard compilers: GCC 7.3.1, CLANG 8.0.0 and ICC 18.0.3. For all the compilers, the compilation optimization flags used were `-O3` and `-march=native`. The flag `-funroll-all-loops` was also used when compiling with GCC. Equivalent flags for aggressive loop unrolling on the other compilers were tested, but they did not result in any performance improvement and therefore were removed. The implementations were executed on two machines: the first one, named Haswell, uses an Intel i7-4770 processor and the second, named Skylake, uses an Intel i7-7820X processor. Both machines run the Fedora operating system and, aiming at experiment reproducibility and cycle accuracy, had the Intel Turbo Boost and Hyper-Threading mechanisms disabled [8]. We measured the number of cycles using the Intel `RDTSC` instruction and reproduced each experiment 10 thousand times, which enabled us to achieve a 99% confidence interval that is negligible compared to the average ($\ll 1$)%, hence we will omit it from the charts.

The performance results of this first vectorization are shown in the chart of Figure 3.1. As can be noted, the execution time, considering the compilation with GCC, reduced from 1,306,618 Skylake cycles and 1,399,015 Haswell cycles to, respectively, 791,829 and 898,996 cycles when using the SSE instruction set, which represents a speedup of 1.6 times; and to 500,146 and 656,294 cycles when using the AVX2 instruction set, which in turn represents

```
word_t bitShiftRight512bitCarry (word_t data, index_t count, word_t * carryOut, word_t carryIn){
  word_t innerCarry, out, countVet, idx, idx1;
  idx = _mm512_set_epi32(0xf, 0xe, 0xd, 0xc, 0xb, 0xa, 0x9, 0x8, 0x7, 0x6, 0x5, 0x4, 0x3, 0x2, 0x1, 0x0);
  countVet = _mm512_set1_epi8((count >> 5) & 0xE);
  idx1 = _mm512_add_epi32(idx, countVet);
  data = _mm512_permutexvar_epi32(idx1, data);
  *carryOut = data;
  data = _mm512_mask_blend_epi32(0xFFFF >> ((count >> 5) & 0xe), carryIn, data);
  // shift less than 64
  count = (count & 0x3F);
  innerCarry = _mm512_mask_blend_epi64(0xFE, carryIn, data);
  innerCarry = _mm512_alignr_epi64 (innerCarry, innerCarry, 1);
  innerCarry = _mm512_slli_epi64 (innerCarry, 64 - count);
  out = _mm512_srli_epi64 (data, count);
  out = _mm512_or_si512 (out, innerCarry);
  return out;
}
```

Listing 4: 512-bit register shift implementation

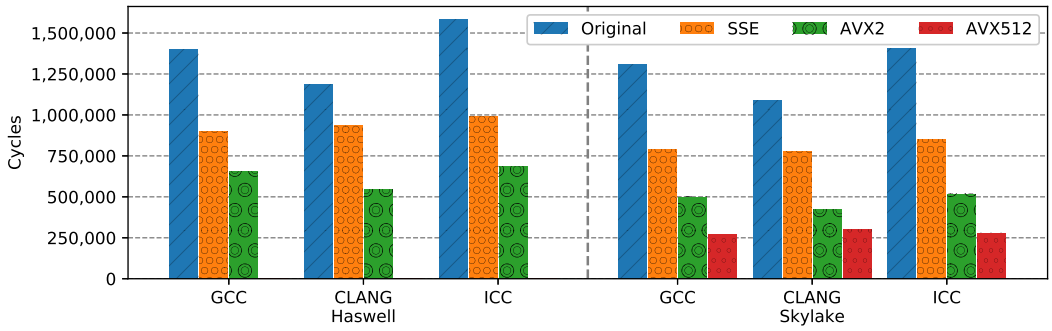


Figure 3.1: Initial vectorization results

speedups of 2.6 and 2.1 times, also respectively. Using the AVX512 instructions, the speedup was 4.76 times, reducing the execution time to 274,423 cycles. The chart also shows the performance improvement between the two processors generations, especially for the vectorized versions: The Skylake processor is 10% faster than the Haswell processor on the original 64-bit version and on the SSE version, while, for the AVX2 version, Skylake is 24% faster than Haswell. These conclusions are based on the average results obtained with the three compilers. The Haswell architecture does not present AVX512 instructions.

3.1.2 Vector Rotation Table

Although there is a likely more efficient implementation for Listing 3, it will probably be always inefficient without special hardware support. Instead of trying to optimize further our implementation, we focused on reducing the number of shift operations executed. The word permutation of the vector rotation, which is shown on line 4 of Algorithm 5 and is composed of conditional copies and register shifts, represented almost 40% of the code execution time and 90 of them were calculated in the decoding implementation, one for each parity check matrix index. However, the permutation is done based on the first bits of each index and, using 256-bit registers and considering the 80-bit for the security

level, there are only 32 possible permutations of words following the rotation logic.

Considering that, we construct a table of all possible word rotations at the beginning of the decoding process and just query that table instead of calculating the permutations every time. The chart in Figure 3.2 shows the correlation between the number of word rotations that were calculated and the number of possible rotations for each word size. As can be seen, the pre-calculated table of rotations would not be worth for the original 64-bit, but it is faster for all our optimized versions.

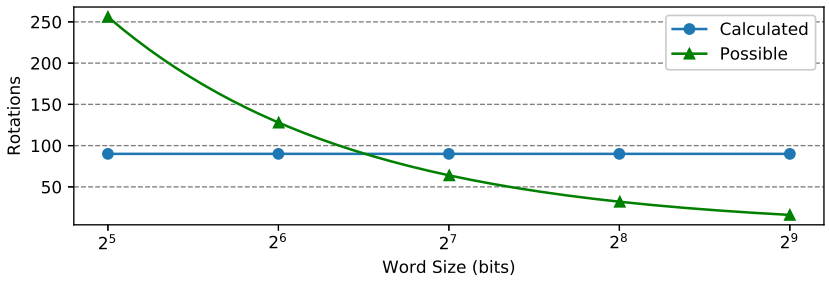


Figure 3.2: Number of word rotations computed and possible for each implementation

This approach, however, has some obstacles to be used in a constant time implementation. The table access pattern cannot depend on the private key because it would leak cache-timing information that could be exploited on a side-channel attack [66]. In order to prevent this leak, we implemented the code so that it iterates over all the table elements conditionally copying each one of them. These extra memory accesses add a great performance penalty and the table of rotation alone became slower than the calculations even on 256-bit registers version.

Despite that, we were able to improve the rotation table by doing a trade-off between the calculation and the table access. Basically, we construct a table with just a small subset of the possible rotations. Then, when a rotation is needed, the implementation iterates over the table, picks the nearest rotation and calculates the pending rotation amount starting from the pre-calculated value. Since the rotation calculation is done based on each bit of the rotation amount, its performance is proportional to the logarithm of the maximum rotation amount. This way, we achieved a 1.24-factor speedup on the AVX version, when comparing to the basic vectorization time, using tables with 3 stored rotations only. The number of Skylake cycles, when compiling with GCC, was further reduced from 500,146 cycles to 404,883 cycles and the overall speedup increase from 2.6 times to 3.2 times. The use of the rotation table also drastically reduced the number of iterations necessary to calculate the rotations. This reduction allowed a manual loop unrolling which leads to a 1.22-factor speedup over the best time, bringing down the number of Skylake cycles when compiling with GCC to 332,974 cycles.

As shown in Figure 3.2, the number of possible rotations when using 512-bit words is only 16. In this way, for the AVX512 implementation, we pre-calculated and stored all the rotations. The number of cycles reduced from 274,423 to 208,989 resulting in an overall speedup of 6.25 times. All the presented optimization techniques were also applied to QcBits `ref` version, which uses only C code. Table 3.1 shows the results for all versions. The speedups relatively to the Original Version execution are shown in Figure 3.3.

Table 3.1: Final optimization results (in cycles)

Machine	Version	Compiler	Original	SSE	AVX2	AVX512
Skylake	CLMUL	GCC	1,306,591	535,232	332,974	208,989
		CLANG	1,079,710	512,913	301,371	242,781
		ICC	1,406,311	724,027	391,451	225,212
	REF	GCC	2,144,799	834,374	504,126	203,508
		CLANG	1,450,582	758,856	387,434	281,339
		ICC	2,175,998	1,135,434	552,619	238,781
Haswell	CLMUL	GCC	1,420,750	767,157	508,900	-
		CLANG	1,158,520	745,768	466,732	-
		ICC	1,582,715	859,520	525,901	-
	REF	GCC	2,190,229	1,101,205	660,981	-
		CLANG	1,566,766	1,074,068	559,744	-
		ICC	2,309,771	1,239,992	678,288	-

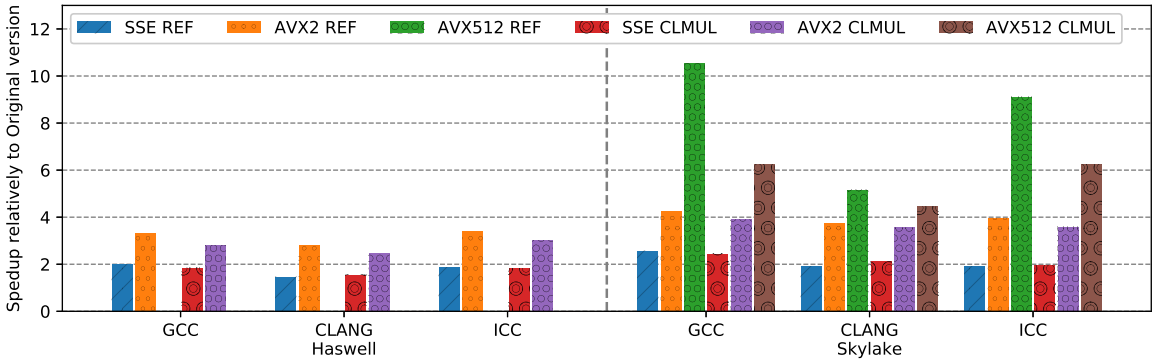


Figure 3.3: Final speedups achieved with the optimization (relatively to the corresponding Original version execution)

Analyzing the `clmul` version results in the chart, we can note that the results using SSE instructions overcome our initial expectation with a speedup of 2.44 times on Skylake with GCC. This was possible thanks to the rotation table and the loop unrolling, previously explained. For the version vectorized with AVX2 instructions, however, the speedup is still below 4 times, being 3.92 times with GCC on Skylake. It happens mostly because of the absence of a 256-bit `clmul` instruction, which creates the need for the use of 128-bit register instructions in the syndrome calculation. This hurts the performance not only because of the use of small registers but also due to the transition between the instruction set extensions, which is known to be expensive [46].

The `ref` version uses the same constant-time rotation process shown in Algorithm 5 to calculate the matrix multiplications. The only modification in the algorithm is that the *BitSliceAdder* is replaced by a simple `XOR`. Once this method does not rely on `clmul` instruction, the `ref` version could be better optimized. The execution time using AVX2 instructions was 504,126 cycles while using AVX512 instructions it was reduced to 203,508 cycles, which was the best time achieved. The speedups were 4.25 and 10.54 times, respectively.

3.1.3 Potential gains with new instructions

As explained at the beginning Section 3.1, the two main hindrances for the vectorization were the absence of vector instructions for register shifts and conditional moves. This procedure is currently done by the `BLENDV` instruction, which is much more powerful and, hence, expensive than we need for this purpose. Although Fog [27] reports a reciprocal throughput of 1 cycle for this instruction, it is difficult to use the instruction in a sequential way to achieve this performance. The introduction of AVX512 instructions solved this problem by allowing conditional moves at a very low cost. However, the absence of shift instructions for register larger than 64 bits is still a major problem for all our vectorized versions.

In order to estimate the possible gains if the shift instruction exists, we experimented with the `clmul` version to suppose its existence. The experiment was done by replacing the vector shift algorithm by a simple 64-bit lanes shift. This version, of course, does not result in the correct output, but it serves as an estimation. Testing the AVX512 version on Skylake and compiling with GCC, we execute it in 185,843 cycles, which represents a 1.14 times speedup compared to our best correct `clmul` version and a total speedup of 7.03 times.

3.2 Enhanced version of QcBits

During the optimization process of the decoding, we held the 80-bit classical security level since the only source of comparison against our results was the original implementation of QcBits. However, when we started to optimize other snippets of the code, new implementations were published using the recommended 128-bit quantum security level. In this way, we decided to rewrite the entire implementation of QcBits, updating the security level, optimizing all its processes and reapplying all our previously developed techniques. Following the trend in other optimized post-quantum implementations, we also decided to use the AVX512 instruction set extension.

Table 3.2 compares the parameters used in the implementation to achieve the 128-bit quantum security level and the ones used by the original implementation. Similarly to Table 2.2, R is the size of the polynomials, and W and T are the Hamming Weight of the secret key and the error vector, respectively. We represented each 32,749-bit size polynomial as 64 words of 512 bits each.

Table 3.2: Parameters of each implementation. Obtained from QcBits [16] and BIKE [2]

	R (bits)	W	T	Security Level
Original QcBits	4,801	90	84	80-bit classical
This Version	32,749	274	264	128-bit quantum

We executed our performance tests in the same Skylake machine used in Section 3.1. The number of cycles was measured using the Intel `RDTSC` instruction and each result is the average of 10,000 measurements, which we analyzed and concluded that the greatest majority of the samples are very close to the average. The standard deviation and a

99% confidence interval were calculated from the data. For the key generation processes and the constant-time decryption, both the values are negligible compared to the average ($\ll 1\%$) and, therefore, were omitted in the results. For the encryption, due to the time variance of the random polynomial generation, and for the uniform decryption, whose the execution time is intrinsically variable, the standard deviation is about 10% while the 99% confidence interval is also negligible and, therefore, was also omitted.

In order to have a fair comparison, we also measured the times for BIKE on the same machine. We used *Additional Implementation* of BIKE[1], which is an optimized version using AVX512 instructions and several code snippets in assembly language; and the version 11.3.0 of the NTL library [63].

3.2.1 Random Polynomial Generation

Both the key generation and the encryption process need randomly generated polynomials with a specific Hamming Weight. Since the generation procedure is not critical in terms of performance, we implemented the same algorithm presented by BIKE, without investigating possible optimizations. The strategy is to randomly generate one index of the polynomial at the time: if the index is not repeated, we add it to the polynomial; otherwise, we discard it. To guarantee the constant time execution, we can establish the number of indexes that we need to generate to obtain a certain number of non-repeated indexes with high probability. Drucker and Gueron [22], when defining this polynomial generation method, demonstrated that generating $2 \times N$ indexes is sufficient to achieve, with high probability, N valid indexes. We followed this definition in our implementation.

For the random generation of the indexes, we used the `RDRAND` instruction to generate a 256-bit value and expanded it using the OpenSSL EVP Library (version 1.1.1-pre2 alpha) and its implementation of the SHAKE256 algorithm. For older versions of OpenSSL, which do not include SHAKE256, we generated all the bytes using the `RDRAND` instruction.

3.2.2 Key Generation

The key generation process of QcBits (Algorithm 2), in terms of implementation, can be resumed to 3 steps:

1. The random generation of two polynomials with a specific Hamming Weight.
2. A binary polynomial inversion.
3. Two binary polynomial multiplications.

The second step is the most expensive in the process. While it takes dozens of millions of cycles to be executed (99.4% of the time), the multiplication takes just a little over one hundred thousand cycles and the random generation only takes a few tens of thousand cycles. Therefore, we focused mainly on it in our optimization.

The original implementation used the Itoh-Tsujii Algorithm [40] for the polynomial inversion. This algorithm works by exponentiating the polynomial to $2^R - 2$, where R is

[1] Available on the official website. Version dated 05/23/2018.

the degree of the modulus polynomial. The exponentiation is carried out as a sequence of squares and multiplications following an addition chain. The degree of the modulus polynomial, which defines the addition chain, is public information. Therefore, to be implemented in constant time, the algorithm only requires constant-time implementations of multiplications and squarings over binary polynomials. It is simple to implement these operations in constant-time, but they rely on the carry-less multiplication instruction for efficiency. As mentioned in Section 3.1, this instruction is limited to 128-bit registers and its performance was already a problem in the AVX2. The problem intensifies when using the AVX512 instruction set and our best alternative was to replace the algorithm.

The Wu *et al.* inversion algorithm, shown in Algorithm 8, is a modified version of Brunner Algorithm [14] designed to be a hardware implementation. For this reason, it is relatively easy to implement in software in a constant time way. The primary function of the algorithm is to calculate the division between two polynomials u and f modulus the polynomial $g = s$. Since we need the inverted value multiple times (lines 4 and 5 of Algorithm 2), we use $u = 1$ to just calculate the inverse of $f = H_0$. The algorithm is composed of polynomial additions (lines 5 and 8) and polynomial divisions per x (line 11). The sum of binary polynomials is just a `XOR` between them, while the divisions per x are just a rotation of 1 bit to the right in the polynomial representation (considering g in the format $x^r - 1$). These operations have a great performance improvement when vectorized, with gains almost reaching the increment of the number of SIMD lanes.

Algorithm 8: Wu *et al.* Inversion Algorithm [68]

Input : f, s and u
Output: $v = \frac{u}{f}\ (mod\ s)$

```
1  v ← 0, δ ← −1, g ← s
2  for i = 0 to 2 × Degree(g) do
3      if f_0 = 1 then
4          if δ < 0 then
5              (f, s, u, v) ← (f + s, f, u + v, u)
6              δ ← −δ
7          else
8              (f, u) ← (f + s, u + v)
9          end
10     end
11     (f, u) ← (f/x, (u/x)_g)
12     δ ← δ − 1
13 end
```

Another important aspect to consider is the constant time execution of each iteration of the algorithm. As explained in Section 2.5.1, the `ifs` on lines 3 and 4 must be replaced by conditional operations of its contents. When vectorizing with AVX2 we would execute the operations, store the results in temporary variables, and then use the `BLENDV` instruction to select between the temporary variables and the original content depending on the value of the `if` condition. Using the AVX512 instructions this process is simplified. Most AVX512 instructions allow the use of a mask to select between the result of the calculation and the

content of another register. This behavior results in a great performance improvement, but also raises some questions about the security of generating masks from the sensitive data, which will be further discussed in Section 3.3. Listing 5 presents two implementations of a conditional addition of polynomials to illustrate the difference between the AVX2 implementation (lines 1 to 6) and the AVX512 implementation (lines 8 to 11).

```
1  void conditional_add_polynomial_AVX2(__m256i p1[N], __m256i p2[N],
   ↪  __m256i output[N], __m256i mask){
2      for(int i = 0; i < N; i++){
3          __m256i tmp = _mm256_xor_si256 (p1[i], p2[i]);
4          output[i] =  _mm256_blendv_epi8 (output[i], tmp, mask);
5      }
6  }
7
8  void conditional_add_polynomial_AVX512(__m512i p1[N], __m512i p2[N],
   ↪  __m512i output[N], __mmask16 mask){
9      for(int i = 0; i < N; i++)
10         output[i] =  _mm512_mask_xor_epi32 (output[i], mask, p1[i],
           ↪  p2[i]);
11 }
```

Listing 5: Comparison between AVX2 and AVX512 implementation of a conditional addition of polynomials

We tested the replacement of Itoh-Tsujii in the 80-bit version of QcBits: the original Itoh-Tsujii algorithm took 402,020 cycles, while the Wu *et al.* algorithm took 715,993 cycles. These results, however, are biased by the great advantage Itoh-Tsujii takes from the composition of the modulus polynomial at the 80-bit version. As mentioned previously, Itoh-Tsujii works by exponentiating the polynomial to $2^R - 2$, where R is the degree of the modulus polynomial. The 80-bit version uses the polynomial $x^{4,801} - 1$ as the modulus polynomial. This polynomial can be decomposed in $(x - 1) \times f_0 \times f_1 \times f_2 \times f_3$, where each f_i has a degree of 1,200. Itoh-Tsujii takes advantage of that composition to calculate the inverse by exponentiating to $2^{1,200} - 2$, rather than $2^{4,801} - 2$. Wu *et al.* does not operate in that way. In fact, the algorithm is indifferent to the composition of the modulus polynomial.

Polynomials composed of small factors (other than $x - 1$) are no longer recommended due to some security concerns [5]. Therefore, Itoh-Tsujii lost its advantage on newer implementations while Wu *et al.* upheld its performance. Our 128-bit quantum security level version, for example, uses the polynomial $x^{32,749} - 1$ as the modulus (the same as BIKE). This polynomial can only be decomposed in $(x - 1) \times f$, where f has a degree of 32748.

Our complete key generation took 40,265,904 cycles to be executed, which represents a 3.1 times slowdown comparing to BIKE, which took only 12,944,920 cycles. Nonetheless, our implementation is fully constant time, while BIKE's utilizes the NTL Library [63], which is not constant-time.

3.2.3 Batch Key Generation

The batch key generation process, presented by BIKE [2], exploits Montgomery's Trick [18] to calculate multiple polynomials inversion at once. If we want to calculate, for example, the inverse of the polynomials A, B, and C:

1. We first multiply them: $F = A \times B \times C$
2. Then, we calculate the inverse of the product $F^{-1} = Polynomial\_Inversion(F)$
3. Finally, we retrieve each inverse through multiplications:
   - $A^{-1} = F^{-1} \times B \times C$
   - $B^{-1} = A \times F^{-1} \times C$
   - $C^{-1} = A \times B \times F^{-1}$

The main observation of the trick is the small cost of the multiplication in comparison with the inverse. The trick can be used in an arbitrary number of polynomials and it costs 3 multiplications per inverted polynomial after the first. The exact algorithm is shown in Algorithm 9. L is a list of N polynomials to be inverted and the output IL is the list of the inverted polynomials. The algorithm is the same implemented by BIKE, our only advantage is the implementations of the multiplications and the inversion.

Algorithm 9: Montgomery's trick for polynomial inversion algorithm (from BIKE [2])

Input : L, N
Output: $IL_i = L_i^{-1} \mid 0 \leq i < N$

$prod_{0,0} \leftarrow L_0$
for $i = 1$ ***to*** $N - 1$ **do**
  $prod_{0,i} \leftarrow prod_{0,i-1} \times L_i$
end
$prod_{1,N-1} \leftarrow prod_{1,N-1}^{-1}$
for $i = N - 2$ ***to*** 1 **do**
  $prod_{1,i} \leftarrow prod_{1,i+1} \times L_{i+1}$
end
$IL_0 \leftarrow prod_{1,1} \times L_1$
for $i = 1$ ***to*** $N - 1$ **do**
  $IL_i \leftarrow prod_{1,i} \times prod_{0,i-1}$
end

Table 3.3 shows the results of the batch key generation execution. For 100 keys, we achieved a 97% execution time reduction over just one inversion. BIKE's paper reports an 84% gain for the same case. We were not able to execute BIKE's batch key generation using the constant time flags provided by the authors (the execution fails). In this way, we compiled and executed its non-constant version. It should be noted that, in this version of BIKE, not only the polynomial inversion is non constant time but also all other procedures. Their version (fully non-constant time) takes 967,331 cycles per inversion using the batch key generation with 100 keys, which is 22% faster than our version (fully constant time).

Table 3.3: Execution time of the Batch Key Generation

Number of Keys	Cycles - Total	Cycles Per Key
1	40,265,904	40,265,904
100	123,169,967	1,231,700
200	208,169,716	1,040,849
300	289,744,382	965,815
400	371,597,787	928,994

3.2.4 Encryption

Our contribution in the encryption process of QcBits (Algorithm 3) concerns only the multiplication implementation. Originally, QcBits prefer dense multiplications implemented using the `PCLMULQDQ` instruction. Even our AVX2 optimized implementation still used it. When we introduced the AVX512 to the implementation, however, the reduced cost of the conditional operations made the sparse multiplication become much faster than the dense one. Algorithm 10 shows the sparse multiplication implementation. It was already used by the `ref` version of QcBits when `PCLMULQDQ` was not supported. Besides the vectorization using the AVX512 instructions, we also optimized it through the techniques presented in Section 3.1.2.

Algorithm 10: Sparse multiplication algorithm.

Input : A sparse polynomial P_1 and a polynomial P_2
Output: The result polynomial P_R

1 $P_R \leftarrow 0$
2 **foreach** *monomial* $m_i \in P_1$ **do**
3 $\quad P_R \leftarrow P_R + P_2 \times m_i$
4 **end**

Since the introduction of the AVX512 represented a great speedup to the rotation calculations, the gain provided by the pre-calculated table of rotations became proportionally small. In Section 3.1, we use a table with 3 pre-calculations, whereas, in this version, we use just 1 pre-calculated rotation, which results in a gain of about 4% in the encryption time. Our encryption takes 259,306 cycles to be executed, being 1.34 times faster than BIKE, which takes 348,227 cycles. These numbers were measured compiling our implementation without the OpenSSL EVP Library, while BIKE has its own implementations of hash functions. When using the OpenSSL SHAKE256 algorithm to expand the randomly generated indexes, we were able to increase our speedup up to 1.6 times.

3.2.5 Decryption

The algorithm implemented for the decryption is the one presented in Algorithm 5. Besides updating the security level and using the AVX512 instructions, our improvements in the decoding process of this version were mainly focused on the Bit Counter implementation (line 5 of the algorithm). For the same reasons mentioned in Section 3.2.4,

the pre-calculated table of rotations was also reduced to just 1 bit in this version, which resulted in a gain of about 2.5% on the decoding time.

Bit Counter Optimization

The original implementation of the Bit Counter is the BitSlice Adder function, shown in Algorithm 6 and explained in Section 4. The number of iterations N is constant in the original implementation, being equal to the number of bits needed to store the maximum possible value of the bit addition. This upper estimation is necessary for the last iterations of the loop in line 3 of Algorithm 5 when the value being stored is large enough to take N bits. However, each loop iteration can add at most 1 to the adder, so the stored value cannot be higher than the current number of iterations. In this way, the number of iterations of the bit counter can be incremented proportionally to the log of the number of iterations executed by the outer loop, as shown in Algorithm 11. Lines 5 to 9 represent the optimized BitSlice Adder function (Algorithm 6).

In order to avoid any significant performance penalty, we pre-calculated the log function and approximate the results. Since the number of iterations in this algorithm phase is not related to any sensitive data, it does not characterize a side-channel leakage.

Algorithm 11: QcBits Bit-flipping implementation logic with BitSlice Adder Optimization

Input : H', c, s and TH
Output: c

1 $N \leftarrow 1 + \lceil log_2(|H'|) \rceil$
2 $sum[N] \leftarrow \{0, 0, ..., 0\}$
3 **foreach** *index* h_i ***in*** H' **do**
4 $\quad w \leftarrow s \lll h_i$
5 $\quad$ **for** $j = 0$ ***to*** $\lceil log_2(i) \rceil$ **do**
6 $\quad\quad c_{out} \leftarrow sum[j] \wedge w$
7 $\quad\quad sum[j] \leftarrow sum[j] \oplus w$
8 $\quad\quad w \leftarrow c_{out}$
9 $\quad$ **end**
10 **end**
11 $sum \leftarrow BitSliceSubtractor(sum, TH, N)$
12 $c \leftarrow \neg sum[N-1] \oplus c$

Another possible optimization in this algorithm and, especially, in its full adder version is the use of the Ternary Logical instruction, presented in the AVX512 instruction set extension. This instruction is capable of executing any logical operation between three operands and it has a great impact on reducing the number of instructions needed to implement these algorithms. It is also useful to reduce slightly the code size and it has a good latency, but its reciprocal throughput is two times the one of the conventional logical instructions. Nonetheless, we could not observe any performance improvement from its use. Future versions of the instruction, with a better reciprocal throughput, may represent a performance enhancement.

Uniform and constant-time implementation

With the update of security level and, consequently, the parameters of the implementation, it was necessary to recalculate the decoding thresholds and the failure rate for the implementation. For the thresholds, we used the same formula as BIKE. For the failure rate, BIKE defined 10^{-7} as a reasonable failure rate for practical purposes and we also follow their definition. To calculate the number of iterations needed to achieve this failure rate, we executed our algorithm 10^8 times: 10^4 encryptions and decryptions for each key pair, for 10^4 generated key pairs. This testing procedure was defined in the specification of QcBits. BIKE does not provide such information since they did not implement a constant time version of the decoding procedure. The results of the experiment are presented in Table 3.4. It shows, in the first row, the number of decoding iterations needed to recover an error vector; and, in the second row, it shows how many instances (a pair of randomly generate keys and error vector) needed that respective number of iterations to be correctly decoded.

Table 3.4: Distribution of instances per iterations needed for the decoding

Iterations:	3	4	5	6	7	8	9	10	11	12	13	14	15+
Instances:	3,741	6.7×10^7	3.2×10^7	1.9×10^5	2,916	176	27	13	5	3	0	0	4

As can be observed from the table, over 99.8% of the instances are solved with 5 iterations or less. And, from the 10^{-8} instances, only 7 took more than 11 iterations to be correctly decoded. Considering these results and the target failure rate of 10^{-7}, we can have a constant time implementation by fixing the number of iteration to eleven. This version takes 9,803,835 cycles to be executed, which represents 891,257 cycles per iteration, on average. Following this metric, a uniform version would execute in a little less than $5 \times 891{,}257 = 4{,}456{,}288$ cycles on average. Unfortunately, this performance was not achieved. The uniform implementation has the overhead of verifying, at each iteration, if the result of the syndrome calculation is zero (line 2 of Algorithm 1). More than that, the simple possibility of a premature abort makes the compiler avoid more aggressive optimization features. Our final result for the uniform implementation was 5,008,429 cycles.

In comparison, BIKE takes 9,572,412 cycles to execute a uniform decoding and does not present constant-time. It is also worth mentioning that most of its decoding code hot-spots are implemented directly in assembly language, while our implementation only uses C code and Intel Intrinsics macros. Table 3.5 summarizes the results presented in this section.

3.3 Power side-channel vulnerability

A simplified snippet of the original implementation code used for the word rotation is shown in Listing 6. As discussed in Section 3.1, the rotation amount depends directly on the secret key bits and, therefore, must be executed mitigating most side-channel leakages. On line 1, a mask is constructed using the variable `sk_bit`, which represents a secret key

Table 3.5: Performance comparison between this work and BIKE (in cycles)

Operation	This work	BIKE 2	Speedup over BIKE
Key Generation	40,265,904	12,944,920	0.32 [a]
Batch Key Gen. (100 keys) [c]	1,231,700	967,331 [b]	0.79 [a]
Batch Key Gen. (400 keys) [c]	928,994	422,133 [b]	0.45 [a]
Encryption	259,306	348,227	1.34
Constant Time Decryption	9,803,835	-	-
Uniform Decryption	5,008,429	9,572,412	1.91

[a] BIKE's polynomial inversion is not constant time.
[b] Result from a fully non-constant time version.
[c] Cost per key

bit: If the bit is one, then the mask will be all ones, otherwise, the mask will be all zeros. Following this, on line 4, the vector is copied shifted or not depending on this mask.

```
1 mask = 0 - sk_bit;
2 us = 1 << i; // shift amount
3 for (j = 0; j < LEN; j++)
4     w[ j ] = (x[ j + us ] &  mask) ^ (x[ j ] & ~mask);
```

Listing 6: Vulnerable implementation of conditional copy for vector rotation

The problem lies on the fact that the power consumption of setting all bits in a register is perceptibly higher than keeping the register with all its bits zero. An attacker can exploit that fact and discover the secret key through a power measurement of the algorithm execution [55]. We are able to mitigate this vulnerability by using the instruction `BLENDV`, as shown in Listing 7. This vulnerability used to occur not only in the word rotation but in all conditional copies implemented in the original version. We fix all of them in the same way and verified that this modification had little impact on the overall performance ($<$ 1%). The performance results presented in Section 3.1 already include this modification.

```
1 mask = _mm_set1_epi8(sk_bit << 7);
2 us = 1 << i; // shift amount
3 for (j = 0; j < LEN; j++)
4     w[ j ] = _mm_blendv_epi8(x[ j ], x[ j + us ], mask);
```

Listing 7: Secure implementation of conditional copy for vector rotation

When implementing these operations using the AVX512 instructions, this issue returns. Although smaller, the AVX512 instructions use very distinguishable masks for the conditional operations. It is not clear, however, in what measure a power-based attack is viable to succeed against modern Intel Architectures. Their high complexity and relatively low power consumption could make such attacks difficult. Nonetheless, protection against the most common power-based attacks, such as DPA [42], CPA [12] and

MIA [30] were considered an important feature for the implementations in the last NIST's standardization process [11].

3.4 Discussion

In this chapter, we presented an optimized implementation of the decoding process of QcBits for 80-bit classical security level and a entirely rewritten version of QcBits for 128-bit quantum security level. In the decoding optimization, we vectorized the entire algorithm, inserted a table of pre-computed vector rotations and unrolled the rotation calculation loop for the versions `ref` and `clmul`. In the `ref` version, using the SSE, AVX2 and AVX512 instruction sets, we achieved a maximum speedup of 2.57, 4.25 and 10.54 times, respectively, while in the `clmul` version we achieved a speedup of 2.44, 3.92 and 6.25 times when using SSE, AVX2 and AVX512 instructions and compiling with GCC. We also implemented countermeasures for some known side channel vulnerabilities without any significant performance penalty.

Our results clearly demonstrate the algorithm's aptitude for vectorization. The `ref` version, which does not rely on the `clmul` instruction, presented higher gains than the register size increment, showing the great impact of the rotation pre-computation technique. The same occurred with the `clmul` version vectorized with SSE instructions. The use of the table could also be much more efficiently implemented if the hardware provided constant-time memory accesses. Besides that, we also demonstrated that some hardware improvements, such as shifting instructions for 128-bit, 256-bit and 512-bit registers, can be very useful for the algorithm performance, as shown by our 1.12-factor speedup estimation considering this instructions. A 256-bit version of the `clmul` instruction would also provide significant performance gains.

In the rewritten version, we optimized all the QcBits procedures using the AVX512 instruction set; we replaced the polynomial inversion algorithm and the dense multiplications by algorithms that better exploit the new AVX512 instructions; we implemented BIKE's batch key generation process using QcBits' algorithms and our optimization techniques; and we presented new optimization techniques for decoding bit counter. We also executed experiments to determine the number of decoding iterations for a constant-time decoding implementation and pointed out that improvements on some instructions, such as the Ternary Logic Instruction, can have great impact on the performance of this code.

Our rewritten version of QcBits confirms the great impact of our previously developed optimization techniques as well as the new improvements. We achieved speedups of up to 1.9 times in comparison with the current state-of-the-art, BIKE, and most of our optimizations can be applied to BIKE itself, which could bring even faster implementations for QC-MDPC code-based cryptography. Considering the current post-quantum cryptography scenario, the code-based cryptography field is just beginning its rise and, considering the latest performance improvements, it is shaping up as one of the most promising candidates for that end.

Chapter 4

Accelerating the arithmetic algorithms

The three main arithmetic operations used in QC-MDPC implementations are the addition, multiplication, and inversion over binary polynomials. The first is efficient, but the others usually represent over 90% of the execution time of the cryptosystem. Algorithms for arithmetic over binary polynomials are very well-known and studied. Hence, new generic optimizations for them are usually restrained to the implementation aspect only.

In this chapter, we take a different approach to optimize binary field arithmetic, focusing specifically on the QC-MDPC code-based cryptography case. We first selected constant-time algorithms from the literature that better exploit the special characteristics of QC-MDPC polynomials on modern computer architectures, such as the large size and relatively low density. Then, we modified the algorithms to accept configurable parameters that greatly accelerate them at the cost of introducing a negligible probability of failure depending on the input. Finally, we defined methods to correlate this probability of failure (or failure rate) of each algorithm with the impact on performance. In this way, we present the following contributions.

- We introduce the concept of using arithmetic subroutines with a controlled failure rate to accelerate QC-MDPC code-based cryptosystems.
- We present constant-time algorithms for multiplication and inversion over binary polynomials that operate with configurable failure rates.
- We define methods to obtain a correlation between failure rate and performance improvement for each algorithm.
- We show that these algorithms provide a significant performance improvement while introducing an arithmetic failure rate that is negligible compared to the security level of the cryptosystem.

We start by reviewing the algorithms of Section 2.4.3 to observe the use of binary field arithmetic in each phase of the cryptosystem. Key generation (Algorithm 2) uses two multiplications and an inversion over binary polynomials. Encryption (Algorithm 3) uses a multiplication and an addition. Decryption (Algorithm 4) explicitly uses two multiplications and two additions (line 13) per iteration, but the monomial multiplications in line 8 and the monomial additions in line 12 can also be implemented as just one

polynomial multiplication and addition, respectively. In this way, the decryption takes four multiplications and four additions per iteration, plus two of each one at the beginning (line 3).

The addition is implemented as a sequence of `XOR` instructions in most architectures, very efficient operations which leave very little space for software optimization, even considering the introduction of failure. Thus, our work focus on the multiplication and inversion operations. We benchmarked our implementations on an Intel i7-7820X processor with Hyper-Threading and TurboBoost disabled to improve the reproducibility of the experiments [8]. We implemented the algorithms in C language using *intrinsics* for the AVX-512 instruction set extension and compiled with GCC 7.3.1. We used inverted binary and sparse representations of polynomials (example in Section 4.1.1) with a maximum degree of 32,748 and Hamming weight of 137. These values are defined based on the parameters for 128-bit quantum security level in QC-MDPC implementations (Table 2.2).

We implemented the conditional statements of the algorithms in constant-time using conditional operations, as discussed in Section 2.5.1. We choose to implement our code using AVX-512 instructions to allow a direct comparison with other highly optimized implementations of QC-MDPC cryptosystems, such as the *Additional Implementation* in BIKE. Moreover, our implementation benefits from the possibility of implementing constant-time operations using mask registers, as shown in Sections 2.5.1 and 3.2.2. This feature was only introduced recently in Intel architectures, but it is common in others (e.g. ARM A32).

4.1 Polynomial Inversion

We based our inversion method on the inversion algorithm by Wu *et al.* [68], which we showed in Algorithm 8 in Section 3.2.2. We reproduce it again in Algorithm 12 using a slightly different notation. The algorithm was created as an extended version of Stein's algorithm [65], which avoids the extra costs of calculating degrees of polynomials that is common in Extended Euclidean Algorithms [39]. It is similar to Brunner *et al.* algorithm [14], differing by the testing of the least significant bit instead of the most significant one. While theoretically equivalent, this modification makes the implementation of Wu *et al.* much simpler than Brunner *et al.*'. Moreover, Wu *et al.* was also designed to be a hardware implementation, which makes it easy to implement in constant-time. The algorithm receives the binary polynomials r, s and u and calculates $\frac{u}{r}$ $(mod\ s)$. It also receives the size of the polynomials, N.

The algorithm iterates over the polynomial r, verifying the existence of a 0-degree monomial and dividing it per x until the monomial is found. Once it occurs, the `if` (line 3) is executed and the resultant polynomial v is modified. As we mentioned, compared to Euclidean Algorithms, one of the main advantages of Stein's algorithm and its derivatives is avoiding expensive degree comparisons between polynomials. To enable the possibility of introducing failures in Wu *et al.*, we had to make a step back on this advantage. We reintroduced a degree verification in the algorithm as a search for the lowest degree monomial of r. This way, we produced the modified version of Wu *et al.* in Algorithm 13.

Algorithm 12: Wu *et al.* Inversion Algorithm [68].

Input : r, s, u and N
Output: $v = \frac{u}{r} \ (mod\ s)$

```
1  v ← 0, δ ← −1, g ← s
2  for i = 0 to 2 × N do
3      if r_0 = 1 then
4          if δ < 0 then
5              (r, s, u, v) ← (r + s, r, u + v, u)
6              δ ← −δ
7          else
8              (r, u) ← (r + s, u + v)
9          end
10     end
11     (r, u) ← (r/x, (u/x)_g)
12     δ ← δ − 1
13 end
```

In Section 4.1.1, we discuss how we implemented the degree verification efficiently.

Algorithm 13: Modified version of the Wu *et al.* Inversion Algorithm [68].

Input : r, s, u, N and F
Output: $v = \frac{u}{r} \ (mod\ s)$

```
1  v ← 0, δ ← −1, g ← s
2  for i = 0 to F × 2N do
3      b ← Smallest_Monomial_Degree(r)
4      (r, u) ← (r/x^b, (u/x^b)_g)
5      δ ← δ − b
6      if r ≠ 0 then
7          if δ < 0 then
8              (r, s, u, v) ← (r + s, r, u + v, u)
9              δ ← −δ
10         else
11             (r, u) ← (r + s, u + v)
12         end
13     end
14 end
```

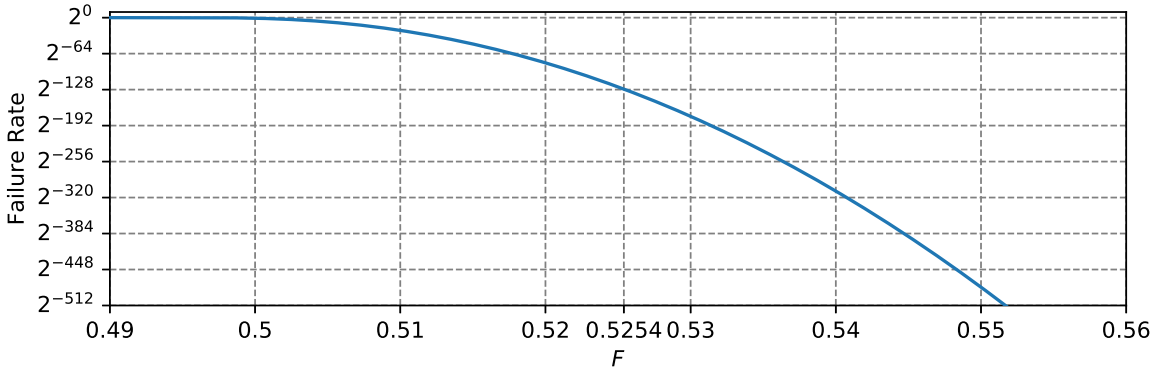
The condition (now in line 6) remained unchanged, a function to find the monomial with the smallest degree was inserted (line 3), the divisors in line 4 were changed from x to x^b, and, finally, the number of iterations is now reduced by the factor $0 < F \leq 1$. Using $F = 1$ would cover all cases, but it would also result in no performance gains. Using lower values of F accelerates the algorithm, but it also inserts the possibility of failure. In this way, our modification exposed a parameter F that allows us to control the trade-off between performance and failure rate in the algorithm. We need now to define a method to precisely associate these two measures and to obtain a good value of F for our case.

Since F changes the number of loop iterations directly, the performance speedup is simply $\frac{1}{F}$. Defining the Failure Rate (FR) in terms of F is significantly more difficult. First, using $r_{0,i}$ as the value of r_0 in line 3 at the i-th iteration of Algorithm 12, we define a polynomial $R(x) = \sum_{i=0}^{2\times N} r_{0,i}\ x^i$. The failure rate can be defined as the probability (function P) of $R(x)$ having a Hamming Weight (HW) greater than $(F \times 2 \times N)$, i.e $FR = P(HW(R(x)) > (F \times 2N))$. To solve this correlation exactly, we would have to consider the number of possible $R(x)$ polynomials and the probability of each one being generated by the algorithm. A simpler approach that results in a good approximation is to consider the probability of each monomial value individually.

Supposing that each monomial coefficient of R has an independent probability p of being 1, we can use the Binomial Expansion [38] to obtain the approximation in Equation 4.1. Further supposing that $p = 0.5$ and using $N = 32749$, we obtain the chart of Figure 4.1, which correlates the Failure Rate with the parameter F and give us the value $F = 0.5254$ to achieve a negligible failure rate. Using this value of F, the cost of Wu *et al.* reduces from 39,747,301 to 20,773,925 cycles, which represents a speedup of 1.9.

$$FR = P(HW(R(x)) > (F \times 2N)) \approx 1 - \sum_{i=0}^{\lceil F\times 2N\rceil} \binom{2N}{i} (1-p)^{(2N-i)} p^i \qquad (4.1)$$

Figure 4.1: Correlation between failure rate and parameter F in the inversion algorithm.



In this way, we defined the method to correlate Failure Rate and Performance Speedup using the parameter F and found a value that fits our case. However, we did it based on the assumption that each monomial coefficient of $R(x)$ has an independent probability $p = 0.5$ of being 1. To show that this is a good estimation for any input polynomials, we define the recurrence relation in Equation 4.2 for the probability of $r_{0,i} = 1$. Analyzing it, we have that if $P(r_{j,0} = 1) < 0.5$ then $P(r_{j,i} = 1) \leq 0.5$ for all $0 \leq i \leq 2N$, which is demonstrated in Proposition 1. This approximation is not as tight as it could for low-density polynomials, but the F parameter resultant is sufficiently close to the optimization

limit (0.5).

$$P(r_{j,i}=1) = \begin{cases} P(r_{j,0}=1), & \text{if } i=0 \\ P(r_{0,i-1}=1) \times P(s_{j+1,i-1} \oplus r_{j+1,i-1}=1) & \\ \qquad + \ P(r_{0,i-1}=0) \times P(r_{j+1,i-1}=1), & \text{if } i>0 \end{cases} \tag{4.2}$$

Proposition 1. *In Equation 4.2, if* $P(r_{j,0}=1) \leq 0.5$ *then* $P(r_{j,i}=1) \leq 0.5$ *for all* $i \geq 0$.

Proof. We prove it using induction on i.
Base case: If $i=0$, then $P(r_{j,i}=1) = P(r_{j,0}=1) \leq 0.5$.
Inductive Hypothesis: if $P(r_{j,0}=1) \leq 0.5$ then $P(r_{j,i}=1) \leq 0.5$ for all $1 \leq i < n$
Inductive Step: Let $i=n$.

$$P(r_{j,n}=1) = P(r_{0,n-1}=1) \times P(s_{j+1,n-1} \oplus r_{j+1,n-1}=1) + P(r_{0,n-1}=0) \times P(r_{j+1,n-1}=1)$$

Knowing that

$$\begin{aligned} P((X \oplus Y)=1) &= P(X=1) \times P(Y=0) + P(X=0) \times P(Y=1) \\ &= P(X=1) \times (1 - P(Y=1)) + (1 - P(X=1)) \times P(Y=1) \\ &= P(X=1) + P(Y=1) - 2 \times P(Y=1) \times P(X=1) \end{aligned} \tag{4.3}$$

We have

$$\begin{aligned} P(r_{j,n}=1) &= P(r_{0,n-1}=1) \times [P(r_{j+1,n-1}=1) + P(s_{j+1,n-1}=1) \\ &-2 \times P(s_{j+1,n-1}=1) \times P(r_{j+1,n-1}=1)] + P(r_{0,n-1}=0) \times P(r_{j+1,n-1}=1) \\ &= P(r_{j+1,n-1}=1) + P(r_{0,n-1}=1) \times [P(s_{j+1,n-1}=1) \\ &-2 \times P(s_{j+1,n-1}=1) \times P(r_{j+1,n-1}=1)] \end{aligned} \tag{4.4}$$

Let $f(X,Y,Z)$ be the value of $P(r_{j,n}=1)$ for $X = P(r_{0,n-1}=1)$, $Y = P(r_{j+1,n-1}=1)$ and $Z = P(s_{j+1,n-1}=1)$. By our inductive hypothesis, $0 \leq P(r_{0,n-1}=1) \leq 0.5$ and $0 \leq P(r_{j+1,n-1}=1) \leq 0.5$. We could obtain a tighter interval for $P(s_{j+1,n-1}=1)$ addressing its own recurrence relation, but that is not necessary. Thus, we consider $0 \leq P(s_{j+1,n-1}=1) \leq 1$. To maximize the value of f in these intervals, we first check the boundaries:

$$f(0,0,0) = f(0,0,1) = f(0.5,0,0) = 0$$

$$f(0,0.5,0) = f(0,0.5,1) = f(0.5,0,1) = f(0.5,0.5,0) = f(0.5,0.5,1) = 0.5$$

The next step would be a search for a local maximum, which clearly does not exist since f is linear in all variables.

□

4.1.1 Implementation

Our modification of Wu *et al.* algorithm introduced two main drawbacks in the performance of the algorithm. The first is the constant-time implementation of the function $Smallest\_Monomial\_Degree$. For large polynomials (such as the ones used in code-based cryptography), it would be very expensive to search the smallest monomial on the entire polynomial. Therefore, we search only the first E bits of the polynomial, change the `If` condition to test if the result is different of E and adjust the number of iterations to compensate for this limitation. Algorithm 14 shows this modification. Using the inverted binary representation of the polynomial (shown in Figure 4.2), we can obtain the degree of the smallest monomial by calculating the number of leading zeros of the representation. Most of the modern architectures enable this calculation with just an instruction. Intel, for instance, provides the leading zeros instructions executed in constant-time for 32-bit words (since i386), 64-bit words (since Haswell), and 64-bit lanes on SIMD registers (AVX-512). Other architectures enable equivalent or complementary operations, such as rounded binary logarithm or trailing zeros, which may require modifications in the polynomial representation, but, ultimately, would not impact the performance.

$$x^7 + x^3 + x^2 + 1 \leftrightarrow \left[1, 0, 1, 1, 0, 0, 0, 1\right] \leftrightarrow \left[0, 2, 3, 7\right]$$

Figure 4.2: Example polynomial, its inverted binary representation and its sparse representation, respectively.

Algorithm 14: Modified version of Wu *et al.* Inversion Algorithm [68] considering the parameter E.

Input : r, s, u, N, F and E
Output: $v = \frac{u}{r}\ (mod\ s)$

$v \leftarrow 0$, $\delta \leftarrow -1$, $g \leftarrow s$
for $i = 0$ ***to*** $F \times (2 \times N + \frac{2 \times N}{E})$ **do**
  $b \leftarrow Smallest\_Monomial\_Degree(r, E)$
  $(r, u) \leftarrow (r/x^b, (u/x^b)_g)$
  $\delta \leftarrow \delta - b$
  if $b \neq E$ **then**
    If's content, unchanged
  end
end

The second drawback in our version are the divisions. In the original algorithm, the divisor was always x. We modified it to x^b, where $0 < b \leq E$. Constant-time divisions usually have its execution time defined by the upper bound of the divisor and, thus, the parameter E also appears as a trade-off between the number of iterations and the performance of each iteration. Fortunately, it is easy to optimize its value in our case. Using SIMD registers in the Intel architecture, the execution time of dividing by x or x^{64} is the same, while greater exponents require much more expensive instructions to move bits across the SIMD lanes. In this way, we choose $E = 64$, which also helps the implementation of the function $Smallest\_Monomial\_Degree$.

4.1.2 Experimenting with higher failure rates

Another optimization through the introduction of the possibility of failure we found on Wu *et al.* algorithm concerns the operations over the polynomials r and s. When a polynomial is large enough to need more than one word to be stored, which is our case, any operation over the polynomial is implemented iteratively over the array of words that stores it. In the Wu *et al.* algorithm, the polynomial r always converges to zero, which implies in a degree reduction of the polynomial along the iterations of the algorithm. As the higher part of the binary representation of the polynomial becomes zeros, it does not need to be processed anymore. In constant-time implementations, however, all the words belonging to the array are always processed. In this way, we cannot check whether the higher parts are zeros or not and only process it based on this information. Nevertheless, we can estimate the degree reduction of the polynomials to decide if the higher parts need to be processed. This estimation aims to cover only the majority of the cases, but not all of them. Therefore, a failure rate can also be explored at this point.

For this modification, however, we did not define a strict correlation between the failure rate and the performance level. Instead, we estimate the failure rate through experimental data targeting at a 10^{-8} failure rate. Since the cryptosystems already have a global failure rate (the DFR) of around 10^{-7} in modern implementations, we can also introduce algorithms for the arithmetic operations that fail with a small but non-negligible probability. If we guarantee that this probability is at least one order of magnitude smaller than the DFR, then the impact on the global failure rate of the cryptosystem will be almost negligible.

To achieve a 10^{-8} failure rate, we measured the degree of 10^8 polynomials along the iterations of the algorithm. Figure 4.3 shows the minimum, maximum and average degrees of the polynomials. The step-function curve represents the upper bound estimation applied to achieve a failure rate smaller than 10^{-8}. It is a step-function because we only choose to process or not entire words of 512 bits. The zoomed portion shows that there is no intersection between the curves of upper bound and maximum value.

A similar optimization can be applied to the polynomials u and v. The convergence of v to the inverse of r is similar to the convergence of r to zero. While r has its degree reduced along the iterations, v grows from the higher part through the insertions of 0-degree monomials followed by modular divisions. In this way, the lower part of v (and, consequently, u) is composed of zeros in the first iterations and, hence, does not need to be processed. We also estimated the degree of the lowest monomial of u along the iterations and establish a lower bound to achieve a 10^{-8} failure rate. Figure 4.4 shows the results. An exception in the values of the lower part is the 0-degree monomial, which is not zero from the beginning. We eliminated it by dividing u per x before the algorithm and multiplying the result by x afterward.

The result of our upper and lower bound estimations for the degree of, respectively, r and u, was that r has its degree reduced at least 0.94 per iteration starting from 32,768 while u has the degree of its smallest monomial decreased at most 2 per iteration starting from 23,552. Using these bound estimations, we further reduced the number of cycles needed to calculate the inverse in our implementation from 20,773,925 to 14,979,764,

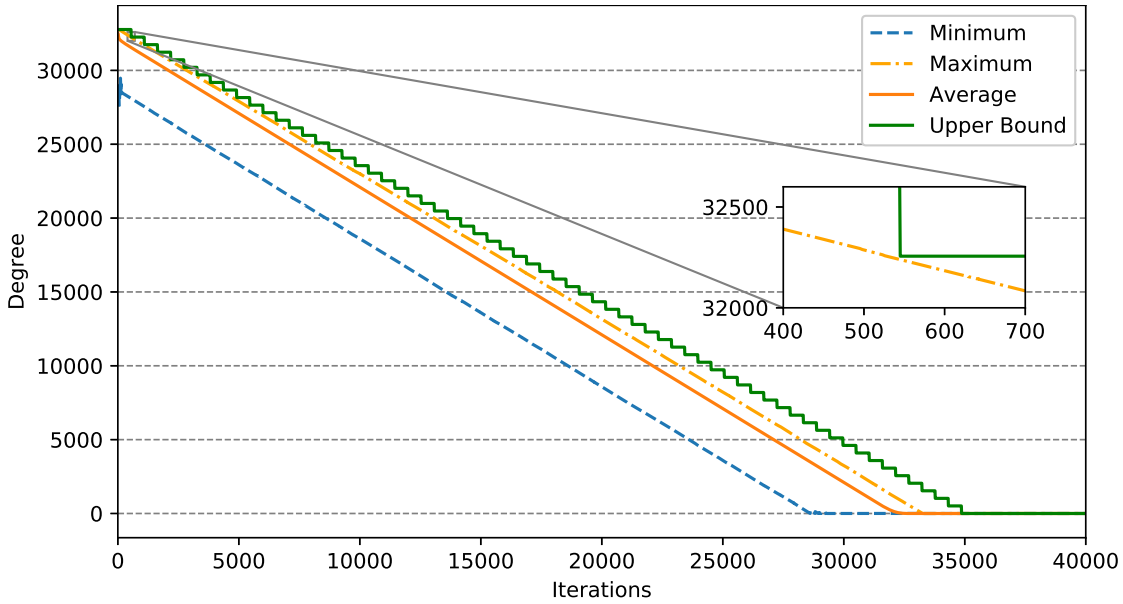


Figure 4.3: Minimum, maximum and average degrees of r for 10^8 randomly generated polynomials along the iterations of Algorithm 14. The step-function curve represents an estimated upper bound to achieve a 10^{-8} failure rate.

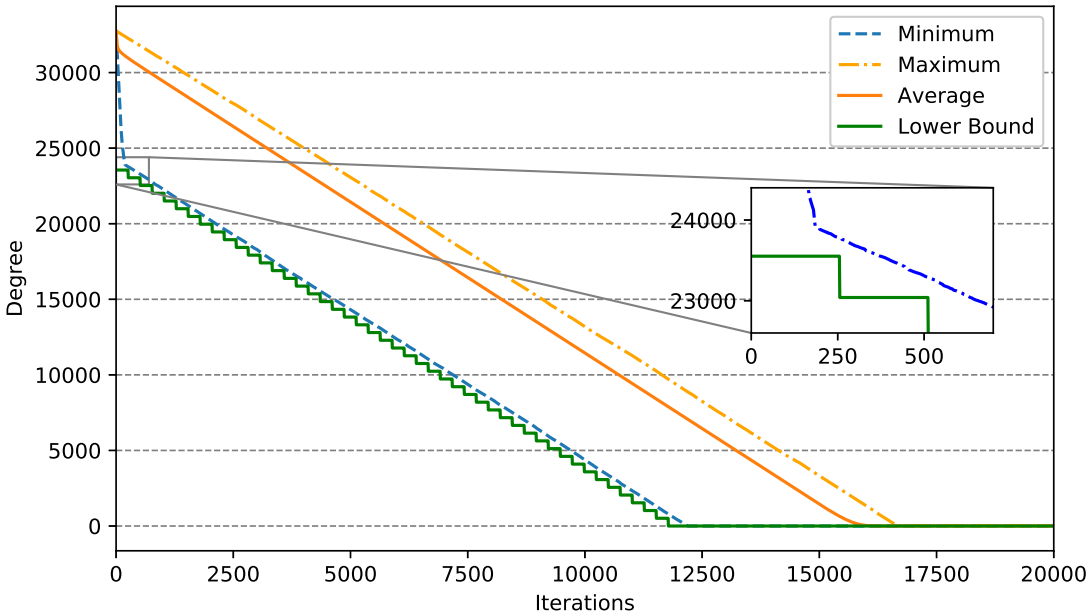


Figure 4.4: Minimum, maximum and average degrees of u for 10^8 randomly generated polynomials along the iterations of Algorithm 14. The step-function curve represents an estimated lower bound to achieve a 10^{-8} failure rate.

which represents an overall speedup of 2.65.

Once we inserted the possibility of non-negligible failures, we also need to provide the methods to detect it. Ideally, we would detect the failure before trying to calculate the inverse, which would avoid a useless execution of this expensive procedure. Unfortunately, it is not simple to describe a correlation between the input and the occurrence of a failure. However, after the execution of the algorithm, it is easy to detect whether the inverse is correct or not by verifying if the product between the result and the input is the identity (one). This verification is already done by most implementations of code-based cryptography and is shown in line 4 of Algorithm 2. It should also be noted that the insertion of non-negligible failure rates requires a more thorough security analysis. The failure occurrence can usually be detected by an attacker and accumulating failures from multiple sources might result in an undesirably high overall failure rate for the cryptosystems.

4.2 Polynomial Multiplication

We start with a very basic algorithm that multiplies an operand by each monomial from the other and adds the results. Algorithm 15 shows this procedure. For generic polynomials, this algorithm is usually outperformed by specialized instructions, such as Intel's `PCLMULQDQ`. However, the algorithm has a good performance if at least one of the polynomials has a relatively low density. As a general rule, this algorithm will outperform the specialized instructions if the density of bits of at least one of the polynomials is less than $\frac{1}{word\ size}$, where *word size* is the word size of the instructions.

Algorithm 15: Sparse multiplication algorithm.

Input : A sparse polynomial P_1 and a polynomial P_2
Output: The result polynomial P_R

1 $P_R \leftarrow 0$
2 **foreach** *monomial* $m_i \in P_1$ **do**
3 $\quad P_R \leftarrow P_R + P_2 \times m_i$
4 **end**

Since additions are very cheap, the cost of the procedure can be estimated as the number of monomials multiplied by the cost of each *monomial* $\times$ *polynomial* multiplication. Considering that it is not possible to reduce the number of monomials, we focused on optimizing the monomial multiplication. Our modulus polynomial has the format $(x^n - 1)$, with $n \geq 1$. Thus, the monomial multiplication can be implemented as just a rotation of the binary representation of the polynomial. This rotation is very simple to implement when the polynomial is small enough to be stored inside just one word of the architecture. For larger polynomials, however, it is necessary to perform a rotation of the words followed by a rotation with carry inside the words.

Algorithm 16 shows how the operation can be implemented in constant-time. It is based on the implementation used by QcBits [16]. M is the degree of the monomial, P is the polynomial in the binary representation, *word_size* is the word size in which a shift

or rotation operation can be executed in the architecture, and M_{max} is an upper bound for M. This procedure has complexity $\mathcal{O}(log_2(M_{max}))$. It should be noted that M in Algorithm 16 is the degree itself. In Algorithms 15, 17, 18 and 19, however, m_i is the entire monomial $m_i = c_i x^{M_i}$, where $(c_i \in 0, 1)$ is the coefficient and M_i is the degree.

Algorithm 16: Constant-time implementation of a $Monomial \times Polynomial$ multiplication

Input : M, P, $word\_size$, M_{max}
Output: $P_{out} = x^M \times P$

1 $P_{out} \leftarrow P$
/* Word rotations */
2 **for** $i \leftarrow \lfloor log_2(M_{max}) \rfloor$ **to** $log_2(word\_size) + 1$ **by** -1 **do**
3 | **if** $M \wedge 2^i$ **then**
4 | | $P_{out} \leftarrow P_{out} \lll 2^i$ // Implemented through word copies
5 | **end**
6 **end**
/* Rotation inside the words (with carry) */
7 $P_{out} \leftarrow P_{out} \lll (M \wedge (word\_size - 1))$

There are two ways of improving the performance of this algorithm: to increase the word size, which would rely on hardware modifications; or to decrease the upper bound of M, which we explore in this work. In the original version, the algorithm multiplies the polynomial with higher density, P_2, by the absolute values of each monomial belonging to the sparse polynomial P_1. Our first modification, presented in Algorithm 17, is to store the result of the previous multiplication and use it to reduce the monomial exponent of the following multiplications. This is basically an application of Horner's rule. For example: If we want to multiply P_{ex} by $(x^{13} + x^7 + x^2)$, Algorithm 15 would calculate $(P_{ex} \times x^{13}) + (P_{ex} \times x^7) + (P_{ex} \times x^2)$ while Algorithm 17 calculates $(P_{ex} \times x^2) \times ((x^5 \times (x^6 + 1)) + 1)$. The number of operations is the same, but Algorithm 17 reduces the value of M_{max} from 13 to 6. This reduction is more significant the higher the number of monomials and, in a non-constant-time implementation, it might provide an immediate gain with this modification.

On constant-time implementations, if we consider the worst-case scenario, the upper bound M_{max} is close to the maximum degree of the monomials, resulting in almost no performance gain. At this point, we introduce a trade-off between performance and failure rate by defining an upper bound M_{max} which does not cover all the cases. Using 512-bit words and 32749-bit polynomials, the possible values for M_{max} are 2^i for $9 \leq i \leq 15$. While this already represents a controlled failure rate, its granularity and the correlation with performance are not good enough. Thus, we produce a third version of the algorithm, depicted in Algorithm 18. If we set, for example, $M_{max} = 512$, we would have a great performance gain, but our failure rate in Algorithm 17 would be almost 100%. The algorithm fails because at least one pair of consecutive monomials has a difference between their exponents greater than the defined upper bound. In Algorithm 18, we introduce an auxiliary polynomial, P_A, which is specially constructed by inserting one or more monomials between those pairs, until the upper bound is respected and the failure is,

Algorithm 17: Sparse multiplication algorithm using the relative degree of the monomials.

Input : A sparse polynomial P_1 and a polynomial P_2
Output: The result polynomial P_R

```
1 P_R ← 0
2 foreach monomial m_i ∈ P_1 do
3 |   if i = 0 then
4 |   |   P_2 ← P_2 × m_0
5 |   else
6 |   |   P_2 ← P_2 × m_i / m_{i-1}
7 |   end
8 |   P_R ← P_R + P_2
9 end
```

consequently, eliminated. When adding the results to P_R (line 9 of Algorithm 18), the intermediate results generated by monomials belonging to P_A are ignored, not affecting the final product. In this way, P_A enables the algorithm to perform the multiplications in which it would fail. Algorithm 19 shows how to construct the auxiliary polynomial P_A in constant time.

Algorithm 18: Sparse multiplication algorithm using the relative degree of the monomials and an auxiliary polynomial.

Input : A sparse polynomial P_1, a polynomial P_2, M_{max} and HW_{P_A}
Output: The result polynomial P_R

```
1  P_R ← 0
2  P_A ← ConstructPA(P_1, M_max, HW_PA)
3  foreach monomial m_i ∈ (P_1 + P_A) do
4  |   if i = 0 then
5  |   |   P_2 ← P_2 × m_0
6  |   else
7  |   |   P_2 ← P_2 × m_i / m_{i-1}
8  |   end
9  |   if m_i ∉ P_A then
10 |   |   P_R ← P_R + P_2
11 |   end
12 end
```

The number of monomials in P_A also needs to be fixed to preserve the constant-time execution. In this way, we have to define the Hamming weight of P_A, HW_{P_A}, for a given upper bound M_{max}, to achieve a desired failure rate. To define this failure rate in terms of HW_{P_A} and M_{max}, we will count the number of polynomials that would either fail or succeed under each of the possible parameters. To do that, we enunciate our problem as a problem of counting the number of restricted compositions of an Integer. A k-composition of n is an ordered sum of k positive integers that results in n [64]. A composition is an

Algorithm 19: Construction of the auxiliary polynomial P_A

Input : P_1, M_{max} and HW_{P_A}
Output: P_A

```
1  P_A ← 0
2  for i ← 0 to HW_{P_A} do
3      foreach monomial m_i ∈ (P_1 + P_A) do
4          if i = 0 then
5              if m_i ≥ x^{M_max} then
6                  P_A ← P_A + x^{M_max − 1}
7                  break
8              end
9          else
10             if m_i / m_{i−1} ≥ x^{M_max} then
11                 P_A ← P_A + m_{i−1} × x^{M_max − 1}
12                 break
13             end
14         end
15     end
16     if HammingWeight(P_A) ≤ i then
           /* Adds monomials with degrees greater than degree of P_1   */
17         P_A ← P_A + x^{Degree(P_1)+i}
18     end
19 end
```

ordered partition, for example: $(2 + 5 + 3)$ and $(2 + 3 + 5)$ are the same partition of 10, but they are two different 3-compositions of 10.

The number of possible polynomials for each key of the cryptosystem is equal to the number of $\frac{W}{2}$-compositions of R, where W and R are the security parameters. To find the failure rate for the multiplication algorithm, we need a restriction to the composition count. For each part p of a composition, we define the number of *M-violations* as $\lfloor \frac{p}{M} \rfloor$. The total number of *M-violations* of a composition, H, is the sum of the number of *M-violations* of each part. Defining a k-composition of n with at most H *M-violations* as a (M, H)-restricted k-composition of n, the failure rate for the multiplication algorithm is

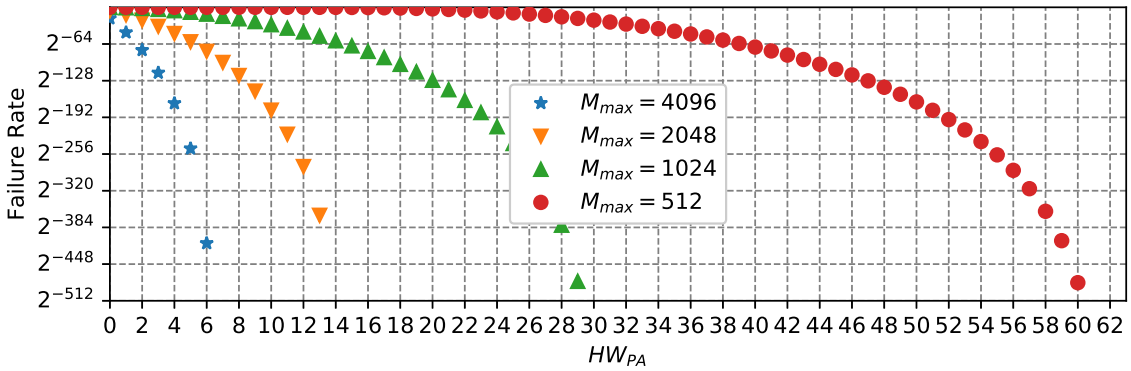
$$\left(1 - \frac{\text{number of } (M_{max}, HW_{P_A})\text{-restricted } \frac{W}{2}\text{-compositions of } R}{\text{number of } \frac{W}{2}\text{-compositions of } R}\right)$$

To calculate the number of compositions, we define the recurrence relation in Equation 4.5, where $C[k, n, H]$ is the number of k-compositions of n with exactly H *M-violations*. Using dynamic programming, we solve the recurrence and produced the chart in Figure 4.5. We did not analyze values of M_{max} greater than 4096 (2^{12}) because their performance considering $HW_{P_A} = 0$ was already worse than $M_{max} = 512$ with

$HW_{P_A} = \lfloor \frac{R}{M_{max}} \rfloor - 1$, which is a case without failure.

$$C[k,n,H] = \begin{cases} 1, & \text{if } k = 1 \text{ and } H = \lfloor \frac{n}{M} \rfloor \\ \sum_{i=max(k-1,n+1-M(H+1))}^{n-1} C[k-1,i,H-\lfloor \frac{n-i}{M} \rfloor], & \text{if } k > 1 \\ 0, & \text{otherwise} \end{cases} \quad (4.5)$$

Figure 4.5: Correlation between failure rate and the value of the parameters M_{max} and HW_{P_A} in the multiplication algorithm.



We estimated the performance cost of each M_{max} presented in the chart using the HW_{P_A} necessary to obtain negligible failure rates. Table 4.1 shows the performance results. The best estimated result was for $M_{max} = 1024$, closely followed by $M_{max} = 512$. Despite being slightly slower, we recommend the use of $M_{max} = 512$ with $HW_{P_A} = 47$ because it facilitates the implementation. Over the initial multiplication algorithm, these parameters provide a speedup of 1.63, reducing the execution time from 130,023 cycles to 76,805 cycles.

Table 4.1: Estimated and measured execution cost (cycles) of the multiplication according to the value of M_{max}.

M_{max}	Multiplication Cost	HW_{P_A}	Estimated Total Cost	Total Cost
512	56,388	47	75,733	76,805
1024	65,747	20	75,345	76,094
2048	74,645	9	79,549	-
4096	87,014	4	89,555	-

Sorting the degree of monomials. Our multiplication algorithm requires the monomials of the sparse polynomial to be sorted by their degree. Thus, a constant-time sorting algorithm is necessary. We implemented a vectorized constant-time version of Counting Sort that sorts by degree and creates the polynomial P_A at the same time. Its execution takes 66,833 cycles, which raises the total cost of the multiplication to 130,464 if we consider that only one multiplication will be executed over a certain sparse polynomial. That is the case in the encryption of QC-MDPC cryptosystems, and, in this step, we

have no performance gain on the multiplication. On the decryption, however, the same sparse monomial is used in many multiplications. In the constant-time case in QcBits, for example, each key needs to be sorted once and is then used in 22 multiplications. In this way, the sorting cost per multiplication is about 3,000 cycles, raising the multiplication cost to only 79,843. Moreover, the sorting can be pre-processed at the key-generation procedure. Alternatives to overcome this problem are to design a polynomial generation algorithm that provides monomials already sorted by the degree or to use more efficient constant-time sorting algorithms, such as Bernstein's djbsort [9].

Consideration about the new `VPCLMULQDQ` instruction. An AVX-512 version of Intel's *PCLMULQDQ* instruction is announced as part of the Ice Lake architecture. Drucker *et al.* [21] estimated that this instruction will bring a speedup of about 2 times to multiplication algorithms that use it, which is the case of the NTL Library, for example. This forthcoming improvement, however, does not diminish the importance of our multiplication algorithm. Our algorithm not only accelerates the generic sparse multiplications but also the Unsatisfied Parity-Check Counting (lines 7 and 8 of Algorithm 4), which is very similar to a sparse multiplication, but that can not be implemented using the *PCLMULQDQ* instruction. Moreover, our implementation relies only on instructions that are much more commonly available than a carry-less multiplicator and that are also more likely to receive hardware improvements.

4.3 Results

First, in order to improve our analysis, we measured the use and cost of each of the arithmetic operations in the implementation. Table 4.2 shows the results. We considered a constant-time decryption using 11 iterations.

Table 4.2: Use and execution cost (cycles) of the arithmetic operations in QcBits at the 128-bit quantum security level.

Procedure	Addition			Multiplication			Inverse			Total
	#	Cost	%	#	Cost	%	#	Cost	%	
Key Generation	0	0	0.00	2	260,046	0.65	1	39,762,389	98.75	40,265,904
Encryption	1	91	0.04	1	130,023	50.14	0	0	0.00	259,306
Decryption	46	4,186	0.04	46	5,981,058	61.01	0	0	0.00	9,803,835

As expected, the cost of the addition is negligible. The inversion takes 98.8% of the key generation and the multiplication takes 61% and 50% of the decryption and encryption, respectively. The remaining time in the key generation and encryption is taken by the random generation of the polynomials. In the decryption, the remaining time corresponds to the counting of satisfied parity-checks, represented by the function *IntegerPolynomialAddition* in Algorithm 4. To obtain a comparison basis for the operations, we executed two other inversion algorithms and one other multiplication algorithm in the same machine and for the same modulus polynomial. All of them use AVX-512 instructions and are highly optimized implementations. Table 4.3 presents the results.

Table 4.3: Comparison among implementations of multiplication and inversion. Bolded lines represent results from this chapter.

Operation	Implementation	Failure Rate	Constant Time	Cost (cycles)
Inversion	**Wu *et al.* modified**	2^{-128}	Yes	20,773,925
	Wu *et al.* modified	10^{-8}	Yes	14,979,764
	Wu *et al.*	0	Yes	39,747,301
	NTL	0	No	12,088,846
	Itoh-Tsujii	0	Yes	243,226,595
Multiplication	**Sparse Mult.**	2^{-128}	Yes	79,843
	Sparse Mult.	0	Yes	130,023
	NTL	0	?	161,715

For benchmarking, we have used version 11.3.0 of the NTL library [63] with gf2x support. Its inversion algorithm, used in BIKE, takes 12,088,846 cycles to invert, which is 1.7 times faster than our algorithm. Considering, however, that NTL's inversion is not constant-time, a 1.7 slowdown is still a good result for our algorithm, which is fully constant-time. The second evaluated algorithm for inversion was Itoh-Tsujii [40], one of the most used constant-time inversion algorithms. We implemented it using NTL's multiplications and squarings and one of the shortest addition chains [26]. It takes 243,226,595 cycles to invert, which is 12 times slower than our algorithm. For the multiplication, using NTL takes 161,715 cycles, which is more than 2 times slower than our multiplication algorithm, even considering the cost of sorting the polynomials (in the context of decryption).

To calculate the impact of our algorithms in a real implementation, we introduced them in QcBits [16]. We are using a version of QcBits at the 128-bit quantum security level, fully constant time and vectorized using AVX-512 instructions [36]. Table 4.4 shows the execution time (cycles) for each of the procedures of the encryption scheme with and without the use of a negligible Failure Rate in the Arithmetic. We also present the execution time of BIKE (Variant 2), for comparison. We compiled the *Additional* version of BIKE (dated 05/23/2018) using NTL version 11.3.0, gf2x support and the following command: `make BIKE_VER=2 CONSTANT_TIME=1 RDTSC=1 AVX512=1 LEVEL=5`.

It is important to note that BIKE paper and this work present different definitions of "constant-time implementations". In this work, we define constant-time implementations as those in which the execution time does not depend on data being processed. The authors of BIKE apparently define constant-time implementations as those which are not vulnerable to known timing side-channel attacks. As we discussed in Section 2.5.1, objectively, both definitions are sufficient to provide security against the currently known attacks, but they need to be differentiated to compare the performance results. For example, the "constant-time" decryption in BIKE executes a variable number of constant-time iterations. In this paper, we refer to it as a *Uniform Implementation*. The number of decoding iterations is dependent on the secret key and can be used to retrieve it [23]. However, all known attacks exploiting the number o decoding iterations rely on a large number of decoding attempts, whereas BIKE uses ephemeral keys [5]. Moreover, BIKE also makes use of masking and blinding techniques on non-constant time procedures,

which we also do not consider as a constant-time implementation.

Table 4.4: Execution time (cycles) of QcBits, with (this chapter) and without (Chapter 3) the Arithmetic Failure Rate; and of the official BIKE implementation, for comparison.

	128-bit QcBits			BIKE-2	
	This chapter	Chapter 3	Speedup	Additional	Speedup
Key Generation	21,332,058	40,265,904	1.89	12,944,920	0.61 *
Encryption	256,655	259,306	1.01	348,227	1.36
Constant-Time Decrypt.	8,016,312	9,803,835	1.22	**	**
Uniform Decryption	3,505,079	5,008,429	1.43	9,572,412	2.73

* BIKE's polynomial inversion is not constant-time.
** BIKE does not present constant-time decryption.
The results for BIKE are equivalent to the column "Constant time implementation" and line "AVX-512" from Table 19 in their paper [2].

The speedups achieved in the Key Generation and in the Decryption were the expected considering the speedup of the operations and their use in the procedures. The Encryption did not present any significant gains, despite having its execution time 50% corresponding to a multiplication. This occurs due to the cost of sorting the monomials, as explained at the end of Section 4.2. The speedups over the official BIKE implementation are not a result solely of the techniques presented in this chapter. As can be observed, the enhanced version of QcBits presented in Section 3.2, which does not present any Failure Rate in the arithmetic, is already faster than BIKE. Nonetheless, the results help to support the advantages of our techniques, which can also be applied to BIKE. Our uniform decryption is 2.7 times faster and our key generation was 1.7 times slower than BIKE's. This slowdown is explained by the use of NTL's polynomial inversion, which is not constant-time.

4.4 Discussion

In this chapter, we presented the concept of inserting a Failure Rate (FR) to the arithmetic of QC-MDPC code-based cryptosystems to achieve significant performance gains. We provided algorithms for multiplication and inversion over binary polynomials that accept a controlled failure rate and defined methods to correlate the failure rate with the performance level. By introducing a negligible ($< 2^{-128}$) FR in these algorithms, we achieved a 1.9-factor performance speedup in the inversion and a 1.63-factor performance speedup in the multiplication. We also showed that our multiplication is 2 times faster than NTL's and that our inversion is 12 times faster than Itoh Tsujii. Finally, we used our algorithms in the QcBits implementation, where we achieved speedups of 1.89 and up to 1.43 in the key generation and decryption, respectively.

Our experimental results show the performance impact of our approach, while the negligible failure rate has basically no downsides to the cryptosystem. The correlation between failure rate and performance improvement was also shown to be very promising, once it is possible to achieve much lower failure rates with little performance penalties. The algorithms we presented have certain advantages when used in QC-MDPC cryptosystems, but, ultimately, they are generic algorithms for arithmetic in GF(2^n) and, thus, could be used in other contexts.

Chapter 5

Conclusion

Our main objective in this work was to optimize the performance of QC-MDPC code-based cryptosystems and we presented several contributions towards this goal. Since we are interested in side-channel protection, we focused on the QcBits implementation and constant-time algorithms. In Chapter 3, the improvements were focused on the implementation aspects. We first vectorized the decoding process of the original QcBits achieving speedups of up to 10.5 times through the use of AVX-512 instructions and pre-calculation. Still in Chapter 3, we presented a new enhanced version of QcBits, entirely vectorized and featuring more efficient algorithms and the updated security parameters from BIKE. Our implementation decrypts messages up to 1.9 times faster than BIKE.

In Chapter 4, we focused on improving the arithmetic procedures used in the implementation of QC-MDPC code-based cryptosystems. We presented methods to accelerate them through the insertion of a controlled probability of failure. By following this approach and introducing a failure rate negligible compared to the security level, we achieved a further speedup of 1.9 times on the key generation and 1.43 times in the decryption. Our final implementation performs the decryption process 2.73 times faster than BIKE.

While we focused on the QcBits implementation, the techniques developed in this work are not restricted to it. Most of them can be applied to other QC-MDPC code-based implementations, such as BIKE, enabling the possibility of further improvements in the field.

5.1 Future Work

As future work, among other contributions, we intend to:

- Apply the techniques developed in the optimization process of QcBits in the BIKE implementation.
- Investigate further optimizations for the Wu *et al.* algorithm and for the syndrome calculation.
- Replicate the approach of failure introduction to other expensive algorithms used in QC-MDPC cryptosystems, for example, the Unsatisfied Parity-Check Counting; or other arithmetic algorithms of cryptographic interest.

Open challenges that might also be considered as future work include the development of viable countermeasures to the Reaction Attack for encryption schemes using non-ephemeral keys; the construction of decoding algorithms that achieve lower failure rates without compromising the performance; the vectorized implementation of QcBits to other architectures, such as ARM, RISC-V and POWER9; and the adaptation of the current implementations to more constrained contexts, such as IoT and embedded devices.

Bibliography

[1] L. Adleman. A subexponential algorithm for the discrete logarithm problem with applications to cryptography. 20th Annual Symposium on Foundations of Computer Science (SFCS 1979), pages 55–60, Oct 1979.

[2] Nicolas Aragon, Paulo S. L. M. Barreto, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Shay Gueron, Tim Guneysu, Carlos Aguilar Melchor, Rafael Misoczki, Edoardo Persichetti, Nicolas Sendrier, Jean-Pierre Tillich, and Gilles Zémor. BIKE: Bit Flipping Key Encapsulation, December 2017. Submission to the NIST post quantum standardization process. Website: `http://bikesuite.org/`.

[3] M. Baldi, F. Chiaraluce, R. Garello, and F. Mininni. Quasi-Cyclic Low-Density Parity-Check Codes in the McEliece Cryptosystem. 2007 IEEE International Conference on Communications, pages 951–956, June 2007.

[4] M. Baldi, P. Santini, and F. Chiaraluce. Soft McEliece: MDPC code-based McEliece cryptosystems with very compact keys through real-valued intentional errors. 2016 IEEE International Symposium on Information Theory (ISIT), pages 795–799, July 2016.

[5] Paulo S. L. M. Barreto, Shay Gueron, Tim Gueneysu, Rafael Misoczki, Edoardo Persichetti, Nicolas Sendrier, and Jean-Pierre Tillich. Cake: Code-based algorithm for key encapsulation. Cryptology ePrint Archive, Report 2017/757, 2017. `http://eprint.iacr.org/2017/757`.

[6] E. Berlekamp, R. McEliece, and H. van Tilborg. On the inherent intractability of certain coding problems (corresp.). *IEEE Transactions on Information Theory*, 24(3):384–386, May 1978.

[7] Daniel J Bernstein. Cache-timing attacks on aes, 2005. `https://cr.yp.to/antiforgery/cachetiming-20050414.pdf`.

[8] Daniel J Bernstein. Supercop: System for unified performance evaluation related to cryptographic operations and primitives, 2009.

[9] Daniel J. Bernstein, Chitchanok Chuengsatiansup, Tanja Lange, and Christine van Vredendaal. NTRU Prime: Reducing Attack Surface at Low Cost. In Carlisle Adams and Jan Camenisch, editors, *Selected Areas in Cryptography – SAC 2017*, pages 235–260, Cham, 2018. Springer International Publishing.

[10] Daniel J. Bernstein, Tanja Lange, and Christiane Peters. Attacking and defending the McEliece cryptosystem. Cryptology ePrint Archive, Report 2008/318, 2008. http://eprint.iacr.org/2008/318.

[11] Guido Bertoni, Joan Daemen, Michael Peeters, Gilles Van Assche, and Ronny Van Keer. Keccak implementation overview, 2012. http://keccak.neokeon.org/Keccak-implementation-3.2.pdf.

[12] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In Marc Joye and Jean-Jacques Quisquater, editors, *Cryptographic Hardware and Embedded Systems - CHES 2004*, pages 16–29, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[13] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701 – 716, 2005.

[14] Hannes Brunner, Andreas Curiger, and Max Hofstetter. On computing multiplicative inverses in GF (2^m). *IEEE Transactions on Computers*, 42(8):1010–1015, 1993.

[15] Lily Chen, Stephen Jordan, Yi-Kai Liu, Dustin Moody, Rene Peralta, Ray Perlner, and Daniel Smith-Tone. Report on post-quantum cryptography (NISTIR 8105 draft), 2016.

[16] Tung Chou. QcBits: Constant-Time Small-Key Code-Based Cryptography. In Benedikt Gierlichs and Axel Y. Poschmann, editors, *Cryptographic Hardware and Embedded Systems – CHES 2016*, pages 280–300, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[17] Mathieu Ciet and Marc Joye. Elliptic curve cryptosystems in the presence of permanent and transient faults. *Designs, Codes and Cryptography*, 36(1):33–43, Jul 2005.

[18] Henri Cohen, Gerhard Frey, Roberto Avanzi, Christophe Doche, Tanja Lange, Kim Nguyen, and Frederik Vercauteren. *Handbook of elliptic and hyperelliptic curve cryptography*. CRC press, 2005.

[19] J Daemen and V Rijmen. Rijndael: The advanced encryption standard. *Doctor Dobbs Journal*, 26(3):137–139, 2001.

[20] Whitfield Diffie and Martin Hellman. New directions in cryptography. *IEEE Trans. Inf. Theor.*, 22(6):644–654, September 2006.

[21] N. Drucker, S. Gueron, and V. Krasnov. Fast multiplication of binary polynomials with the forthcoming vectorized VPCLMULQDQ instruction. In *2018 IEEE 25th Symposium on Computer Arithmetic (ARITH)*, pages 115–119, June 2018.

[22] Nir Drucker and Shay Gueron. A toolbox for software optimization of QC-MDPC code-based cryptosystems. *Journal of Cryptographic Engineering*, Jan 2019.

[23] Edward Eaton, Matthieu Lequesne, Alex Parent, and Nicolas Sendrier. QC-MDPC: A Timing Attack and a CCA2 KEM. In Tanja Lange and Rainer Steinwandt, editors, *Post-Quantum Cryptography*, pages 47–76, Cham, 2018. Springer International Publishing.

[24] Tomáš Fabšič, Viliam Hromada, Paul Stankovski, Pavol Zajac, Qian Guo, and Thomas Johansson. A Reaction Attack on the QC-LDPC McEliece Cryptosystem. In Tanja Lange and Tsuyoshi Takagi, editors, *Post-Quantum Cryptography*, pages 51–68, Cham, 2017. Springer International Publishing.

[25] Jean-Charles Faugère, Ayoub Otmani, Ludovic Perret, and Jean-Pierre Tillich. Algebraic Cryptanalysis of McEliece Variants with Compact Keys. In Henri Gilbert, editor, *Advances in Cryptology – EUROCRYPT 2010*, pages 279–298, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[26] Achim Flammenkamp. Shortest addition chains. Achim's WWW Domain, 2018. `http://wwwhomes.uni-bielefeld.de/achim/addition_chain.html`.

[27] Agner Fog. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs, 2018. `http://www.agner.org/optimize/instruction_tables.pdf`.

[28] RG Gallager. Low-density parity-check codes. MIT press, 1963.

[29] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering*, 8(1):1–27, Apr 2018.

[30] Benedikt Gierlichs, Lejla Batina, Pim Tuyls, and Bart Preneel. Mutual information analysis. In Elisabeth Oswald and Pankaj Rohatgi, editors, *Cryptographic Hardware and Embedded Systems – CHES 2008*, pages 426–442, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[31] Valerii Denisovich Goppa. A new class of linear correcting codes. *Problemy Peredachi Informatsii*, pages 24–30, 1970.

[32] Lov K. Grover. A fast quantum mechanical algorithm for database search. Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing, pages 212–219, New York, NY, USA, 1996. ACM.

[33] Shay Gueron. Personal communication, 2018.

[34] Shay Gueron and Michael E Kounavis. Intel® carry-less multiplication instruction and its usage for computing the gcm mode. *Intel White Paper*, 07 2010.

[35] Antonio Guimarães, Diego F Aranha, and Edson Borin. Optimizing the decoding process of a post-quantum cryptographic algorithm. *XVIII Simpósio em Sistemas Computacionais de Alto Desempenho-WSCAD*, 18(1/2017):160–171, 2017.

[36] Antonio Guimarães, Diego F. Aranha, and Edson Borin. Optimized implementation of QC-MDPC code-based cryptography. *Concurrency and Computation: Practice and Experience*, 2018.

[37] Qian Guo, Thomas Johansson, and Paul Stankovski. A key recovery attack on mdpc with cca security using decoding errors. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *Advances in Cryptology – ASIACRYPT 2016*, pages 789–815, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[38] Richard W. Hamming. *Coding and Information Theory (2Nd Ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986.

[39] Darrel Hankerson, Alfred J. Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography*. Springer-Verlag, 2004.

[40] Toshiya Itoh and Shigeo Tsujii. A fast algorithm for computing multiplicative inverses in GF(2^m) using normal bases. *Information and Computation*, 78(3):171 – 177, 1988.

[41] Neal Koblitz. Elliptic curve cryptosystems. *Mathematics of computation*, 48(177):203–209, 1987.

[42] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In Michael Wiener, editor, *Advances in Cryptology — CRYPTO' 99*, pages 388–397, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[43] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. Advances in Cryptology — CRYPTO '96, pages 104–113, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[44] R. Koetter and A. Vardy. Algebraic soft-decision decoding of Reed-Solomon codes. *IEEE Transactions on Information Theory*, 49(11):2809–2825, Nov 2003.

[45] Yu Kou, Jun Xu, Heng Tang, Shu Lin, and K. Abdel-Ghaffar. On circulant low density parity check codes. Proceedings IEEE International Symposium on Information Theory, pages 200–, 2002.

[46] Chris Lomont. Introduction to Intel Advanced Vector Extensions. *Intel White Paper*, 2011.

[47] Robert J McEliece. A public-key cryptosystem based on algebraic coding theory. *Deep Space Network Progress Report*, 44:114–116, 1978.

[48] Alfred Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[49] Carl D. Meyer. *Matrix analysis and applied linear algebra*. SIAM: Society for Industrial and Applied Mathematics, 2010.

[50] Victor S. Miller. Use of elliptic curves in cryptography. In Hugh C. Williams, editor, *Advances in Cryptology — CRYPTO '85 Proceedings*, pages 417–426, Berlin, Heidelberg, 1986. Springer Berlin Heidelberg.

[51] R. Misoczki, J. P. Tillich, N. Sendrier, and P. S. L. M. Barreto. MDPC-McEliece: New McEliece variants from Moderate Density Parity-Check codes. 2013 IEEE International Symposium on Information Theory, pages 2069–2073, July 2013.

[52] Rafael Misoczki and Paulo S. L. M. Barreto. Compact McEliece Keys from Goppa Codes. In Michael J. Jacobson, Vincent Rijmen, and Reihaneh Safavi-Naini, editors, *Selected Areas in Cryptography*, pages 376–392, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[53] C. Monico, J. Rosenthal, and A. Shokrollahi. Using low density parity check codes in the McEliece cryptosystem. 2000 IEEE International Symposium on Information Theory, pages 215–, 2000.

[54] Dustin Moody. Post-quantum cryptography: NIST's plan for the future. Talk given at PQCrypto, 2016. `https://pqcrypto2016.jp/data/pqc2016_nist_announcement.pdf`.

[55] Erick Nascimento, Łukasz Chmielewski, David Oswald, and Peter Schwabe. Attacking Embedded ECC Implementations Through cmov Side Channels. In Roberto Avanzi and Howard Heys, editors, *Selected Areas in Cryptography – SAC 2016*, pages 99–119, Cham, 2017. Springer International Publishing.

[56] Harald Niederreiter. Knapsack-type cryptosystems and algebraic coding theory. *Prob. Control and Inf. Theory*, 15(2):159–166, 1986.

[57] NIST. Submission requirements and evaluation criteria for the post-quantum cryptography standardization process. NIST web page, 2016. `http://csrc.nist.gov/groups/ST/post-quantum-crypto/documents/call-for-proposals-final-dec-2016.pdf`.

[58] Ayoub Otmani, Jean-Pierre Tillich, and Léonard Dallot. Cryptanalysis of Two McEliece Cryptosystems Based on Quasi-Cyclic Codes. *Mathematics in Computer Science*, 3(2):129–140, Apr 2010.

[59] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, February 1978.

[60] Mélissa Rossi, Mike Hamburg, Michael Hutter, and Mark E. Marson. A Side-Channel Assisted Cryptanalytic Attack Against QcBits. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems – CHES 2017*, pages 3–23, Cham, 2017. Springer International Publishing.

[61] P. W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134, Nov 1994.

[62] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Comput.*, 26(5):1484–1509, October 1997.

[63] Victor Shoup. Number Theory C++ Library (NTL), 2003.

[64] Richard P. Stanley. *What Is Enumerative Combinatorics?*, pages 1–63. Springer US, Boston, MA, 1986.

[65] J Stein. Computational problems associated with Racah algebra. *Journal of Computational Physics*, 1(3):397 – 405, 1967.

[66] Falko Strenzke, Erik Tews, H. Gregor Molter, Raphael Overbeck, and Abdulhadi Shoufan. Side Channels in the McEliece PKC. In Johannes Buchmann and Jintai Ding, editors, *Post-Quantum Cryptography*, pages 216–229, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[67] Mark Weiser, Brent Welch, Alan Demers, and Scott Shenker. *Scheduling for Reduced CPU Energy*, pages 449–471. Springer US, Boston, MA, 1996.

[68] Chien-Hsing Wu, Chien-Ming Wu, Ming-Der Shieh, and Yin-Tsung Hwang. High-speed, low-complexity systolic designs of novel iterative division algorithms in GF(2^m). *IEEE Transactions on Computers*, 53(3):375–380, 2004.