Selected constructive and destructive approaches to post-quantum cryptography

Selected Constructive and Destructive Approaches to Post-Quantum Cryptography

Gustavo Banegas

E-mail: gustavo@cryptme.in
Website: www.cryptme.in

First edition October 2019

This research is supported by the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 643161.

Printed by Printservice Technische Universiteit Eindhoven

A catalogue record is available from the Eindhoven University of Technology Library.

ISBN: 978-90-386-4890-3.

The cover illustrates random waves over binary fields by Liana Clara Pra Baldi da Silveira.
Printed with permission of Liana Clara Pra Baldi da Silveira.

Selected constructive and destructive approaches to Post-Quantum Cryptography

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven,
op gezag van de rector magnificus prof.dr.ir. F. P. T. Baaijens, voor een commissie
aangewezen door het College voor Promoties, in het openbaar te verdedigen
op dinsdag 12 november 2019 om 16:00

door

Gustavo Souza Banegas

geboren te Panambi, Brazilië

Dit proefschrift is goedgekeurd door de promotoren en de samenstelling van de promotiecommissie is als volgt:

voorzitter:	prof. dr. ir. O. J. Boxma
1e promotor:	prof. dr. T. Lange
2e promotor:	prof. dr. D. J. Bernstein
leden:	prof. dr. H. Buhrman (Universiteit van Amsterdam)
	prof. dr. T. Johansson (Lunds Universitet)
	prof. dr. A. May (Ruhr-Universität Bochum)
	dr. P. Schwabe (Radboud Universiteit)
	dr. B. M. M. de Weger

Het onderzoek of ontwerp dat in dit proefschrift wordt beschreven is uitgevoerd in overeenstemming met de TU/e Gedragscode Wetenschapsbeoefening.

Acknowledgments

In my quest to obtain a Ph.D., I have met outstanding people that helped me to develop my knowledge. For this, I am really glad. In a few words, I will try to acknowledge them.

First, I need to say that I did not expect that in my time as a Ph.D. student I was going to acquire such a large cultural, emotional and scientific experience as I received.

I would like to start by saying thank you to my supervisors for helping me to achieve my goal, that is, finishing my Ph.D. I would like to thank my math and chili mom Tanja for always helping me with math in special proofs and writing style, for being really, really patient with me, for giving me chocolate muffins, spicy muffins, introducing me to vegetarian Mapo tofu extra spicy, for chili seeds and of course warning me that when I am in Florida or other US cities that I should not trespass since they will not ask in a polite way for me to get out of their property. Furthermore, I would like to thank you, Dan, for teaching me a lot about coding, quantum computing, math, how to write papers, and to make jokes during talks. I learned a lot with this and I enjoyed a lot to go to the talks because of the content and most importantly the jokes. I thank you both for the opportunity to do my Ph.D. and for all the knowledge that I have acquired in the 4 years that I was your Ph.D. student and the places that I have seen in the world that I have never imagined that one day I would be. I would like to thank both for the guidance during my Ph.D. and to say that if I pursue an academic life they are the major reason for this, I will carry forever with me the values, thoughts, and the love about research and of course the love about math and cryptography.

Secondly, I would like to thank my committee Harry Buhrman, Thomas Johansson, Alex May, Benne de Weger and Peter Schwabe for taking the time to read this thesis and to travel to Eindhoven. I would like extend to an extra thanks to Peter Schwabe. I met you because I was a curious student asking how it was possible to implement securely in Java Cards, after a brief discussion on Skype with Andreas Hülsing, he sent me to talk with you since you were in the same summer school. Since 2013 our paths have crossed several times. In 2014 at Latincrypt we met again in Florianópolis and in the same year you invited me to go to Hamburg to attend CCC. Finally, because of you, I met Dan and Tanja who in the near future became my supervisors.

I had the opportunity to work with many outstanding researchers during my Ph.D. and I would like to thank my coauthors which is a big list because of DAGS. I would like to thank in alphabetic order, all my coauthors which are Paulo S. L. M. Barreto, Daniel J. Bernstein, Brice Odilon Boidje, Pierre-Louis Cayrel, Ricardo Custódio, Gilbert Ndollane Dione, Kris Gaj, Cheikh Thiécoumba Gueye, Richard Haeussler, Jean Belo Klamti, Douglas Martins, Ousmane N'diaye, Duc Tri Nguyen, Daniel Panario, Edoardo Persichetti, Jefferson E. Ricardini, Simona Samardjiska and Paolo Santini. I would like to say that I am very happy to be part of the Italian mob that Edoardo and Paolo made me part of and for the invitations from Edoardo to go to Florida.

I am grateful for the amazing friends that I made during my stay in Eindhoven, I would say that you were really great and that you made my life easier and funny in Eindhoven. In no special order I would like to thank: Andy, Christine, Stefan, Anita, Sober Ale, Drunk Ale, Bouke, Mahdi, Meilof, Franscisco, Rien, Daan, Davide, Ömer, Manos, Taras, Sownya, Murtaza, Pavlo, Dominik, Iggy, Leon, Lorenz, Chloe, Benjamin, Estuardo, Dion, Laura, Niels, Frank, Maran, Manon, (right) Joost, (wrong) Joost, Pedro, Veelasha, Ko, Niels, Matthias, Ruben, Tony, Chitchanok, Thijs, Kai-Chun, Jake and Florian. I would like to extend my many thanks to the other ECRYPT-NET Ph.D. students.

I would like to say thanks to the “Brazilian Storm” and that includes the “gang" from Brazil that helped me and some of which are my coauthors such as Paulo, Jefferson, Douglas and Ricardo, others are cryptography-related friends such as Pedro, Amanda, Thales, Diego, Rafael, Fábio, Lucas, and Gustavo. Moreover, I have friends that stayed in Brazil but maintain contact with me and that includes my former research group LabSEC and my rugby team Goitaka Rugby, I would like to say a special thanks to Felipe for sharing memes, videos, and jokes that made my days more enjoyable and his lovely wife Soraya. I would like to thank Wei Chi, my ninjutsu group in Brazil, especially my sensei Rodrigo. In addition, I would like to say thanks to Liana to be like a sister since 2005 and even with the distance every time that we meet it is like we have not seen each other for one week. At last but not least, I would like to thank Monica for sharing laughs about the similar problems of Ph.D. life even though we are from different areas.

I had several “homes” along my PhD. The first one, the coding and cryptography group in Eindhoven which I am really greatful to spend most of my time here. The second, I went to a internship at Riscure in Delft and I would like to thank them for receiving me, especially Ileana Buhan and Lukasz Chmielewski. The third, I would like to thank CryptoExperts from Paris for the warm welcome, I had a great time with them and I learned a lot with Sonia Belaïd, Matthieu Rivain, Pascal Pailler, Louis Goubin and Junwei Wang. I would like to extend an extra thanks to Mélissa Rossi for helping me in Paris and for several conferences that we have been together.

There is a special place for the ones that paid me beers during my stay in Eindhoven and the responsibles are: Christine, Stefan, Andy, Lorenz and Daan. I am really grateful for all the Fridays at GEWIS, the trips, the parties, the conferences, the conference dinners, the bachelor party, the wakeboarding, the BBQs, and I could write much more occasions that we spend quality time together but I will stop here and just say thank you so much for all the beers.

I will thank Andy for being like a brother, we have been in several trips for work and for surfing. We were surfing side by side in several places that when I was younger I could not imagine that one day I was going to be surfing there, places like Portugal, Canary Islands, California and Netherlands. I am glad that I was surfing in these places with you. I think that my acknowledgements would not be complete if I forgot about the several punk rock festivals that we went to see NOFX, Rancid and other amazing bands.

I would like to thank my mom, my brother and my wife for supporting my decision to move out of my home country. Especially, I would like to thank my mom for always finding a way and send Brazilian food such as Pão de queijo and Pinhão. To my brother, I would like to say: “Haha! I got my Ph.D. younger than you!” and thank you for coming to visit me in Europe and for bringing my Jiujitsu gi. I would say that without this support I think that I could not finish my PhD.

At last but not least, I would like to say thanks to Carolina, my wife. You have been at my side since I decided to start the academic life that is since I started my master's and always helped me when I was in doubt of myself. We have been together for more than 7 years, you are the reason for my first transatlantic travel when during winter, we went to Budapest for volunteer work and since then we always keep traveling together. When I decided to come to the Netherlands for my Ph.D. even knowing that we would be far apart for a time, you supported me and then one day you said that you were quitting your job, selling everything and moving here to do another master's and to be together with me, that is just one of the reasons that I admire you. I would say that you have been my friend, my lover, my inspiration, my sanity and my spellchecker for English and Portuguese. I would say that words cannot express the joy, the happiness and the determination that you always bring to me.

I would like to finish my acknowledgments with a phrase from Invictus from William Ernest Henley that I have as a motto since I read long it time ago.

> "I am the master of my fate, I am the captain of my soul." (William Ernest Henley, 1888)

Gothenburg, September 2019

Gustavo Banegas

Contents

Chapter 1

Introduction

Humans are not machines, and so it was that long before our modern era, people with communication security needs used codes and ciphers which were labor intensive for people. A classic example is the Caesar cipher named after Julius Caesar [Sin00]. As is often the case, and like Caesar, the original Caesar cipher falls to Brutus [Str15]. While many humans may be flummoxed by trying many different combinations, a machine excels at such tedious work. Classical ciphers such as the Caesar cipher would do little to no good in our digital era, and certainly we require methods which relate to the current threats. The Caesar cipher's threats were the mail carrier, or a courier, or perhaps a spy, where even to copy the message encrypted with an unknown system, was labor intensive. Times have changed, though many of the concerns about spies [Orw83] have stayed largely the same. It is the methods for copying messages and the methods for protecting messages which have undergone a monumental shift.

In our current digital era, our communications channels are no longer the human couriers on horseback, regular people deal with larger volumes of messages on a daily basis. It is for this reason, amongst others, that radically different protection schemes are often used to protect data as it travels around the world at the speed of light. When messages need to be protected, we utilize cryptography. When information needs to be hidden, we utilize modern information hiding techniques such as steganography. Even if we were to find ourselves in an Orwellian nightmare [Orw83], cryptography would help to protect messages but as the lessons of Caesar illustrate: cryptography is necessary but considered alone it may not be sufficient. None the less, if cryptography is necessary, what would it entail if one needed it to be done correctly? Modern cryptography relies on mathematical problems, it requires efficient and secure constructions, not merely problems that would stump a horse back rider on their break between cities. Whereas the original Caesar cipher would substitute each letter with another, requiring the communicating parties to agree on a substitution pattern before a message could be understood, modern cryptography allows two strangers to speak privately without any previous coordination.

The mathematical problems that are used in cryptography are selected to ensure that they are efficient for those who are allowed to decode messages, and inefficient for those who are not. For example, Rivest, Shamir, and Adleman created a scheme by the name of RSA [RSA78]. This cryptographic scheme is based on factoring numbers that are products of two primes. At first glance, it seems trivial to solve this problem. If one selects two prime numbers with 2048 bits each, it is even hard to recover the original numbers from the multiplication.

As was previously foreshadowed, things which are hard for humans are sometimes easier for computers. So it is that things difficult for today's computers are likely to be much easier with tomorrow's computers. Not just any computer: the game changes when an attacker has a quantum computer. As early as the 1990s Shor [Sho97] presented an algorithm to take advantage of quantum computation and showed a fast method to factor the product of two primes. This algorithm is much faster than the ones used in classical computers. The consequence of this quantum algorithm is that it breaks cryptographic schemes such as RSA, DSA, and ECDSA. The imminent problem of the break of those cryptosystems is that they are the ones deployed in most of the software and hardware of the modern world's electronic devices. Fortunately, there are some solutions for this problem, that is, post-quantum cryptography.

Post-quantum cryptography is the study of mathematical problems that are believed to be hard to solve with a quantum computer. Nowadays, we have five large areas of mathematics relevant to this problem space, namely, lattices, hash, multivariate quadratic equations, isogenies and codes. The latter is based on linear codes and the first scheme proposed was by McEliece [McE78] in 1978. As is shown in the implementation provided in [BCL$^+$], the scheme is fast and secure but it requires the use of very large public keys. However, there are researchers trying to keep the security and efficiency of this scheme while reducing the size of the keys. In the original proposal, McEliece presents the scheme using Goppa codes. A natural idea is to study the behavior of the scheme when a different code is used. If the keys are reduced, will the scheme continue to provide the same security? One of the first solutions to this question is presented in [MB09,MTSB13] where the authors try to make a compact version of McEliece.

In this thesis, the first part is the explanation of cryptography based on linear codes. Chapter 2 gives the mathematical background necessary to understand the first part of the thesis. In this chapter, there are the definition of dyadic matrices, linear codes and a brief explanation of two cryptosystems based on codes, namely, McEliece and Niederreiter. Chapter 3 continues in the mathematical area and it shows how it is possible to perform multiplication and inversion in an efficient way. Chapter 4 joins the previous chapters and shows that is possible to create a cryptosystem using Generalized Srivastava [Hel72] codes; this cryptosystem is believed to be secure against quantum computers. Furthermore, the chapter details the implementation of this cryptosystem.

The implementation of a cryptosystem is an important step in the deployment of the cryptosystem. It can determine the efficiency of the operations and the security of the scheme. The latter is related to side-channel attacks, and of course, human mistakes. Chapter 5 exploits a type of side-channel attack, i.e., it takes advantage of the leakage of time in certain operations to recover secret information. Later on, the chapter proposes a countermeasure to avoid timing attacks, it shows how to find roots of an error locator polynomial (ELP) avoiding timing leakages.

Another way to attack a cryptosystem is using a reaction attack. A reaction attack consists of sending several messages and waiting for a message that cannot be decoded. When such a message is found, we exploit properties of the failure. In [GJS16], the authors propose a reaction attack against Moderate Density Parity-Check Codes. Chapter 6 proposes a similar attack but the main focus is on Low-Rank Parity-Check (LRPC) codes.

In this chapter, there is an attack that exploits the decoding failure rate, which is an inherent feature of these codes, to devise a reaction attack aimed at recovering the private key. Furthermore, it shows that the same technique can be applied in a quantum setting, i.e., it is possible to apply Grover's algorithm [Gro96] to speed up the attack.

Post-quantum cryptography is not limited to the creation of cryptographic schemes; it is also the study of breaking cryptography using quantum algorithms. This branch of research is often called quantum cryptanalysis. In this field, researchers are developing new quantum algorithms for breaking cryptography, tuning the current quantum algorithms to improve on various aspects of attacks, and estimating the amount of resources that is required to run a given quantum algorithm.

The second part of this thesis is focused on quantum cryptanalysis. Chapter 7 gives an introduction about quantum computing and its limitations. In this chapter, the basics of quantum computation are shown, and an explanation of Grover's algorithm is presented. Chapter 8 shows how it is possible to create a quantum circuit for the Advanced Encryption Standard (AES). This chapter shows a construction of the S-Box that uses fewer quantum gates than is found in the literature.

In the final chapter, Chapter 9, the thesis shows a new quantum algorithm that uses a combination of distinguished points and Grover's algorithm for finding preimages in a hash function. The complexity of the creation of these quantum circuits is that the computation needs to be reversible. This requirement demands a more careful design of the circuit. Another feature of the algorithm presented is that it is designed to run in parallel.

PART I

CODE-BASED CRYPTOGRAPHY

Chapter 2

Background on Code-Based Cryptography

2.1 — Mathematical Background

In this chapter, we will explain the main components of Srivastava codes, Goppa codes and the McEliece cryptosystem. First, we will introduce dyadic matrices and their definitions since we will later use these to build a cryptosystem using Srivastava codes, built on these matrices. Second, we will give a brief explanation of Goppa codes and the McEliece cryptosystem.

2.1.1 – Dyadic Matrices.

Definition 2.1. Given a ring $\mathcal{R}$ and a vector $\mathbf{h} = (h_0, h_1, \ldots, h_{n-1}) \in \mathcal{R}^n$, with $n = 2^r$ for some $r \in \mathbb{N}$, the *dyadic* matrix $\boldsymbol{\Delta}(\mathbf{h}) \in \mathcal{R}^{n \times n}$ is the symmetric matrix with components $\Delta_{ij} = h_{i \oplus j}$ where $\oplus$ stands for bitwise exclusive-or on the binary representations of i and j and $i \oplus j$ is seen as an integer in $[0, n-1]$. Such a matrix is said to have *order* r. The vector $\mathbf{h}$ is called the *signature* of the matrix $\boldsymbol{\Delta}(\mathbf{h})$, and corresponds to its first row. The set of dyadic $n \times n$ matrices over $\mathcal{R}$ is denoted $\boldsymbol{\Delta}(\mathcal{R}^n)$.

As a toy example, let us consider the signature $\mathbf{h} = (a, b, c, d)$; the corresponding dyadic matrix is:

$$M = \begin{pmatrix} \mathbf{h}_{[0,0]\oplus[0,0]} & \mathbf{h}_{[0,0]\oplus[0,1]} & \mathbf{h}_{[0,0]\oplus[1,0]} & \mathbf{h}_{[0,0]\oplus[1,1]} \\ \mathbf{h}_{[0,1]\oplus[0,0]} & \mathbf{h}_{[0,1]\oplus[0,1]} & \mathbf{h}_{[0,1]\oplus[1,0]} & \mathbf{h}_{[0,1]\oplus[1,1]} \\ \mathbf{h}_{[1,0]\oplus[0,0]} & \mathbf{h}_{[1,0]\oplus[0,1]} & \mathbf{h}_{[1,0]\oplus[1,0]} & \mathbf{h}_{[1,0]\oplus[1,1]} \\ \mathbf{h}_{[1,1]\oplus[0,0]} & \mathbf{h}_{[1,1]\oplus[0,1]} & \mathbf{h}_{[1,1]\oplus[1,0]} & \mathbf{h}_{[1,1]\oplus[1,1]} \end{pmatrix} =$$

$$= \begin{pmatrix} \mathbf{h}_{[0,0]} & \mathbf{h}_{[0,1]} & \mathbf{h}_{[1,0]} & \mathbf{h}_{[1,1]} \\ \mathbf{h}_{[0,1]} & \mathbf{h}_{[0,0]} & \mathbf{h}_{[1,1]} & \mathbf{h}_{[1,0]} \\ \mathbf{h}_{[1,0]} & \mathbf{h}_{[1,1]} & \mathbf{h}_{[0,0]} & \mathbf{h}_{[0,1]} \\ \mathbf{h}_{[1,1]} & \mathbf{h}_{[1,0]} & \mathbf{h}_{[0,1]} & \mathbf{h}_{[0,0]} \end{pmatrix} =$$

$$= \begin{pmatrix} \mathbf{h}_0 & \mathbf{h}_1 & \mathbf{h}_2 & \mathbf{h}_3 \\ \mathbf{h}_1 & \mathbf{h}_0 & \mathbf{h}_3 & \mathbf{h}_2 \\ \mathbf{h}_2 & \mathbf{h}_3 & \mathbf{h}_0 & \mathbf{h}_1 \\ \mathbf{h}_3 & \mathbf{h}_2 & \mathbf{h}_1 & \mathbf{h}_0 \end{pmatrix} =$$

$$= \begin{pmatrix} a & b & c & d \\ b & a & d & c \\ c & d & a & b \\ d & c & b & a \end{pmatrix}.$$

Theorem 2.2. Every dyadic matrix M of order $r > 0$ can be written in the form

$$\mathbf{M} = \begin{pmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{B} & \mathbf{A} \end{pmatrix} \tag{2.1}$$

where $\mathbf{A}$ and $\mathbf{B}$ are two dyadic matrices of order $r - 1$. Conversely, every matrix of the form (2.1) is dyadic.
In other words, $\Delta\left(\mathcal{R}^n\right) = \Delta\left(\Delta\left(\mathcal{R}^{n/2}\right)^2\right)$.

Proof. Base case $r = 1$, by Definition 2.1 we have that $\mathbf{h} = (h_0, h_1)$, so

$$\Delta(\mathbf{h}) = \begin{pmatrix} h_0 & h_1 \\ h_1 & h_0 \end{pmatrix},$$

which satisfies the condition. Conversely, a 2×2 matrix of the form (2.1) is dyadic by definition.

The inductive step is to show that the statement holds for $r+1$ assuming that it holds for r, i.e. we have dimension $2n$.
So, $\mathbf{h} = (h_0, \ldots, h_{n-1}, h_n, \ldots, h_{2n-1}) = (\mathbf{h}_0, \mathbf{h}_1)$. We have $r + 1$ bits as indices. Equation 2.2 shows the indices in the matrix along the upper left part of the matrix. Note that those are the indices and not the elements of the matrix.

$$\Delta(\mathbf{h}) = \begin{array}{c|c|c} & 0 \quad \cdots \quad n-1 & n \quad \cdots \quad 2n-1 \\ \hline \begin{array}{c} 0 \\ \vdots \\ n-1 \end{array} & \Delta(h_0) & \Delta(h_1) \\ \hline \begin{array}{c} n \\ \vdots \\ 2n-1 \end{array} & \Delta(h_1) & \Delta(h_0) \end{array} \tag{2.2}$$

Since by definition the xor of a number with itself is always 0, the indices on the upper left and bottom right of Equation 2.2 have that the most significant bit (MSB) will be 0. Furthermore, xor is commutative so these two $n \times n$ matrices are identical. In fact, they are equal to the dyadic matrix of signature $\mathbf{h}_0$ with an additional top bit equal to 0.

Likewise, the MSB of bottom left and upper right matrices will be 1 since one of the indices will have MSB as 1 in the upper half and the other index will have MSB 0 in the other half. The matrices are identical and are equal to the dyadic matrix of signature $\mathbf{h}_1$, where the MSB of the indices is removed to compute the dyadic matrix and then changed to MSB equal to 1 to place it in the upper right or bottom left corner. This means that we can represent the signature with r bits and use the induction hypothesis to show that the matrix $\Delta(\mathbf{h})$ has the form of Equation 2.1.

Conversely, taking $\mathbf{A} = \Delta(h_0)$ and $\mathbf{B} = \Delta(h_1)$ and using the same considerations on the MSB indices as above shows that M is the dyadic matrix of signature $(\mathbf{h}_0, \mathbf{h}_1)$. □

Definition 2.3. A *dyadic permutation* is a dyadic matrix $\mathbf{\Pi}^i \in \mathbf{\Delta}(\{0,1\}^n)$ characterized by the signature $\boldsymbol{\pi}^i = (\delta_{ij} \mid j = 0, \dots, n-1)$, where δ_{ij} is the Kronecker delta (hence $\boldsymbol{\pi}^i$ corresponds to the i-th row or column of the identity matrix).

A dyadic permutation is clearly an involution, i.e. $(\mathbf{\Pi}^i)^2 = \mathbf{I}$. The i-th row, or equivalently the i-th column, of the dyadic matrix defined by a signature $\mathbf{h}$ can be written as $\mathbf{\Delta}(\mathbf{h})_i = \mathbf{h}\mathbf{\Pi}^i$.

A dyadic matrix can be efficiently represented by its signature; furthermore, all operations between dyadic matrices can be computed only using the corresponding signatures. Indeed as we will show below, for any two length-n vectors $\mathbf{a}, \mathbf{b} \in \mathcal{R}^n$, we have:

$$\mathbf{\Delta}(\mathbf{a}) + \mathbf{\Delta}(\mathbf{b}) = \mathbf{\Delta}(\mathbf{a} + \mathbf{b}) \tag{2.3}$$

which means that, given two dyadic matrices $\mathbf{A}$ and $\mathbf{B}$, with respective signatures $\mathbf{a}$ and $\mathbf{b}$, their sum is the dyadic matrix described by the signature $\mathbf{a} + \mathbf{b}$. In an analogous way, the multiplication between dyadic matrices can be done by considering only the corresponding signatures, we will discuss efficient ways for computing multiplications in Section 3. Moreover, it is easy to see that if a dyadic matrix is invertible, its inverse is also a dyadic matrix; this can be easily computed using Sylvester-Hadamard matrices, see Section 3.2. We will expand on this in Section 3.5.

Theorem 2.4. Given two dyadic matrices $\mathbf{A}$ and $\mathbf{B}$, the sum of $\mathbf{A} + \mathbf{B} = \mathbf{C}$ is a dyadic matrix.

Proof. Base case: $r = 0, n = 2^0 = 1$ and $A = (a), B = (b), A + B = (a + b)$.
The inductive step is to show that the statement holds for $r + 1$ assuming that it holds for r. Assume that, $n = 2^{r+1} = 2 \cdot 2^r$ and A_i, B_i are dyadic of order r, we have:

$$\mathbf{A} = \begin{pmatrix} A_0 & A_1 \\ A_1 & A_0 \end{pmatrix}, \mathbf{B} = \begin{pmatrix} B_0 & B_1 \\ B_1 & B_0 \end{pmatrix}$$

$$\mathbf{A} + \mathbf{B} = \begin{pmatrix} A_0 + B_0 & A_1 + B_1 \\ A_1 + B_1 & A_0 + B_0 \end{pmatrix}$$

by the hypothesis and Theorem 2.2 we have that the conclusion holds. □

Theorem 2.5. Given two dyadic matrices $\mathbf{A}$ and $\mathbf{B}$, the product of $\mathbf{A} \cdot \mathbf{B} = \mathbf{C}$ is a dyadic matrix and $\mathbf{\Delta}(\mathbf{a}) \cdot \mathbf{\Delta}(\mathbf{b}) = \mathbf{A} \cdot \mathbf{B}$, where $\mathbf{a}$ and $\mathbf{b}$ are the signatures of matrix $\mathbf{A}$ and $\mathbf{B}$, respectively.

Proof. Base case:
$n = 2^r, r = 0, n = 1$ and $A = (a), B = (b), A \cdot B = (ab)$.

The inductive step is to show that the statement holds for $r+1$ assuming that it holds for r. So, $n = 2^{r+1} = 2 \cdot 2^r$ and A_i, B_i are dyadic of order r, we have:

$$\mathbf{A} = \begin{pmatrix} A_0 & A_1 \\ A_1 & A_0 \end{pmatrix}, \mathbf{B} = \begin{pmatrix} B_0 & B_1 \\ B_1 & B_0 \end{pmatrix}.$$

$$\mathbf{C} = \mathbf{A} \cdot \mathbf{B} = \begin{pmatrix} A_0B_0 + A_1B_1 & A_1B_0 + A_0B_1 \\ A_1B_0 + A_0B_1 & A_0B_0 + A_1B_1 \end{pmatrix}$$

by the hypothesis, Theorem 2.4 and Theorem 2.2 we have that the conclusion holds. □

It is clear that a dyadic matrix is symmetric, which means that $M^T = M$. Another property is that given a dyadic matrix A, $-A = \mathbf{\Delta}(-\mathbf{a})$ is a dyadic matrix. Finally, we will introduce a relaxed notion of dyadicity, which will be useful throughout the thesis.

Definition 2.6. A *quasi-dyadic* matrix is a (possibly non-dyadic) block matrix whose blocks are dyadic submatrices, i.e., elements of $\mathbf{\Delta}\left(\mathcal{R}^n\right)^{d_1 \times d_2}$.

2.2 — Coding Theory

We can say that coding theory started as an engineering problem trying to solve communication problems such as what was the best way for a sender to encode the information and send it through a channel. It started with Shannon in the 1940's and later on it advanced by Golay and Hamming. Nowadays, we have several ramifications from ways to encode, decode and verify the data that we want to send through a communication channel. In this thesis, we are interested in error correction more specifically how we can apply this to cryptography. The background in coding theory follows closely the PhD thesis by Persichetti [Per12b].

We start by introducing the notion of linear code.

Definition 2.7. Let $\mathbb{F}_q$ be the finite field with q elements. An $[n, k]$ Linear Code $\mathcal{C}$ is a subspace of dimension k of the vector space $\mathbb{F}_q^n$.

Elements of the code are called *codewords*. Each message is represented as a vector of $\mathbb{F}_q^k$ and mapped to a unique codeword. The parameter n is the code length, k is the code dimension and the difference $n-k$ is the redundancy of the code. The redundancy is added since we are sending information through a noisy channel and we need a way of verifying if a codeword is correct and if it is not correct of correcting it.

Codes are usually studied in the context of the Hamming metric, determined by the distance defined below.

Definition 2.8. Let $\mathcal{C}$ be an $[n, k]$ linear code over $\mathbb{F}_q$. Let $\boldsymbol{x} = (x_1, \ldots, x_n), \boldsymbol{y} = (y_1, \ldots, y_n) \in \mathcal{C}$ be two codewords. The *Hamming Distance* $d_h(x, y)$ between the codewords is the number of positions in which they differ, that is

$$d_h(x, y) = \#\{i : x_i \neq y_i, 1 \leqslant i \leqslant n\}. \tag{2.4}$$

Definition 2.9. Let $\mathcal{C}$ be an $[n, k]$ linear code over $\mathbb{F}_q$. Let $\boldsymbol{x} = (x_1, \ldots, x_n) \in \mathcal{C}$ be a codeword. The *Hamming Weight* $wt_h(x)$ of the codeword is the number of non-zero positions, that is

$$wt_h(x) = \#\{i : x_i \neq 0, 1 \leqslant i \leqslant n\}. \tag{2.5}$$

Definition 2.10. Let $\mathcal{C}$ be an $[n, k]$ linear code over $\mathbb{F}_q$. The *Minimum Distance d* of $\mathcal{C}$ is the minimum of the distances among all the codewords, that is

$$d = \min\{d_h(x, y) : x, y \in \mathcal{C}, x \neq y\}. \tag{2.6}$$

The minimum distance of a code is important to determine its *error-correction* capabilities. Consider that a codeword $\mathbf{x}$ is transmitted over a noisy channel, and errors occur in w positions of the codeword. We represent this as an *error vector* $\mathbf{e}$ of weight w and $\mathbf{e}$ has non-zero positions exactly where the errors occur. Instead of receiving the codeword $\mathbf{x}$, we receive $\mathbf{z} = \mathbf{x} + \mathbf{e}$. We say that a code $\mathcal{C}$ is able to correct w errors if, for each codeword, it is possible to detect and correct any configuration of w errors.

Theorem 2.11. Let $\mathcal{C}$ be an $[n, k]$ linear code over $\mathbb{F}_q$ with minimum distance d. Then $\mathcal{C}$ is able to correct up to $w = \lfloor \frac{d-1}{2} \rfloor$ errors.

Proof. For every codeword $\mathbf{x} \in \mathcal{C}$ define the sphere of radius w centered in $\mathbf{x}$ as $S_x = \{\mathbf{z} \in \mathbb{F}_q^n : d_h(\mathbf{z}, \mathbf{x}) \leqslant w\}$. Now consider two spheres S_x and S_y for $\mathbf{x} \neq \mathbf{y}$ and let $\mathbf{z} \in S_x \cap S_y$. Then $d_h(\mathbf{z}, \mathbf{x}) \leqslant w$, hence $d_h(\mathbf{z}, \mathbf{x}) + d_h(\mathbf{z}, \mathbf{y}) \leqslant 2w$ and this is a contradiction since, by triangular inequality, $d_h(\mathbf{z}, \mathbf{x}) + d_h(\mathbf{z}, \mathbf{y}) \geqslant d_h(\mathbf{x}, \mathbf{y}) \geqslant d$. This shows that the two spheres are disjoint; hence, if the error vector added to a codeword has weight $\leqslant w$, the corresponding vector $\mathbf{z}$ belongs to a uniquely determined sphere and it is then possible to recover the codeword by decoding to the center of the sphere. □

An efficient way to represent linear codes is using matrices.

Definition 2.12. Let $\mathcal{C}$ be an $[n, k]$ linear code over $\mathbb{F}_q$. Let $\mathcal{B} = \{v_1, \ldots, v_k\}$ be a basis for the vector subspace determined by $\mathcal{C}$. The $k \times n$ matrix G having the vectors in $\mathcal{B}$ as rows is called *Generator Matrix* for $\mathcal{C}$, that is

$$G = \begin{pmatrix} v_1 \\ \vdots \\ v_k \end{pmatrix}. \tag{2.7}$$

The matrix G generates the code as a linear map: for each message $\mathbf{m} \in \mathbb{F}_q^k$ we obtain the corresponding codeword $\mathbf{m}G$. Of course, since the choice of basis is not unique, neither is the choice of generator matrix. In a more specific explanation, given a generator matrix G, then the matrix SG, where S is any invertible matrix, generates the same code. It is often possible to choose S in a particular way, so that $G = (I_k|M)$. This is called *systematic form* of the generator matrix.

Note that when using a generator matrix in systematic form each message appears in the first k positions of the corresponding codeword (i.e. the first k positions carry the information symbols).

We now provide several definitions that are important for the understanding of the remainder of the thesis.

Definition 2.13. Let $\mathcal{C}$ be an $[n, k]$ linear code over $\mathbb{F}_q$. The *Dual Code* of $\mathcal{C}$ is the set $\mathcal{C}^\perp = \{\mathbf{x} \in \mathbb{F}_q^n : \mathbf{x} \cdot \mathbf{y} = 0 \ \forall \mathbf{y} \in \mathcal{C}\}$.

Theorem 2.14. Let $\mathcal{C}$ be an $[n,k]$ linear code over $\mathbb{F}_q$. Then the dual code $\mathcal{C}^{\perp}$ is an $[n, n-k]$ linear code. Moreover, if $G = (I_k|M)$ is a generator matrix in systematic form for $\mathcal{C}$, then $H = (-M^T|I_{n-k})$ is a generator matrix for $\mathcal{C}^{\perp}$.

The matrix H is an important matrix for the code $\mathcal{C}$ as you can see in Definition 2.15.

Definition 2.15. Let $\mathcal{C}$ be an $[n,k]$ linear code over $\mathbb{F}_q$ and let $\mathcal{C}^{\perp}$ be its dual code. Any generator matrix for $\mathcal{C}^{\perp}$ is called parity-check matrix for $\mathcal{C}$.

The parity-check matrix describes the code as follows. Given $\mathbf{x} \in \mathbb{F}_q^n$:

$$\mathbf{x} \in \mathcal{C} \iff H\mathbf{x}^T = 0. \tag{2.8}$$

As its name suggests the parity-check matrix is used for checking if the codeword is correct, i.e., it checks if the codeword has not an error. Originally it was used for codes in which a single redundancy bit is added at the end of a codeword, the bit being a 0 if the codeword has an even number of 1's and a 1 otherwise. If a received word has an odd number of 1's, it is sure that at least an error has occurred.

The vector $H\mathbf{x}^T$ is called *syndrome* of $\mathbf{x}$, and gives its name to an error-correcting method, known as *syndrome decoding*. It works by splitting the code $\mathcal{C}$ in q^{n-k} cosets and then pre-computing a table containing the syndromes of all the corresponding coset leaders.

Algorithm 1: High level description of syndrome decoding.

```
Data: An $(n-k) \times n$ parity check matrix and $\mathbf{z} = \mathbf{x} + \mathbf{e} \in \mathbb{F}_q^n$
Result: The codeword $\mathbf{x}$
Compute the syndrome as $s = H\mathbf{z}^T$;
find the coset leader $l$ associated to $s$;
if $l$ is found, return $\mathbf{x} = \mathbf{z} - l$, else return $\perp$;
```

The syndrome decoding method succeeds as long as $w = wt_h(\mathbf{e})$ is within the correcting radius of the code, i.e. $w \leqslant \lfloor \frac{d-1}{2} \rfloor$, where d is the minimum distance of the code. In fact, since $\mathbf{x}$ is a codeword, we have $H\mathbf{z}^T = H\mathbf{x}^T + H\mathbf{e}^T = 0 + H\mathbf{e}^T = H\mathbf{e}^T$ and because its weight is within the correcting radius, $\mathbf{e}$ is the uniquely determined coset leader and it is possible to use the Algorithm 1 for finding the codeword $\mathbf{x}$.

We now move to a concept about codes that will be useful for understanding Chapter 4 of the thesis.

2.2.1 – Cyclic codes.

Definition 2.16. Let $\mathcal{C}$ be an $[n,k]$ linear code over $\mathbb{F}_q$. We call $\mathcal{C}$ cyclic if

$$\forall \mathbf{a} = (a_0, \dots, a_{n-1}) \in \mathcal{C} \Rightarrow \mathbf{a}' = (a_{n-1}, a_0, \dots, a_{n-2}) \in \mathcal{C}. \tag{2.9}$$

If the property holds, then all the right shifts, for any number of positions have to belong to $\mathcal{C}$ as well.

It is possible to use polynomial rings for building cyclic codes. In fact, it is natural to build a bijection between cyclic codes and ideals of a polynomial ring defined as $\mathbb{F}_q[x]/(x^n - 1)$. It is possible to identify the vector $(a_0, a_1, \dots, a_{n-1})$ with the polynomial $a_0 + a_1 x + \cdots + x^{n-1} a_{n-1}$, and then the right shift operation corresponds to the multiplication of the polynomial by x in the ring $\mathbb{F}_q[x]/(x^n - 1)$.

We can generate different ideals using different monic polynomials $g(x)$, such that $g(x)$ divides $x^n - 1$ as shown below. Each polynomial corresponds to a different cyclic code, and we therefore call g the generator polynomial of the code. From this polynomial we can produce a generator matrix and this will have a special form.

Definition 2.17. Let $g(x) \in \mathbb{F}_q[x]$ be monic of degree $n-k$ and a divisor of $x^n - 1$. Then g generates a cyclic $[n, k]$ code over $\mathbb{F}_q$, $\mathcal{B} = \{g(x), xg(x), \dots, x^{k-1}g(x)\}$ is a basis for $\mathcal{C}$, and we obtain the generator matrix

$$G = \begin{pmatrix} g(x) \\ xg(x) \\ \vdots \\ x^{k-1}g(x) \end{pmatrix}. \tag{2.10}$$

The matrix G will be in circulant form, where the i-th row corresponds to the cyclic right shift by i positions of the first row. A *circulant matrix* is a matrix in which every row is obtained as a cyclic right shift of the previous, see Section 2.2.5 for more details about circulant matrices.

Definition 2.18. Let q be a prime power and n, k positive integers such that $1 \leqslant k \leqslant n \leqslant q$. Let m be the multiplicative order of q modulo n, α a primitive n-th root of unity in $\mathbb{F}_{q^m}$ and $\mathbb{P}_{m,k}$ be the set of polynomials of degree $\leqslant k$ over $\mathbb{F}_{q^m}$. Fix $\mathbf{x} \in \mathbb{F}_q^n$ with $x_i \neq x_j$ and $\mathbf{y} \in \mathbb{F}_{q^m}^n$ with $y_i \neq 0$. Then the Generalized Reed-Solomon (GRS) Code of order $r = n - k$ is the code $\mathrm{GRS}_r(\mathbf{x}, \mathbf{y}) = \{(y_1 f(x_1), y_2 f(x_2), \dots, y_n f(x_n)) : f \in \mathbb{P}_{m,k}\}$.

2.2.2 – Alternant Codes. We now present the notion of alternant codes, that are defined as subfield subcodes of GRS codes.

Definition 2.19. Let $\mathcal{C}$ be an $[n, k]$ linear code over $\mathbb{F}_{q^m}$. The *subfield subcode* $\mathcal{C}|_{\mathbb{F}_q}$ of $\mathcal{C}$ over $\mathbb{F}_q$ is the vector space $\mathcal{C} \cap \mathbb{F}_q^n$.

The easiest way to obtain a subfield subcode is to use the trace construction.

Definition 2.20. Let $H = \{h_{i,j}\}$ be an $r \times n$ matrix over $\mathbb{F}_{q^m}$. Fix an ordered basis $E = \{e_1, \dots, e_m\}$ for $\mathbb{F}_{q^m}$ over $\mathbb{F}_q$ and the corresponding projection function $\phi_E : \mathbb{F}_{q^m} \to \mathbb{F}_q^m$ defined by $\phi_E(\alpha) = (a_1, \dots, a_m)^T$ for $\alpha = a_1 e_1 + \dots + a_m e_m$. We define the *Trace Matrix* $\mathcal{T}(H)$ as the $rm \times n$ matrix obtained by replacing each element $h_{i,j}$ with the column vector $\phi_E(h_{i,j})$; and the Co-Trace Matrix $\mathcal{T}'(H)$ as the $rm \times n$ matrix whose $(((l-1)r) + i, j)$ element is $\phi_E(h_{i,j})_l$, for $i = 1, \dots, r$, $j = 1, \dots, n$ and $l = 1, \dots, m$. Note that $\mathcal{T}'(H)$ is equivalent to $\mathcal{T}(H)$ by a left rotation. The code defined by $\mathcal{T}(H)$ is the trace code of the code defined by the parity-check matrix H.

MacWilliams and Sloane [MS77] show that the dual of $\mathcal{C}|_{\mathbb{F}_q}$ is the trace of the dual of $\mathcal{C}$. The generator matrix for the dual code is in fact a parity-check matrix for $\mathcal{C}$, in practice this means that we can build a parity-check matrix for the subfield subcode directly from $\mathcal{C}$.

Theorem 2.21. Let $\mathcal{C}$ be an $[n, k, d]$ linear code over $\mathbb{F}_{q^m}$ and H be a parity-check matrix for $\mathcal{C}$. Then the subfield subcode $\mathcal{C}|_{\mathbb{F}_q}$ is an $[n, k', d']$ linear code over $\mathbb{F}_q$, where $k' \geqslant n - m(n-k)$, $d' \geqslant d$ and $\hat{H} = \mathcal{T}(H)$ is a parity-check matrix for it.

The proof from Theorem 2.21 can be found in [Per12b].

Definition 2.22. Let $\mathrm{GRS}_r(\mathbf{x}, \mathbf{y})$ be a GRS code of order r over $\mathbb{F}_{q^m}$ for a certain prime power q and extension degree $m > 1$. The *Alternant Code* $A_r(\mathbf{x}, \mathbf{y})$ is the subfield subcode $\mathrm{GRS}_R(\mathbf{x}, \mathbf{y})|_{\mathbb{F}_q}$.

An alternant code $A_r(\mathbf{x}, \mathbf{y})$ has a special form for the parity-check matrix:

$$H(\mathbf{x}, \mathbf{y}) = \begin{pmatrix} y_1 & \cdots & y_n \\ y_1 x_1 & \cdots & y_n x_n \\ \vdots & \vdots & \vdots \\ y_1 x_1^{r-1} & \cdots & y_n x_n^{r-1} \end{pmatrix} \tag{2.11}$$

Definition 2.23. Let $A_r(\mathbf{x}, \mathbf{y})$ be an alternant code over $\mathbb{F}_q$ as defined in Definition 2.22. Suppose that one receives a vector with error $\mathbf{e}$ having $wt(\mathbf{e}) = w$ within the correction range, with error values $v_1, \dots, v_w$ in positions $p_1, \dots, p_w$. We call:

- *Error Locators* the elements $x_{p_1}, \dots, x_{p_w}$;
- *Error Locator Polynomial* the polynomial $\Lambda(z) = \prod_{i=1}^{w}(1 - x_{p_i} z)$; and
- *Error Evaluator Polynomial* the polynomial $\Omega(z) = \sum_{j=1}^{w} v_j y_{p_j} \prod_{\substack{1 \leqslant i \leqslant w \\ i \neq j}} (1 - x_{p_i} z)$.

The error positions are uniquely determined by the reciprocals of the roots of Λ. Once these are found, the error values are given by Equation 2.12.

$$v_j = \frac{\Omega(x_{p_j}^{-1})}{y_{p_j} \prod_{\substack{1 \leqslant i \leqslant w \\ i \neq j}} (1 - x_{p_i} x_{p_j}^{-1})} \tag{2.12}$$

Algorithm 2 gives a high level view for how it is possible to find and correct errors from a syndrome $\mathbf{s}$ using alternant decoding.

Algorithm 2: High level of alternant decoding algorithm.

```
Data: An $r \times n$ parity-check matrix $H(\mathbf{x}, \mathbf{y})$ and syndrome $\mathbf{s} \in \mathbb{F}_q^r$
Result: The error vector $\mathbf{e}$
Compute the corresponding polynomial from syndrome $\mathbf{s}$ as $S(z) = \sum_{i=0}^{r-1} s_i z^i$;
Use Euclidean algorithm for polynomials to solve the key equation

    $$\Omega(z) \equiv \Lambda(z) S(z) \mod z^r \tag{2.13}$$

  and retrieve $\Lambda$ and $\Omega$;
Find roots of $\Lambda$;
Build the error vector $\mathbf{e}$ having value $v_i$ in position $p_i$ for $i = 1, \dots, w$;
return $\mathbf{e}$;
```

We will present methods for finding roots in Chapter 5.

Among alternant codes there are several families but in this thesis we have interest in the following families of codes:

- Goppa codes
- Generalized Srivastava codes.

2.2.3 – Goppa codes. A binary Goppa code $\Gamma(L, g(z))$ is defined by a polynomial $g(z) = g_0 + g_1 z + \cdots + g_t z^t = \sum_{i=0}^{t} g_i z^i$ over $\mathbb{F}_{2^m}$ with degree t which is square free and a vector $L = (\alpha_0, \alpha_1, \ldots, \alpha_{n-1}) \in \mathbb{F}_{2^m}^n$, $\alpha_i \neq \alpha_j$, and $g(\alpha_i) \neq 0$ for all $\alpha_i \in L$. The ordered set L is known as support. For more details about algebraic codes, see [Ber15].

For any vector $c = (c_0, c_1 \ldots, c_{n-1}) \in \mathbb{F}_2^n$ we define the syndrome polynomial

$$S_c(z) = \sum_{i=1}^{n} \frac{c_i}{z + \alpha_i},$$

where $\frac{1}{z+\alpha_i}$ is the unique polynomial of degree $< t$ with $(z + \alpha_i)\frac{1}{z+\alpha_i} \equiv 1 \mod g(z)$.

Definition 2.24. The binary Goppa code $\Gamma(L, g(z))$ consists of all vectors $c \in \mathbb{F}_2^n$ such that

$$S_c(z) \equiv 0 \mod g(z). \tag{2.14}$$

The parameters of a code are the length n, dimension k and minimum distance d. In this thesis, we will use the notation $[n, k, d]-$Goppa to refer to a binary Goppa code with parameters n, k and d. A Goppa code $\Gamma(L, g(z))$ is a linear code over $\mathbb{F}_{2^m}$.

The length of a Goppa code is given by $n = |L|$, its dimension is $k \geqslant n - mt$, where $t = \deg(g)$, and its minimum distance $d \geqslant t + 1$. The syndrome polynomial $S_c(z)$ satisfies the following property:

$$S_c(z) \equiv \frac{w(z)}{\Lambda(z)} \mod g(z), \tag{2.15}$$

for some $w(z) \in \mathbf{F}_2[z]$ of degree $\deg(w) < \deg(\Lambda)$ and $\Lambda(z) = \prod_{i=1}^{t}(1 + z\alpha_i)$ is called the error locator polynomial (ELP) and the roots of this polynomial give the error positions.

2.2.4 – Generalized Srivastava Codes. Srivastava Codes are an alternative to Goppa Codes that can be used in code-based cryptography. They originated from an unpublished work of J.N. Srivastava from 1967. In 1972, Helgert [Hel72] used Srivastava's original work to propose an alternative for BCH codes[1].

Let H be the parity-check matrix in the form of Equation 2.11. For every $r \times r$ invertible matrix S, the matrix SH is an equivalent parity-check matrix. It is then clear that an

[1] For more details see [CL04].

alternative form for $H(\mathbf{x}, \mathbf{y})$ is

$$\begin{aligned} H(\mathbf{x}, \mathbf{y}) &= \begin{pmatrix} s_{1,1} & \cdots & s_{1,r} \\ s_{2,1} & \cdots & s_{2,r} \\ \vdots & \vdots & \vdots \\ s_{r,1} & \cdots & s_{r,r} \end{pmatrix} \begin{pmatrix} y_1 & \cdots & y_n \\ y_1 x_1 & \cdots & y_n x_n \\ \vdots & \vdots & \vdots \\ y_1 x_1^{r-1} & \cdots & y_n x_n^{r-1} \end{pmatrix} \\ &= \begin{pmatrix} y_1 f_1(x_1) & \cdots & y_n f_1(x_n) \\ y_1 f_2(x_1) & \cdots & y_n f_2(x_n) \\ \vdots & \vdots & \vdots \\ y_1 f_r(x_q) & \cdots & y_n f_r(x_n) \end{pmatrix} \end{aligned} \tag{2.16}$$

where $f_i(x) = s_{i,1} + s_{i,2}x + s_{i,3}x^2 + \cdots + s_{i,r}x^{r-1}$ for each $i = 1, \ldots, r$.

Definition 2.25. Fix a finite field $\mathbb{F}_{q^m}$with $m > 1$. Let $\alpha_1, \ldots, \alpha_n, w_1, \ldots, w_s$ be $n + s$ distinct elements of $\mathbb{F}_{q^m}$, and $z_1, \ldots, z_n$ be non-zero elements of $\mathbb{F}_{q^m}$. The *Generalized Srivastava* (GS) code of order $r = st$ and length n is the alternant code $\mathcal{A}_r(\mathbf{x}, \mathbf{y})$ defined by the parity-check matrix (Equation 2.16) having

$$f_{(l-1)t+k}(x) = \frac{\prod_{j=1}^{s}(x - w_j)^t}{(x - w_l)^k} \text{ for } l = 1, \ldots, s \text{ and } k = 1, \ldots, t$$

$$y_i = \frac{z_i}{\prod_{j=1}^{s}(\alpha_i - w_j)^t} \text{ for } i = 1, \ldots, n.$$

This implies

$$y_i f_{(l-1)t+k}(\alpha_i) = \frac{z_i}{(\alpha_i - w_l)^k} \tag{2.17}$$

for $i = 1, \ldots, n$, $l = 1, \ldots, s$ and $k = 1, \ldots, t$. It is then possible to deduce a standard form for the parity-check matrix of GS codes as

$$H = \begin{pmatrix} L_1 \\ L_2 \\ \vdots \\ L_s \end{pmatrix}, \tag{2.18}$$

where each block L_i is

$$L_i = \begin{pmatrix} \frac{z_1}{\alpha_1 - w_i} & \cdots & \frac{z_n}{\alpha_n - w_i} \\ \frac{z_1}{(\alpha_1 - w_i)^2} & \cdots & \frac{z_n}{(\alpha_n - w_i)^2} \\ \vdots & \vdots & \vdots \\ \frac{z_1}{(\alpha_1 - w_i)^t} & \cdots & \frac{z_n}{(\alpha_n - w_i)^t} \end{pmatrix}. \tag{2.19}$$

Since GS codes are alternant codes, the parameters are length $n \leqslant q^m - s$, dimension $k \geqslant n - mst$ and minimum distance $d \geqslant st + 1$. By analogy with BCH codes, GS codes are called primitive if the α_i's are chosen to be all the elements of $\mathbb{F}_{q^m}$ apart from the w_i's. In this case the code length is exactly $n = q^m - s$.

GS codes are a large family of codes that includes other families as a special case. For example, when $m = 1$ these are called Gabidulin codes. Moreover, it is easy to prove that every GS code with $t = 1$ is a Goppa code. This makes GS codes good for several applications in particular the usage in cryptography. One of the first uses of GS codes in cryptography came in [Per12a], where Persichetti proposed a compact McEliece cryptosystem using Quasi-Dyadic Srivastava codes. Later on, DAGS [BBB+18] used the same codes for a key encapsulation mechanism safe against quantum computers.

2.2.5 – Circulant Matrices and Quasi-Cyclic Codes. Recall that a *circulant matrix* is a matrix in which every row is obtained as a right cyclic shift of the previous. Equation (2.20) shows a circulant matrix of size[2] p.

$$\mathbf{C}_p = \begin{pmatrix} t_0 & t_1 & \cdots & t_{p-1} \\ t_{p-1} & t_0 & \cdots & t_{p-2} \\ \vdots & & \ddots & \vdots \\ t_1 & t_2 & \cdots & t_0 \end{pmatrix} \tag{2.20}$$

Circulant $p \times p$ matrices over $\mathbb{F}_{q^m}$ form a ring that we will denote by $\mathcal{C}_p(\mathbb{F}_{q^m})$. Its cardinality is $|\mathcal{C}_p(\mathbb{F}_{q^m})| = q^{mp}$.

Proposition 2.26. Let $x^p - 1 = p_1^{\alpha_1}(x) \cdot \cdots \cdot p_\tau^{\alpha_t}(x)$ be the factorization of $x^p - 1$ over $\mathbb{F}_{q^m}$ into powers of irreducible factors. The number of invertible circulant matrices in $\mathcal{C}_p(\mathbb{F}_{q^m})$ is equal to $\prod_{i=1}^{\tau}(q^{m\cdot d_i\alpha_i} - q^{m\cdot d_i(\alpha_i-1)})$, where d_i is the degree of $p_i(x)$ in the factorization of $x^p - 1$.

Proof. The ring $\mathcal{C}_p(\mathbb{F}_{q^m})$ is isomorphic to $\mathbb{F}_{q^m}[x]/\langle x^p - 1\rangle$ where the matrix $\mathbf{C}_p$ corresponds to the polynomial $t(x) = \sum_{i=0}^{p-1} t_i x^i$. From the factorization

$$x^p - 1 = p_1^{\alpha_1}(x) \cdot \cdots \cdot p_\tau^{\alpha_\tau}(x),$$

and the Chinese Remainder Theorem, $\mathbb{F}_{q^m}[x]/\langle x^p - 1\rangle$ is isomorphic to the direct product:

$$\mathbb{F}_{q^m}[x]/\langle x^p - 1\rangle \cong \mathbb{F}_{q^m}[x]/\langle p_1^{\alpha_1}(x)\rangle \times \cdots \times \mathbb{F}_{q^m}[x]/\langle p_\tau^{\alpha_t}(x)\rangle.$$

The number of invertible elements in $\mathbb{F}_{q^m}[x]/\langle p_i^{\alpha_i}(x)\rangle$ is $q^{m\cdot d_i\alpha_i} - q^{m\cdot d_i(\alpha_i-1)}$ where d_i is the degree of $p_i(x)$. Now it is easy to count the number of invertible elements in $\mathbb{F}_{q^m}[x]/\langle x^p - 1\rangle$. It is precisely the product of the number of invertible elements in each $\mathbb{F}_{q^m}[x]/\langle p_i^{\alpha_i}(x)\rangle$, i.e., $\prod_{i=1}^{\tau}(q^{m\cdot d_i\alpha_i} - q^{m\cdot d_i(\alpha_i-1)})$.

Note that when $\alpha_1 = \alpha_2 = \cdots = \alpha_\tau = 1$, $\mathbb{F}_{q^m}[x]/\langle x^p - 1\rangle$ factors into a direct product of fields, and our formula turns into $\prod_{i=1}^{\tau}(q^{m\cdot d_i} - 1)$.

□

[2] A circulant matrix can be defined as a special case of Toeplitz matrix; for more details about Toeplitz matrices see [Gra06].

A *quasi-cyclic code* is a code with generator matrix of the form

$$G = \begin{pmatrix} \mathbf{C}_{11} & \mathbf{C}_{12} & \cdots & \mathbf{C}_{1n_0} \\ \mathbf{C}_{21} & \mathbf{C}_{22} & \cdots & \mathbf{C}_{2n_0} \\ \vdots & & \ddots & \vdots \\ \mathbf{C}_{k_0 1} & \mathbf{C}_{k_0 2} & \cdots & \mathbf{C}_{k_0 n_0} \end{pmatrix} \tag{2.21}$$

where each matrix $\mathbf{C}_{ij}$ is a circulant matrix of the form 2.20.

2.2.6 – Rank Metric Codes.

Definition 2.27. Let $\boldsymbol{x} = (x_1, \ldots, x_n) \in \mathbb{F}_{q^m}^n$ and let $(\beta_1, \ldots, \beta_m) \in \mathbb{F}_{q^m}^m$ be a basis of $\mathbb{F}_{q^m}$ viewed as an m-dimensional vector space over $\mathbb{F}_q$. Each coordinate x_j is associated to a vector of $\mathbb{F}_{q^m}^m$ in this basis: $x_j = \sum_{i=1}^m m_{ij}\beta_i$. The $m \times n$ matrix associated to $\boldsymbol{x}$ is given by $\boldsymbol{M}(\boldsymbol{x}) = (m_{ij})_{\substack{1 \leqslant i \leqslant m \\ 1 \leqslant j \leqslant n}}$.

The rank weight $||\boldsymbol{x}||$ of vector $\boldsymbol{x}$ is defined as:

$$||\boldsymbol{x}|| \overset{\text{def}}{=} \text{Rank}\, \boldsymbol{M}(\boldsymbol{x}),$$

where Rank $\boldsymbol{M}(\boldsymbol{x})$ is the usual rank of a binary matrix.

Similar to how the Hamming weight leads to the Hamming distance, we can define a distance based on rank weight. We can define the distance between two elements as $d(\boldsymbol{x}, \boldsymbol{y}) = ||\boldsymbol{x} - \boldsymbol{y}||$ where $x, y \in \mathbb{F}_{q^m}^n$.

We define an $\mathbb{F}_{q^m}$-linear code using rank metric as:

Definition 2.28. An $\mathbb{F}_{q^m}$-linear code $\mathcal{C}$ of dimension k and length n is a subspace of dimension k of $\mathbb{F}_{q^m}^n$ endowed with the rank metric. It is denoted by $[n, k]_{q^m}$. The code $\mathcal{C}$ can be represented by two equivalent ways:

- by a generator matrix $G \in \mathbb{F}_{q^m}^{k \times n}$, each row of G is an element of a basis of $\mathcal{C}$,

$$\mathcal{C} = \{\boldsymbol{x}G : \boldsymbol{x} \in \mathbb{F}_{q^m}^k\}.$$

- by a parity-check matrix $H \in \mathbb{F}_{q^m}^{(n-k) \times n}$, each row of H determines a parity check equation verified by the elements of $\mathcal{C}$:

$$\mathcal{C} = \{\boldsymbol{x} \in \mathbb{F}_{q^m}^n : H\boldsymbol{x}^\top = 0\}.$$

The systematic form of a generator matrix G is in the form $(\mathbf{I}_k | \boldsymbol{A})$ where $\mathbf{I}_k$ is the identity matrix. The systematic form of a parity-check matrix H is similar to G and it can be described as $(-\boldsymbol{A}^\top | \mathbf{I}_{n-k})$, where $\mathbf{I}_{n-k}$ is the identity matrix.

Definition 2.29. Let $\boldsymbol{x} = (x_1, \ldots, x_n) \in \mathbb{F}_{q^m}^n$. The support E of $\boldsymbol{x}$, denoted by $\mathsf{supp}(\boldsymbol{x})$ is the $\mathbb{F}_q$-subspace of $\mathbb{F}_{q^m}$ generated by the coordinates of $\boldsymbol{x}$, i.e.,

$$E = \langle x_1, \ldots, x_n \rangle_{\mathbb{F}_q}$$

and the dimension equals to the rank weight $||\boldsymbol{x}|| = \dim(E)$.

2.2.7 – LRPC Codes. A *Low-Rank Parity-Check (LRPC) code* $\mathcal{C}$ over $\mathbb{F}_{q^m}$ of length n, dimension k and rank d is defined as below.

Definition 2.30. A rank-d code is described by an $(n-k) \times n$ parity-check matrix $H = \{h_{i,j}\} \in \mathbb{F}_{q^m}^{(n-k)\times n}$, whose coefficients $h_{i,j}$ generate a subspace of $\mathbb{F}_{q^m}$ of dimension at most d. More precisely, each coefficient $h_{i,j}$ can be written as

$$h_{i,j} = \sum_{l=1}^{d} h_{i,j,l} F_l, h_{i,j,l} \in \mathbb{F}_q, \tag{2.22}$$

where each $F_i \in \mathbb{F}_{q^m}$, and $F = \langle F_1, F_2, \cdots, F_d \rangle$ is an $\mathbb{F}_q$ subspace of $\mathbb{F}_{q^m}$ of dimension at most d generated by the basis $\{F_1, F_2, \cdots, F_d\}$.

2.2.7.1 – Decoding of LRPC codes. Consider an LRPC code with parity-check matrix H of length n, dimension k and rank d, with basis $F = \{F_1, \cdots, F_d\}$. Let $\boldsymbol{e} = \{e_i\} \in \mathbb{F}_{q^m}^n$ be a vector of rank r, let the support $\mathrm{supp}(\boldsymbol{e})$ of $\boldsymbol{e}$ have basis $E = \{E_1, \cdots, E_r\}$. Considering the matrix representation, the vector $\boldsymbol{e}$ can be described as a matrix $\bar{\mathbf{E}} = \{e_{i,j}\}$, with $i \in [1;n]$, $j \in [1;r]$, such that

$$e_i = \sum_{j=1}^{r} e_{i,j} E_j, \quad e_{i,j} \in \mathbb{F}_q. \tag{2.23}$$

Let $\mathbf{s} \in \mathbb{F}_{q^m}^{n-k}$ be the *syndrome* of $\boldsymbol{e}$ with respect to H, i.e. $H\boldsymbol{e}^\top = \mathbf{s}$. Decoding consists in recovering $\boldsymbol{e}$, from the knowledge of $\mathbf{s}$. A decoding procedure, specific for the case of LRPC codes, has been proposed in [GMRZ13], and is shown in Algorithm 3. Under proper conditions, the syndrome equation can be rewritten as a linear system whose unknowns are nr scalars in $\mathbb{F}_q$. Indeed, for the i-th coordinate of $\mathbf{s}$, we have

$$s_i = \sum_{j=1}^{n} h_{i,j} e_j = \sum_{j=1}^{n} \left(\sum_{l=1}^{d} h_{i,j,l} F_l \right) \left(\sum_{v=1}^{r} e_{j,u} E_u \right) \tag{2.24}$$

$$= \sum_{l=1}^{d} \sum_{u=1}^{r} F_l E_u \left(\sum_{j=1}^{n} h_{i,j,l} e_{j,u} \right). \tag{2.25}$$

Then, by considering Equation 2.24 for all $i \in [1; n-k]$, the syndrome equation can be rewritten as

$$\mathbf{s}' = \mathbf{A}_H \boldsymbol{e}'^\top, \tag{2.26}$$

where $\mathbf{s}' \in \mathbb{F}_q^{(n-k)rd}$, $\mathbf{A}_H \in \mathbb{F}_q^{(n-k)rd \times nr}$ and $\boldsymbol{e}' \in \mathbb{F}_q^{nr}$.

Essentially, the above equation corresponds to the writing of the syndrome equation in the base field $\mathbb{F}_q$. In particular, $\mathbf{s}'$ contains the coefficients of the syndrome in the basis $\{F_i E_j\}_{\substack{i \leqslant i \leqslant d \\ 1 \leqslant j \leqslant r}}$, while $\mathbf{A}_H$ and $\boldsymbol{e}'$ are obtained through a rewriting of H and $\boldsymbol{e}$. Then, decoding can be performed through Algorithm 3.

The output of the algorithm corresponds to the vector $\boldsymbol{e}'$, that is, to the coefficients of $\boldsymbol{e}$ in the basis E.

Note that Algorithm 3 is characterized by a certain failure probability, which can be estimated according to the system parameters. In particular, decoding failures can happen only because of the following three events [GMRZ13].

Algorithm 3: Decoding of LRPC codes.

```
Data: $\mathbf{s} \in \mathbb{F}_{q^m}^{n-k}$, $\mathbf{s}' \in \mathbb{F}_q^{(n-k)rd}$, $\mathbf{A}_H \in \mathbb{F}_q^{(n-k)rd \times nr}$, basis $\{F_i\}_{1 \leqslant i \leqslant d}$
Result: $\mathbf{e}' \in \mathbb{F}_q$ in the basis $E$
$S \leftarrow \langle s_1, s_2, \dots, s_{n-k} \rangle$ ;                    // Syndrome space
for $i \leftarrow 1$ to $d$ do
    $S_i \leftarrow F_i^{-1} S$;
end
$E \leftarrow \bigcap_{j=1}^{d} S_j$;                            // Compute the error support
$\{E_1, \dots, E_r\} \leftarrow$ basis for $E$;
Solve $\mathbf{s}' = \mathbf{A}_H \mathbf{e}'^{\top}$ ; // Find the coefficients of $\mathbf{e}'$ in the basis $E$
return $\mathbf{e}'$;
```

a) Case of $\dim(\langle EF \rangle) < rd$: this happens with probability $P_1 = \frac{d}{q^{m-rd}}$ (see [GMRZ13, Sec. 3, Prop. 1]).
b) Case of $E \neq \bigcap_{i=1}^{d} S_i$: when $m > rd + 8$, this happens with probability $P_2 \ll 2^{-30}$ (see [GMRZ13, Sec. 3, Remark 3]).
c) Case of $\dim(S) < rd$: this happens with probability $P_3 = \frac{1}{q^{n-k+1-rd}}$ (see [GMRZ13, Sec. 5, Prop. 4]).

For parameters of practical interest, we usually have $P_1, P_2 \ll P_3$: this fact is crucial for the success of the attack presented in Chapter 6.

2.2.8 – McEliece cryptosystem. In this part, we describe the three important algorithms of the McEliece cryptosystem [McE78], i.e., key generation, message encryption, and message decryption. To give a practical explanation, we describe the McEliece scheme based on binary Goppa codes. However, it can be used with any q-ary Goppa code or Generalized Srivastava codes with small modifications as shown by [MB09, BLP10] and the work in Chapter 4.

Algorithm 4 is the key generation of McEliece. First, it starts by generating a binary Goppa polynomial $g(z)$ of degree t, which can be an irreducible Goppa polynomial. Second, it generates the support L as an ordered subset of $\mathbb{F}_{2^m}$ satisfying the root condition. Third, the computation of the systematic form of $\hat{H}$ is done using the Gauss-Jordan elimination algorithm, if possible. Steps four, five, and six compute the generator matrix from the previous systematic matrix and return secret and public key. Biswas and Sendrier [BS08] show that the safest method to obtain the public matrix is simply to compute the systematic form of the private matrix.

Algorithm 5 shows the encryption process of McEliece. The process is simple and efficient, requiring only a random vector e with $w_h(e) \leqslant t$ and a multiplication of a vector by a matrix.

Algorithm 6 gives the decryption part of McEliece. This algorithm consists of the removal of the applied errors using a decoding algorithm. First, we compute the syndrome polynomial $S_c(z)$. Second, we recover the error vector e from the syndrome polynomial. Finally, we can recover the plaintext m computing $c \oplus e$, i.e., the exclusive-or of the ciphertext and the error vector.

Note that this model should not be used in practice since c has an almost unmodified

Algorithm 4: McEliece key generation.

```
Data: $t, k, n, m$ as integers.
Result: pk as public key, sk as secret key.
Select a random binary Goppa polynomial $g(z)$ of degree $t$ over $\mathbb{F}_{2^m}$;
Randomly choose $n$ distinct elements of $\mathbb{F}_{2^m}$ that are not roots of $g(z)$ as the
  support $L$;
Compute the $k \times n$ parity-check matrix $\hat{H}$ according to $L$ and $g(z)$;
Bring $H$ to systematic form: $H_{sys} = [I_{k-n}|H']$ if possible;
Compute generator matrix $G$ from $H_{sys}$;
return $sk = (L, g(z)), pk = (G)$;
```

Algorithm 5: McEliece encryption.

```
Data: Public key $pk = G$, message $m \in \mathbb{F}_2^k$.
Result: $c$ as ciphertext of length $n$.
Choose randomly an error vector $e$ of length $n$ with $w_h(e) \leqslant t$;
Compute $c = (m \cdot G) \oplus e$;
return $c$;
```

copy of m in the first k positions. In modern KEM versions of McEliece, $m \in \mathbb{F}_2^k$ is a random bit string used to compute a session key using a hash function. Hence, there is no intelligible information in seeing the first k positions of m with almost no error, see Section 2.2.9 for details.

Algorithm 6: McEliece decryption.

```
  Data: $c$ as ciphertext of length $n$, secret key $sk = (L, g(z))$.
  Result: Message $m$.
1 Compute the syndrome polynomial $S_c(z) = \sum \frac{c_i}{z+\alpha_i} \mod g(z)$;
2 Compute $\tau(z) = \sqrt{S_c^{-1}(z) + z} \mod g(z)$;
3 Compute $b(z)$ and $a(z)$, so that $b(z)\tau(z) = a(z) \mod g(z)$, such that
    $\deg(a) \leqslant \lfloor \frac{t}{2} \rfloor$ and $\deg(b) \leqslant \lfloor \frac{t-1}{2} \rfloor$;
4 Compute the error locator polynomial $\Lambda(z) = a^2(z) + zb^2(z)$ and $\deg(\Lambda) \leqslant t$;
5 The positions in $L$ of the roots of $\Lambda(z)$ define the error vector $e$;
6 Compute the plaintext $m = c \oplus e$;
7 return $m$;
```

In the decryption algorithm, steps 2-5 are the description of Patterson's algorithm [Pat75]. This same strategy can be used in schemes that make use of the Niederreiter cryptosystem [CR88]. These schemes differ in their public-key structure, encryption, and decryption step, but both of them, in the decryption steps, decode the message from the syndrome.

The roots of the ELP can be acquired with different methods. Although these methods can be implemented with different forms, it is essential that the implementations do not leak any timing information about their execution. This leakage can lead to a side-channel attack using time differences in the decryption algorithm, as we explore in a scheme in Chapter 5.

2.2.9 – Niederreiter cryptosystem. In this part, we describe a Key Encapsulation Mechanism (KEM) version of the Niederreiter cryptosystem [Nie86]. The key generation works essentially as in Algorithm 4. The only difference is, that the output is the parity-check matrix in systematic form instead of the generator matrix. The encapsulation algorithm takes a public key and returns a ciphertext and a symmetric key. The latter can be used for encryption of data using symmetric ciphers. The ciphertext is sent to the other party. The encapsulation for the Niederreiter cryptosystem is given as Algorithm 7.

Algorithm 7: Niederreiter key encapsulation.

```
Data: Public key $pk = H$.
Result: ciphertext $c \in \mathbb{F}_2^{n-k}$, symmetric key $K$.
Choose randomly an error vector $e$ of length $n$ with $w_h(e) \leqslant t$;
Compute $c = (H \cdot e)$;
Compute $K = \mathrm{Hash}(e)$;
return $c, K$;
```

The decapsulation takes a ciphertext and a secret key and produces a symmetric key K. For honestly generated ciphertexts the symmetric key K output during encapsulation is the same as the one output by the decapsulation algorithm. The steps of the decapsulation follow closely the McEliece decryption process and are given as Algorithm 8.

Algorithm 8: Niederreiter key decapsulation.

```
Data: ciphertext $c \in \mathbb{F}_2^{n-k}$ , secret key $sk = (L, g(z))$.
Result: Symmetric key $K$.
Compute the syndrome polynomial $S_c(z) = \sum \frac{c_i}{z+\alpha_i} \mod g(z)$;
Compute $\tau(z) = \sqrt{S_c^{-1}(z) + z} \mod g(z)$;
Compute $b(z)$ and $a(z)$, so that $b(z)\tau(z) = a(z) \mod g(z)$, such that
  $\deg(a) \leqslant \lfloor \frac{t}{2} \rfloor$ and $\deg(b) \leqslant \lfloor \frac{t-1}{2} \rfloor$;
Compute the error locator polynomial $\Lambda(z) = a^2(z) + zb^2(z)$ and $\deg(\Lambda) \leqslant t$;
The positions in $L$ of the roots of $\Lambda(z)$ define the error vector $e$;
Compute the symmetric key $K = \mathrm{Hash}(e)$;
return $K$;
```

2.2.10 – LRPC Cryptosystems. LRPC cryptosystems were introduced in [GMRZ13], where the authors first present the low-rank parity-check codes and their application in cryptography. In [GMRZ13], the authors describe a McEliece type of cryptosystem but the Niederreiter version can be used as well. The key generation, encryption and decryption of the typical LRPC cryptosystem are summarized in Figure 2.1.

Figure 2.1 and the LRPC cryptosystem provide a general framework for schemes based on LRPC codes. However, the usual setting in practical schemes is to use specific types

1 **Key generation:** Choose a random LRPC code over $\mathbb{F}_{q^m}$ of low rank d with support F and parity check $(n-k)\times n$ matrix H, generator matrix G and decoding matrix D_H which can correct errors of rank r and a random invertible $(n-k)\times(n-k)$ matrix R.
Secret Key: the low rank matrix H, the masking matrix R.
Public Key: the matrix $G' = RG$.

2 **Encryption:** Translate the message m into a word $\mathbf{x}$, generate $\boldsymbol{e} \in \mathbb{F}_{q^m}$ randomly with rank r. Compute $\mathbf{c} = \mathbf{x}G' + \boldsymbol{e}$.

3 **Decryption:** Compute syndrome $\mathbf{s} = H\mathbf{c}^T$, recover the error vector $\boldsymbol{e}$ by decoding the LRPC code, then compute $\mathbf{x}G' = \mathbf{c} - \boldsymbol{e}$ and $\mathbf{x}$.

Figure 2.1: LRPC cryptosystem.

of LRPC codes. The motivation is that these allow for shorter keys. The specific types of LRPC codes used include quasi-cyclic codes, as for example in [GRSZ14] (and later also used in McNie [KKG+18] and Ouroboros-R [MAAB+17]), and ideal codes (which are a generalization of LRPC codes and used in LAKE [ABG+17a], Locker [ABG+17b] and Rollo [MAAB+19]). All of the previous cryptosystems show clear advantages over cryptosystems in the Hamming metric – for the same level of security, the keys are orders of magnitude smaller. For instance, the public key in Classic McEliece is 132KB while Rollo-II has a public key of size 2.4KB. Furthermore, ideal codes (with additional assumptions) have been used in the construction of the signature scheme Durandal [ABG+19] – showing once more the advantage over the Hamming metric where the construction of efficient signature schemes is still a problem.

Chapter 3

Fast Multiplication and Inversion in Dyadic Matrices

In this chapter, we consider efficient methods for computing the multiplication of two dyadic matrices and the inversion of a dyadic matrix. In fact, we mentioned in Subsection 2.1.1 properties about the arithmetic of dyadic matrices and showed that multiplication of two dyadic matrices can be computed from their signatures. Similarly, inversion of a dyadic matrix if it is not singular can be computed from the signature.[1] In particular, we analyze three different algorithms and provide estimations for their complexities; we then compare the performance of the various algorithms. For ease of notation, we will refer to the two $n \times n$ matrices that we want to multiply simply as $\mathbf{A}$ and $\mathbf{B}$, with $\mathbf{a} = [a_0, a_1, \dots, a_{n-1}]$ and $\mathbf{b} = [b_0, b_1, \dots, b_{n-1}]$ being the respective signatures. Maintaining the same notation, the product matrix $\mathbf{C} = \mathbf{AB}$, which is also dyadic, will have signature $\mathbf{c} = [c_0, c_1, \dots, c_{n-1}]$. In particular, we focus on the special case of quasi-dyadic matrices with elements belonging to a field $\mathbb{F}$ of characteristic 2.

3.1 — Standard Multiplication

The first algorithm we analyze is described in Algorithm 9; we refer to it as the *standard multiplication*. The element of $\mathbf{C}$ in position (i, j) is obtained as the multiplication between the i-th row of $\mathbf{A}$ and the j-th column of $\mathbf{B}$. Since dyadic matrices are symmetric, this is equivalent to the inner product between the i-th row of $\mathbf{A}$ and the j-th one of $\mathbf{B}$. The signature $\mathbf{c}$ (i.e., the first row of $\mathbf{C}$) is obtained by inner products involving only $\mathbf{a}$ (i.e., the first row of $\mathbf{A}$) and $\mathbf{b}$. Thus, we can just construct the rows of $\mathbf{B}$, by permuting

[1] This chapter is a joint work with Paulo S.L.M. Barreto, Edoardo Persichetti and Paolo Santini and it was presented at Mathcrypt 2018 [BBPS18]

the elements in **b**, and then computing the inner products.

Algorithm 9: Standard multiplication of dyadic matrices

```
Data: $r \in \mathbb{N}$, $n = 2^r$ and $\mathbf{a}, \mathbf{b} \in \mathbb{F}^n$.
Result: $\mathbf{c} \in \mathbb{F}^n$ such that $\mathbf{\Delta}(\mathbf{c}) = \mathbf{\Delta}(\mathbf{a})\mathbf{\Delta}(\mathbf{b})$.
$\mathbf{c} \leftarrow$ vector of length $n$, initialized with null elements;
$c_0 \leftarrow a_0 \cdot b_0$;
for $i \leftarrow 1$ to $n - 1$ do
    $c_0 \leftarrow c_0 + a_i b_i$;
    $i^{(b)} \leftarrow$ binary representation of $i$, using $r$ bits;
    for $\{j = 0, 1, \cdots, n - 1\}$ do
        $j^{(b)} \leftarrow$ binary representation of $j$, using $r$ bits;
        $l^{(b)} \leftarrow i^{(b)} \oplus j^{(b)}$;
        $l \leftarrow$ conversion of $l^{(b)}$ into an integer;
        $c_i \leftarrow c_i + a_i b_l$;
    end
end
return $\mathbf{c}$;
```

The complexity of the algorithm is due to two different types of operations:

1. In order to construct the rows of **B**, we need the indices of the corresponding permutations. Each index is computed as the modulo 2 sum of two binary vectors of length r, so it can be obtained with a complexity of r binary operations. Thus, considering that we need to repeat this operation for $2^r - 1$ rows (for the first one, no permutation is needed), the complexity of this procedure can be estimated as $r \cdot 2^r \cdot (2^r - 1)$.
2. Each element of **c** is obtained as the inner product between two vectors of 2^r elements, assuming values in $\mathbb{F}$. This operation requires 2^r multiplications and $2^r - 1$ sums in $\mathbb{F}$. If we denote as $C_{\mathtt{mult}}$ and $C_{\mathtt{sum}}$ the costs of, respectively, a multiplication and a sum in $\mathbb{F}$, the total number of binary operations needed to compute 2^r inner products can be estimated as $2^{2r} \cdot C_{\mathtt{mult}} + (2^{2r} - 2^r) \cdot C_{\mathtt{sum}}$.

The complexity of a standard multiplication between two dyadic signatures can be estimated as:

$$C_{\mathtt{std}} = r \cdot \left(2^{2r} - 2^r\right) + 2^{2r} \cdot C_{\mathtt{mult}} + (2^{2r} - 2^r) \cdot C_{\mathtt{sum}} \tag{3.1}$$

3.2 — Dyadic Convolution

Definition 3.1. The *dyadic convolution* of two vectors $\mathbf{a}, \mathbf{b} \in \mathcal{R}^n$, denoted by $\mathbf{a} \vartriangle \mathbf{b}$, is the unique vector of $\mathcal{R}$ such that $\mathbf{\Delta}(\mathbf{a} \vartriangle \mathbf{b}) = \mathbf{\Delta}(\mathbf{a})\mathbf{\Delta}(\mathbf{b})$.

Of particular interest to us is the case where $\mathcal{R}$ is actually a field $\mathbb{F}$. Dyadic matrices over $\mathbb{F}$ form a commutative subring $\mathbf{\Delta}(\mathbb{F}^n) \subset \mathbb{F}^{n \times n}$, and this property gives rise to efficient arithmetic algorithms to compute the dyadic convolution. In particular, we here consider the fast Walsh-Hadamard transform (FWHT), which is well known [Gul73] but seldom found in a cryptographic context. We describe it here for ease of reference. We firstly recall the FWHT for the case of a field $\mathbb{F}$ such that $\text{char}(\mathbb{F}) \neq 2$ as in shown [FA76], and then describe how this technique can be generalized to consider also the case of $\text{char}(\mathbb{F}) = 2$ (which, again, is the one we are interested in).

Definition 3.2. Let $\mathbb{F}$ be a field with $\mathrm{char}(\mathbb{F}) \neq 2$. The *Sylvester-Hadamard matrix* $\mathbf{H}_r \in \mathbb{F}^n$ is recursively defined as

$$\begin{aligned} \mathbf{H}_0 &= \begin{pmatrix} 1 \end{pmatrix}, \\ \mathbf{H}_r &= \begin{pmatrix} \mathbf{H}_{r-1} & \mathbf{H}_{r-1} \\ \mathbf{H}_{r-1} & -\mathbf{H}_{r-1} \end{pmatrix}, r > 0. \end{aligned}$$

One can show by straightforward induction that $\mathbf{H}_r^2 = 2^r \mathbf{I}_r$ and hence $\mathbf{H}_r^{-1} = 2^{-r}\mathbf{H}_r$, which can also be expressed recursively as

$$\begin{aligned} \mathbf{H}_0^{-1} &= \begin{pmatrix} 1 \end{pmatrix}, \\ \mathbf{H}_r^{-1} &= \frac{1}{2}\begin{pmatrix} \mathbf{H}_{r-1}^{-1} & \mathbf{H}_{r-1}^{-1} \\ \mathbf{H}_{r-1}^{-1} & -\mathbf{H}_{r-1}^{-1} \end{pmatrix}, r > 0. \end{aligned}$$

Lemma 3.3. Let $\mathbb{F}$ be a field with $\mathrm{char}(\mathbb{F}) \neq 2$. If $\mathbf{M} \in \mathbb{F}^{n \times n}$ is dyadic, then $\mathbf{H}_r^{-1}\mathbf{M}\mathbf{H}_r$ is diagonal.

Proof. The lemma clearly holds for $r = 0$. Now let $r > 0$, and write

$$\mathbf{M} = \begin{pmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{B} & \mathbf{A} \end{pmatrix}$$

where $\mathbf{A}$ and $\mathbf{B}$ are dyadic. It follows that

$$\begin{aligned} \mathbf{H}_r^{-1}\mathbf{M}\mathbf{H}_r &= \frac{1}{2}\begin{pmatrix} \mathbf{H}_{r-1}^{-1} & \mathbf{H}_{r-1}^{-1} \\ \mathbf{H}_{r-1}^{-1} & -\mathbf{H}_{r-1}^{-1} \end{pmatrix}\begin{pmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{B} & \mathbf{A} \end{pmatrix}\begin{pmatrix} \mathbf{H}_{r-1} & \mathbf{H}_{r-1} \\ \mathbf{H}_{r-1} & -\mathbf{H}_{r-1} \end{pmatrix} \\ &= \begin{pmatrix} \mathbf{H}_{r-1}^{-1}\mathbf{M}_+\mathbf{H}_{r-1} & \mathbf{O} \\ \mathbf{O} & \mathbf{H}_{r-1}^{-1}\mathbf{M}_-\mathbf{H}_{r-1} \end{pmatrix}, \end{aligned}$$

and since both $\mathbf{M}_+ = \mathbf{A}+\mathbf{B}$ and $\mathbf{M}_- = \mathbf{A}-\mathbf{B}$ are dyadic, $\mathbf{H}_{r-1}^{-1}\mathbf{M}_+\mathbf{H}_{r-1}$ and $\mathbf{H}_{r-1}^{-1}\mathbf{M}_-\mathbf{H}_{r-1}$ are diagonal by induction, as is thus also $\mathbf{H}_r^{-1}\mathbf{M}\mathbf{H}_r$. □

Lemma 3.3 establishes that Sylvester-Hadamard matrices diagonalize all dyadic matrices. In this way, the factors in a product of dyadic matrices are thus simultaneously diagonalized, suggesting an efficient way to carry out the matrix multiplication, namely, computing $\mathbf{H}_r^{-1}(\mathbf{M}\mathbf{N})\mathbf{H}_r = (\mathbf{H}_r^{-1}\mathbf{M}\mathbf{H}_r)(\mathbf{H}_r^{-1}\mathbf{N}\mathbf{H}_r)$ given the diagonal forms $\mathbf{H}_r^{-1}\mathbf{M}\mathbf{H}_r$ and $\mathbf{H}_r^{-1}\mathbf{N}\mathbf{H}_r$ of two dyadic matrices $\mathbf{M}$ and $\mathbf{N}$ requires only n multiplications of the diagonal elements.

In fact, it is not necessary to compute $\mathbf{H}_r^{-1}\mathbf{M}\mathbf{H}_r$ in full to obtain the diagonal form of $\mathbf{M}$, as indicated by the following result:

Lemma 3.4. Let $\mathbb{F}$ be a field with $\mathrm{char}(\mathbb{F}) \neq 2$. The diagonal form of a dyadic matrix $\mathbf{M} \in \mathbb{F}^{n \times n}$ is the first row of $\mathbf{M}\mathbf{H}_r$. In other words, $\mathbf{H}_r^{-1}\boldsymbol{\Delta}(\mathbf{m})\mathbf{H}_r = \mathrm{diag}(\mathbf{m}\mathbf{H}_r)$.

Proof. The lemma clearly holds for $r = 0$. Now let $r > 0$, and with the notation of Lemma 3.3, the diagonal of $\mathbf{H}_r^{-1}\mathbf{M}\mathbf{H}_r$ is the concatenation of the diagonals of $\mathbf{H}_{r-1}^{-1}\mathbf{M}_+\mathbf{H}_{r-1}$ and $\mathbf{H}_{r-1}^{-1}\mathbf{M}_-\mathbf{H}_{r-1}$. Similarly, since

$$\mathbf{M}\mathbf{H}_r = \begin{pmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{B} & \mathbf{A} \end{pmatrix}\begin{pmatrix} \mathbf{H}_{r-1} & \mathbf{H}_{r-1} \\ \mathbf{H}_{r-1} & -\mathbf{H}_{r-1} \end{pmatrix} = \begin{pmatrix} \mathbf{M}_+\mathbf{H}_{r-1} & \mathbf{M}_-\mathbf{H}_{r-1} \\ \mathbf{M}_+\mathbf{H}_{r-1} & -\mathbf{M}_-\mathbf{H}_{r-1} \end{pmatrix},$$

the first row of $\mathbf{MH}_r$ is the concatenation of the first rows of $\mathbf{M}_+\mathbf{H}_{r-1}$ and $\mathbf{M}_-\mathbf{H}_{r-1}$, which by induction are the diagonals of $\mathbf{H}_{r-1}^{-1}\mathbf{M}_+\mathbf{H}_{r-1}$ and $\mathbf{H}_{r-1}^{-1}\mathbf{M}_-\mathbf{H}_{r-1}$ respectively, yielding the claimed property. □

Corollary 3.5. Computing $\mathbf{c}$ such that $\mathbf{\Delta}(\mathbf{a})\mathbf{\Delta}(\mathbf{b}) = \mathbf{\Delta}(\mathbf{c})$ involves only three multiplications of vectors by Sylvester-Hadamard matrices.

Proof. By Lemma 3.4,

$$\begin{aligned}\operatorname{diag}(\mathbf{aH}_r)\ \operatorname{diag}(\mathbf{bH}_r) &= (\mathbf{H}_r^{-1}\mathbf{\Delta}(\mathbf{a})\mathbf{H}_r)(\mathbf{H}_r^{-1}\mathbf{\Delta}(\mathbf{b})\mathbf{H}_r)\\ &= \mathbf{H}_r^{-1}\mathbf{\Delta}(\mathbf{a})\mathbf{\Delta}(\mathbf{b})\mathbf{H}_r = \mathbf{H}_r^{-1}\mathbf{\Delta}(\mathbf{c})\mathbf{H}_r\\ &= \operatorname{diag}(\mathbf{cH}_r).\end{aligned}$$

Now simply retrieve $\mathbf{c}$ from $\mathbf{z} = \mathbf{cH}_r$ as $\mathbf{c} = \mathbf{zH}_r^{-1} = 2^{-r}\mathbf{zH}_r$. □

The structure of Sylvester-Hadamard matrices leads to an efficient algorithm to compute $\mathbf{aH}_r$ for $\mathbf{a} \in \mathbb{F}^n$, which is known as the fast Walsh-Hadamard transform. Let $[\mathbf{a}_0, \mathbf{a}_1]$ be the two halves of $\mathbf{a}$. Thus

$$\mathbf{aH}_r = (\mathbf{a}_0, \mathbf{a}_1)\begin{pmatrix}\mathbf{H}_{r-1} & \mathbf{H}_{r-1}\\ \mathbf{H}_{r-1} & -\mathbf{H}_{r-1}\end{pmatrix} = ((\mathbf{a}_0 + \mathbf{a}_1)\mathbf{H}_{r-1}, (\mathbf{a}_0 - \mathbf{a}_1)\mathbf{H}_{r-1}).$$

This recursive algorithm, which can be easily written in a purely sequential fashion (Algorithm 10), has complexity $\Theta(n \log n)$, specifically, rn additions or subtractions in $\mathbb{F}$. It is therefore somewhat more efficient than the fast Fourier transform, which involves multiplications by n-th roots of unity, when they are available at all (otherwise working in extension fields is unavoidable, and more expensive).

Algorithm 10: The fast Walsh-Hadamard transform (FWHT)

```
Data: $r \in \mathbb{N}$, $n = 2^r$ and $\mathbf{a} \in \mathbb{F}^n$ with $\mathrm{char}(\mathbb{F}) \neq 2$.
Result: $\mathbf{aH}_r$, where $\mathbf{H}_r$ is a Sylvester-Hadamard matrix.
$v \leftarrow 1$;
for $j \leftarrow 1$ to $n$ do
    $w \leftarrow v$;
    $v \leftarrow 2v$;
    for $i \leftarrow 0$ to $n-1$ by $v$ do
        for $l \leftarrow 0$ to $w-1$ do
            $s \leftarrow a_{i+l}$;
            $q \leftarrow a_{i+l+w}$;
            $a_{i+l} \leftarrow s + q$;
            $a_{i+l+w} \leftarrow s - q$ ;
        end
    end
end
return $\mathbf{a}$;
```

The product of two dyadic matrices $\mathbf{\Delta}(\mathbf{a})$ and $\mathbf{\Delta}(\mathbf{b})$, or equivalently the dyadic convolution $\mathbf{a} \vartriangle \mathbf{b}$, can thus be efficiently computed as described in Algorithm 11. The total cost is $3rn$ additions or subtractions and $2n$ multiplications (half of these by the constant $2^{-r} = 1/n$) in $\mathbb{F}$, with an overall complexity $\Theta(n \log n)$.

Algorithm 11: Dyadic convolution via the FWHT

```
Data: $r \in \mathbb{N}$, $n = 2^r$ and $\mathbf{a}, \mathbf{b} \in \mathbb{F}^n$ with $\mathrm{char}(\mathbb{F}) \neq 2$.
Result: $\mathbf{a} \vartriangle \mathbf{b} \in \mathbb{F}^n$ such that $\boldsymbol{\Delta}(\mathbf{a})\boldsymbol{\Delta}(\mathbf{b}) = \boldsymbol{\Delta}(\mathbf{a} \vartriangle \mathbf{b})$.
$\mathbf{c} \leftarrow$ vector of length $n$, initialized with null elements;
$\tilde{\mathbf{c}} \leftarrow$ vector of length $n$, initialized with null elements;
Compute $\tilde{\mathbf{a}} \leftarrow \mathbf{a}\mathbf{H}_r$ via Algorithm 10 ;
Compute $\tilde{\mathbf{b}} \leftarrow \mathbf{b}\mathbf{H}_r$ via Algorithm 10 ;
for $j \leftarrow 0$ to $n - 1$ do
    $\tilde{c}_j \leftarrow \tilde{a}_j \tilde{b}_j$ ;
end
Compute $\mathbf{c} \leftarrow \tilde{\mathbf{c}}\mathbf{H}_r$ via Algorithm 10;
$\mathbf{c} \leftarrow 2^{-r}\mathbf{c}$;
return $\mathbf{c}$;
```

The fast Walsh-Hadamard transform itself is not immediately possible on fields of characteristic 2, since it depends on Sylvester-Hadamard matrices which must contain a primitive square root of unity. Yet the FWHT algorithm can be lifted to characteristic 0, namely, from $\mathbb{F}_2 = \mathbb{Z}/2\mathbb{Z}$ to $\mathbb{Z}$, or more generally from $\mathbb{F}_{2^N} = (\mathbb{Z}/2\mathbb{Z})[x]/P(x)$ (for some irreducible $P(x)$ of degree N) to $\mathbb{Z}[x]$. Algorithm 11 can then be applied, and its output mapped back to the relevant binary field by modular reduction. This incurs a space expansion by a logarithmic factor, though. Each bit from $\mathbb{F}_2$ is mapped to an intermediate value that can occupy as much as $3r + 1$ bits; correspondingly, each element from $\mathbb{F}_{2^N}$ is mapped to N intermediate values that can occupy as much as $(3r + 1)N$ bits. Thus the component-wise multiplication in Algorithm 11 becomes more complicated to implement for large N. However, the method remains very efficient for the binary case as long as each expanded integer component fits a computer word. For a typical word size of 32 bits and each binary component being expanded by a factor of $3r + 1$, this means that blocks as large as 1024 × 1024 can be tackled efficiently. On more restricted platforms where the maximum available word size is 16 bits, dyadic blocks of size 32 × 32 can still be handled with relative ease.

3.3 — Karatsuba Multiplication

In this section, we propose a method that is inspired by Karatsuba's algorithm for the multiplication of two integers [KO62]. Let us denote by $\mathbf{a}_0$ and $\mathbf{a}_1$, respectively, the first and second halves of $\mathbf{a}$, i.e.:

$$\begin{aligned} \mathbf{a}_0 &= \left(a_0, a_1, \ldots, a_{\frac{n}{2}-1}\right) \\ \mathbf{a}_1 &= \left(a_{\frac{n}{2}}, a_{\frac{n}{2}+1}, \ldots, a_{n-1}\right). \end{aligned} \tag{3.2}$$

The same notation is used for $\mathbf{b}_0$ and $\mathbf{b}_1$ and $\mathbf{c}_0$ and $\mathbf{c}_1$, corresponding to the halves of $\mathbf{b}$ and $\mathbf{c}$. Some straightforward computations in $\mathrm{char}(\mathbb{F}) = 2$ show that the following relations hold for $\boldsymbol{\Delta}(\mathbf{c}) = \boldsymbol{\Delta}(\mathbf{a})\boldsymbol{\Delta}(\mathbf{b})$:

$$\begin{aligned} \mathbf{c}_0 &= \mathbf{a}_0 \vartriangle \mathbf{b}_0 + \mathbf{a}_1 \vartriangle \mathbf{b}_1 \\ \mathbf{c}_1 &= (\mathbf{a}_0 + \mathbf{a}_1) \vartriangle (\mathbf{b}_0 + \mathbf{b}_1) + \mathbf{c}_0 \end{aligned} \tag{3.3}$$

which compute the result with 3 dyadic convolutions rather than the obvious 4, as observed by Karatsuba [KO62] in the context of polynomial and long integer multiplication.

The iterative application of Equation 3.3 allows to compute multiplications between dyadic matrices of any size. Let us denote by $C^{(2^z)}_{\mathtt{mul}}$ and $C^{(2^z)}_{\mathtt{sum}}$ the complexities of a multiplication and a sum between two signatures of length 2^z. For the sum of two dyadic signatures of size 2^z we have:

$$C^{(2^z)}_{\mathtt{mul}} = 2^z \cdot C_{\mathtt{sum}}, \tag{3.4}$$

where $C_{\mathtt{sum}}$ again denotes the complexity of a sum in the finite field. The complexity of this algorithm can thus be estimated as:

$$\begin{aligned} C_{\mathtt{Kar}} &= 3 \cdot C^{(2^{r-1})}_{\mathtt{mul}} + 4 \cdot C^{(2^{r-1})}_{\mathtt{sum}} \\ &= 3 \cdot C^{(2^{r-1})}_{\mathtt{mul}} + 4 \cdot 2^{r-1} \cdot C_{\mathtt{sum}} \\ &= 3 \cdot \left(3 \cdot C^{(2^{r-2})}_{\mathtt{mul}} + 4 \cdot C^{(2^{r-2})}_{\mathtt{sum}}\right) + 4 \cdot 2^{r-1} \cdot C_{\mathtt{sum}} \\ &= 3 \left(3 \cdot C^{(2^{r-2})}_{\mathtt{mul}} + 4 \cdot 2^{r-2} \cdot C_{\mathtt{sum}}\right) + 4 \cdot 2^{r-1} \cdot C_{\mathtt{sum}} \\ &= 3^2 \cdot C^{(2^{r-2})}_{\mathtt{mul}} + 4 \cdot \left(3 \cdot 2^{r-2} + 2^{r-1}\right) C_{\mathtt{sum}} \\ &= 3^3 \cdot C^{(2^{r-3})}_{\mathtt{mul}} + 4 \cdot \left(3^2 \cdot 2^{r-3} + 3 \cdot 2^{r-2} + 2^{r-1}\right) C_{\mathtt{sum}} \\ &= \cdots \\ &= 3^r \cdot C_{\mathtt{mul}} + 4 \cdot \left(\sum_{j=1}^{r} 3^{j-1} 2^{r-j}\right) \cdot C_{\mathtt{sum}} \\ &= 3^r \cdot C_{\mathtt{mul}} + \frac{4}{3} \cdot 2^r \cdot \left(\sum_{j=1}^{r} \left(\frac{3}{2}\right)^j\right) \cdot C_{\mathtt{sum}} \end{aligned} \tag{3.5}$$

Taking into account the well known sum of a geometric series, we have:

$$\begin{aligned} \sum_{j=1}^{r} \left(\frac{3}{2}\right)^j &= -1 + \sum_{j=0}^{r} \left(\frac{3}{2}\right)^j \\ &= -1 + \frac{1 - (\frac{3}{2})^{r+1}}{1 - \frac{3}{2}} = \frac{3^{r+1}}{2^r} - 3. \end{aligned} \tag{3.6}$$

Considering this result, equation (3.5) leads to:

$$C_{\mathtt{Kar}} = 3^r \cdot C_{\mathtt{mul}} + 4 \cdot (3^r - 2^r) \cdot C_{\mathtt{sum}} \tag{3.7}$$

3.4 — Comparisons

Table 3.1 shows the number of cycles for an implementation of the methods described in the previous subsections, i.e., standard multiplication, Karatsuba multiplication and dyadic convolution. We remark that the dyadic signatures are vectors of size 2^r and we used two different fields $\mathbb{F}_{2^6}$ and $\mathbb{F}_{2^8}$ with x^6+x^5+1 and $x^8+x^4+x^3+x^2+1$ as irreducible polynomials. However, the implementation code is generic and it is easily modifiable for other parameters such as different fields or r.

The implementation was developed in the C language. In all cases, we use no optimizations apart from the optimization from the GCC compiler ("-O3"). The GCC version used was 8.3.0, the code was compiled for the processor Intel(R) Core(TM) i5-5300UCPU @ 2.30GHz with 16GB of memory and operating system Arch linux version 2018.05.01 with kernel 4.19.45. We ran 100 times each piece of code and computed the average value of all measurements; to obtain the number of cycles, we used the file "cpucycles.h" from SUPERCOP[2].

Table 3.1: Comparison between multiplication methods. The numbers state the cycles used for one dyadic multiplication.

		Standard	Karatsuba	Dyadic Convolution
$\mathbb{F}_{2^6}$	$r = 5$	$18,214$	$5,850$	$11,262$
	$r = 6$	$68,761$	$14,109$	$19,857$
$\mathbb{F}_{2^8}$	$r = 5$	$24,583$	$6,992$	$11,666$
	$r = 6$	$98,706$	$18,769$	$20,772$

3.5 — Efficient Inversion of Dyadic and Quasi-Dyadic Matrices

In this section we propose an efficient algorithm for computing the inverse of quasi-dyadic matrices. The algorithm is targeted to matrices that are not fully dyadic (even though, obviously, they have to be square matrices). While it is of course possible to apply our procedure to fully dyadic matrices, these can in general be inverted much more easily, as we will see next.

To begin, remember that by definition a quasi-dyadic matrix (Definition 2.6) is an element of $\boldsymbol{\Delta}\left(\mathcal{R}^n\right)^{d_1\times d_2}$.

3.5.1 – Dyadic Matrices. The inverse of a dyadic matrix (i.e. $d_1 = d_2 = 1$) can be efficiently computed, using only the signature, as described by the following Lemma.

Theorem 3.6. Let $n = 2^r$ for $r \in \mathbb{N}$ and let $\boldsymbol{\Delta}(\mathbf{a}) \in \mathcal{R}^{n\times n}$, with $\mathsf{char}(\mathcal{R}) \neq 2$, be a dyadic matrix with signature $\mathbf{a}$. Then the inverse $\boldsymbol{\Delta}(\mathbf{a})^{-1}$ is the dyadic matrix $\boldsymbol{\Delta}(\mathbf{b})$, for $\mathbf{b} = \frac{1}{2^r}\tilde{\mathbf{b}}\mathbf{H}_r$, where $\tilde{\mathbf{b}}$ is the vector such that $\mathrm{diag}(\tilde{\mathbf{b}}) = [\mathrm{diag}\left(\mathbf{a}\mathbf{H}_r\right)]^{-1}$.

Proof. By definition we have $\boldsymbol{\Delta}(\mathbf{b})\boldsymbol{\Delta}(\mathbf{a}) = \mathbf{I}_n = \boldsymbol{\Delta}([1,0,\ldots,0])$. The diagonal form of $\mathbf{I}_n$ corresponds to the first row of the product $\mathbf{I}_n\mathbf{H}_r$, and so it is equal to the first row of $\mathbf{H}_r$, that is the length-n vector made of all ones. According to Corollary 3.5, we can write:

$$\mathrm{diag}(\mathbf{b}\mathbf{H}_r)\,\mathrm{diag}(\mathbf{a}\mathbf{H}_r) = \mathrm{diag}([1,1,\ldots,1]).$$

We then define $\mathbf{a}\mathbf{H}_r = [\lambda_0, \lambda_1, \ldots, \lambda_{n-1}]$, and obtain:

$$\mathrm{diag}(\mathbf{b}\mathbf{H}_r) = \mathrm{diag}([1,1,\ldots,1])(\mathrm{diag}(\mathbf{a}\mathbf{H}_r))^{-1} = \mathrm{diag}([\lambda_0^{-1}, \lambda_1^{-1}, \ldots, \lambda_{n-1}^{-1}]).$$

[2] https://bench.cr.yp.to/supercop.html

Because of Lemma 3.4, we finally have:

$$\mathbf{b} = (\mathrm{diag}(\mathbf{a}\mathbf{H}_r))^{-1}\mathbf{H}_r^{-1} = \tfrac{1}{2^r}(\mathrm{diag}(\mathbf{a}\mathbf{H}_r))^{-1}\mathbf{H}_r.$$

□

As we mentioned before, the above Lemma yields a very simple way for computing the inverse of a dyadic matrix: given a signature $\mathbf{a}$, we just need to compute its diagonalized form as $\mathbf{a}\mathbf{H}_r$, compute the reciprocals of its elements and put it in a vector $\tilde{\mathbf{b}}$. Finally, the inverse of $\boldsymbol{\Delta}(\mathbf{a})$ can be obtained as $\frac{1}{2^r}\tilde{\mathbf{b}}\mathbf{H}_r$. This property also leads to a very simple way to check the singularity of $\boldsymbol{\Delta}(\mathbf{a})$: if its diagonalized form contains some null elements, then it is singular.

We now focus on the case of dyadic matrices over a field $\mathbb{F}$ with characteristic 2. One can show by induction that in such a case a dyadic matrix $\boldsymbol{\Delta}(\mathbf{a})$ of dimension n satisfies $\boldsymbol{\Delta}(\mathbf{a})^2 = (\sum_i a_i)^2\mathbf{I}$, and hence its inverse, when it exists, is $\boldsymbol{\Delta}(\mathbf{a})^{-1} = (\sum_i a_i)^{-2}\boldsymbol{\Delta}(\mathbf{a})$, which can be computed in $O(n)$ steps since it is entirely determined by its first row. It is equally clear that $\det\boldsymbol{\Delta}(\mathbf{a}) = (\sum_i a_i)^n$, which can be computed with the same complexity (notice that raising to the power of $n = 2^r$ only involves r squarings). Basically, verifying whether a dyadic matrix has full rank or not can be easily done by checking whether the sum of the elements of the signature equals 0.

3.5.2 – Quasi-Dyadic Matrices. Consider a quasi-dyadic matrix $\mathbf{M}$. Since the matrix has to be square, we have $d_1 = d_2 = d$, and the matrix has dimension $dn \times dn$. Such a matrix can be compactly represented just by the signatures of the dyadic blocks. To simplify notation, we can denote the signature of the dyadic-block in position (i, j) as $\hat{\mathbf{m}}_{i,j}$, and store all such vectors in a matrix $\hat{\mathbf{M}} \in \mathcal{R}^{d\times dn}$:

$$\hat{\mathbf{M}} = \begin{pmatrix} \hat{\mathbf{m}}_{0,0} & \hat{\mathbf{m}}_{0,1} & \cdots & \hat{\mathbf{m}}_{0,d-1} \\ \hat{\mathbf{m}}_{1,0} & \hat{\mathbf{m}}_{1,1} & \cdots & \hat{\mathbf{m}}_{1,d-1} \\ \vdots & \vdots & \ddots & \vdots \\ \hat{\mathbf{m}}_{d-1,0} & \hat{\mathbf{m}}_{d-1,1} & \cdots & \hat{\mathbf{m}}_{d-1,d-1} \end{pmatrix}. \tag{3.8}$$

We focus again on the special case of quasi-dyadic matrices over a field $\mathbb{F}$ with characteristic 2.

The LUP decomposition is a method which factorizes a matrix $\mathbf{M}$ as $\mathbf{LUP}$, where $\mathbf{L}$ and $\mathbf{U}$ are lower triangular and upper triangular matrices, respectively, and $\mathbf{P}$ is a permutation. Exploiting this factorization, the inverse of $\mathbf{M}$ can thus be expressed as:

$$\mathbf{M}^{-1} = \mathbf{P}^{-1}\mathbf{U}^{-1}\mathbf{L}^{-1}. \tag{3.9}$$

The advantage of this method is that the inverses appearing in (3.9) can be easily computed, because of their particular structures. In fact, the inverse of an upper (lower) triangular matrix is obtained via a simple backward (forward) substitution procedure, while the inverse of $\mathbf{P}$ is its transpose.

In some cases, applying a block-wise LUP decomposition might lead to some complexity reduction; for instance, see [PS17] for the inversion of a sparse matrix. Here, we consider the case of a quasi-dyadic matrix; the corresponding procedure is shown in Algorithm 12.

Our proposed procedure consists in using a block decomposition, which works directly on the signatures, in order to exploit the simple and efficient algebra of dyadic matrices.

The operations in Algorithm 12 only refer to the signatures in $\hat{\mathbf{M}}$: for instance, the expression $\hat{\mathbf{m}}_{i,j}\hat{\mathbf{m}}_{i,l}$ means the product between the dyadic matrices having as signatures $\hat{\mathbf{m}}_{i,j}$ and $\hat{\mathbf{m}}_{i,l}$, i.e., $\Delta(\hat{m}_{i,j} \triangle \hat{m}_{i,l})$. This choice may result in some abuse of notation, but is useful to emphasize the fact that, as we have explained in the previous sections, operations with dyadics can be efficiently computed just by taking into account their signatures. It can be easily shown that, for a quasi-dyadic matrix, its factors **L**, **U** and **P** are in quasi-dyadic form as well: as we have done for the matrix **M**, we refer to their compact representations as $\hat{\mathbf{L}}$, $\hat{\mathbf{U}}$ and $\hat{\mathbf{P}}$, respectively.

```
Algorithm 12: LUP Decomposition of a Quasi-Dyadic Matrix
Data: $d, r \in \mathbb{N}$, $n = 2^r$ and $\hat{\mathbf{M}} \in \mathbb{F}^{d \times dn}$ with $\mathrm{char}(\mathbb{F}) = 2$.
Result: $\hat{\mathbf{M}} \in \mathbb{F}^{d \times dn}$, $\hat{\mathbf{P}} \in \mathbb{N}^d$.
$\hat{\mathbf{P}} \leftarrow [0, 1, \ldots, d-1]$;
$u \leftarrow 0$;
for $j \leftarrow 0$ to $d-1$ do
    Update $u$, $\hat{\mathbf{M}}$ and $\hat{\mathbf{P}}$ via Algorithm 13   /* Pivoting of the signatures
     in the j-th column */;
    if $u = 0$ then
        return $u$                    /* $\hat{\mathbf{M}}$ is singular */;
    end
    for $i \leftarrow j+1$ to $d$ do
        $\hat{\mathbf{m}}_{i,j} \leftarrow \hat{\mathbf{m}}_{i,j}\hat{\mathbf{m}}_{j,j}^{-1}$;
    end
    for $i \leftarrow j+1$ to $d-1$ do
        for $l \leftarrow j+1$ to $d-1$ do
            $\hat{\mathbf{m}}_{i,l} \leftarrow \hat{\mathbf{m}}_{i,l} + \hat{\mathbf{m}}_{i,j}\hat{\mathbf{m}}_{j,l}$;
        end
    end
end
return $\hat{\mathbf{M}}$, $\hat{\mathbf{P}}$;
```

Algorithm 12 takes as input a matrix $\hat{\mathbf{M}}$, as in (3.8), and computes its LUP factorization; outputs of the algorithm are the modified matrix $\hat{\mathbf{M}}$, having as elements the ones of its factors $\hat{\mathbf{L}}$ and $\hat{\mathbf{U}}$, and the permutation $\hat{\mathbf{P}}$. As in (3.8), we denote as $\hat{\mathbf{m}}_{i,j}$ the signature in position (i, j) in the output matrix $\hat{\mathbf{M}}$. The matrices $\hat{\mathbf{L}}$ and $\hat{\mathbf{U}}$ can then be expressed as:

$$\hat{\mathbf{L}} = \begin{pmatrix} \hat{1} & \hat{0} & \hat{0} & \cdots & \hat{0} \\ \hat{\mathbf{m}}_{1,0} & \hat{1} & \hat{0} & \cdots & \hat{0} \\ \hat{\mathbf{m}}_{2,0} & \hat{\mathbf{m}}_{2,1} & \hat{1} & \cdots & \hat{0} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \hat{\mathbf{m}}_{d-1,0} & \hat{\mathbf{m}}_{d-1,1} & \hat{\mathbf{m}}_{d-1,2} & \cdots & \hat{1} \end{pmatrix}, \tag{3.10}$$

$$\hat{\mathbf{U}} = \begin{pmatrix} \hat{\mathbf{m}}_{0,0} & \hat{\mathbf{m}}_{0,1} & \hat{\mathbf{m}}_{0,2} & \cdots & \hat{\mathbf{m}}_{0,d-1} \\ \hat{0} & \hat{\mathbf{m}}_{1,1} & \hat{\mathbf{m}}_{1,2} & \cdots & \hat{\mathbf{m}}_{1,d-1} \\ \hat{0} & \hat{0} & \hat{\mathbf{m}}_{2,2} & \cdots & \hat{\mathbf{m}}_{2,d-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \hat{0} & \hat{0} & \hat{0} & \cdots & \hat{\mathbf{m}}_{d-1,d-1} \end{pmatrix} \tag{3.11}$$

where $\hat{1}$ and $\hat{0}$ denote, respectively, the signature of the identity matrix and the one of the null matrix (i.e. the length-k vectors $[1, 0, \ldots, 0]$ and $[0, 0, \ldots, 0]$).

The matrix $\hat{\mathbf{P}}$ is represented by a length-d vector $[p_0, p_1, \ldots, p_{d-1}]$, containing a permutation of the integers $[0, 1, \ldots, d-1]$; the rows of $\hat{\mathbf{M}}$ get permuted according to the elements of $\hat{\mathbf{P}}$. In particular, the elements of $\hat{\mathbf{P}}$ are obtained through a block pivoting procedure, which is described in Algorithm 13.

Algorithm 13: Block pivoting

```
Data: $d, j, r \in \mathbb{N}$, $n = 2^r$, $\hat{\mathbf{P}} \in \mathbb{N}^d$ and $\hat{\mathbf{M}} \in \mathbb{F}^{d \times dn}$ with $\mathrm{char}(\mathbb{F}) = 2$.
Result: $u \in \mathbb{N}$ as 1 if it needs continue pivoting or 0 otherwise.
$u \leftarrow 0$;
$i \leftarrow j$;
while $i \leqslant d-1$ do
    $w \leftarrow \mathtt{sum}(\hat{\mathbf{m}}_{i,j})$          /* Sum of the elements in $\hat{\mathbf{m}}_{i,j}$ */;
    if $w = 0$ then
        $z \leftarrow p_j$;
        $p_j \leftarrow p_i$;
        $p_i \leftarrow z$;
        for $l \leftarrow 0$ to $d-1$ do
            $z \leftarrow \hat{\mathbf{m}}_{j,l}$;
            $\hat{\mathbf{m}}_{j,l} \leftarrow \hat{\mathbf{m}}_{i,l}$;
            $\hat{\mathbf{m}}_{i,l} \leftarrow z$;
            $i \leftarrow i+1$;
        end
    else
        $i \leftarrow d$;
        $u \leftarrow 1$;
    end
end
return $u$;
```

The function of block pivoting in Algorithm 13 takes as input $\hat{\mathbf{M}}$, $\hat{\mathbf{P}}$ and an integer j, and searches for a pivot (i.e., a non singular signature) in the j-th column of $\hat{\mathbf{M}}$, starting from $\hat{\mathbf{m}}_{j,j}$, and places it in position (j, j). As the procedure goes on, every time a singular signature is tested, the rows of $\hat{\mathbf{M}}$ get permuted; the elements of $\hat{\mathbf{P}}$ are accordingly modified. We give up on inverting $\hat{\mathbf{M}}$ if the j-th column contains only singular blocks.

We point out that, for the matrices we are considering, we expect Algorithm 12 to be particularly efficient. First of all, as we have already said, this is due to the possibility of efficiently performing operations involving dyadic matrices. In addition, the dyadic structure should also speed-up the pivoting procedure. For our applications, we can consider a signature in $\hat{\mathbf{M}}$ as a collection of k random elements picked from $\mathbb{F}_{2^m}$: thus, their sum can be assumed to be a random variable with uniform distribution among the elements of the field $\mathbb{F}_{2^m}$. So, the probability of it being equal to 0, which corresponds to the probability of the corresponding signature to be singular, equals 2^{-m}. This probability gets lower as m increases: this fact means that the expected number of operations performed by Algorithm 13 should be particularly low. Basically, most of the times the function will just compute the sum of the elements in $\hat{\mathbf{m}}_{j,j}$ and verify that it is nonzero.

Once the factorization of $\hat{\mathbf{M}}$ has been obtained, we just need to perform the computation of $\mathbf{M}^{-1}$ through (3.9). Since the inverse of a triangular matrix maintains the original triangular structure, the computation of the inverses $\hat{\mathbf{L}}^{-1}$ and $\hat{\mathbf{U}}^{-1}$ can be efficiently performed. A possible way for computing these matrices is to store the elements of both matrices in just one output matrix $\hat{\mathbf{T}}$. Those operations are made in Algorithm 14.

In Algorithm 14, the matrix $\hat{\mathbf{I}}_d$ is the compact representation of a $dn \times dn$ identity matrix, and so is composed of signatures $\delta_{i,j}\hat{1}$, where $\delta_{i,j}$ denotes the Kronecker delta.

Algorithm 14: Computation of the inverses $\hat{\mathbf{L}}^{-1}$ and $\hat{\mathbf{U}}^{-1}$ as just one output matrix $\hat{\mathbf{T}}$.

```
Data: $d, r \in \mathbb{N}$, $n = 2^r$ and $\hat{\mathbf{M}} \in \mathbb{F}^{d \times dn}$ with $\mathrm{char}(\mathbb{F}) = 2$.
Result: $\hat{\mathbf{T}} \in \mathbb{F}^{d \times dn}$ as $\hat{\mathbf{L}}^{-1}$ and $\hat{\mathbf{U}}^{-1}$.
$\hat{\mathbf{T}} \leftarrow \hat{\mathbf{I}}_d$;
for $j \leftarrow 0$ to $d-1$ do
    for $i \leftarrow j+1$ to $d-1$ do
        for $l \leftarrow j$ to $i-1$ do
            $\hat{\mathbf{t}}_{i,j} \leftarrow \hat{\mathbf{t}}_{i,j} + \hat{\mathbf{m}}_{i,k}\hat{\mathbf{t}}_{k,j}$;
        end
    end
    for $i \leftarrow j$ to $d-1$ do
        for $l \leftarrow j$ to $i-1$ do
            $\hat{\mathbf{t}}_{j,i} \leftarrow \hat{\mathbf{t}}_{j,i} + \hat{\mathbf{m}}_{k,i}\hat{\mathbf{t}}_{j,k}$;
        end
        $\hat{\mathbf{t}}_{j,i} \leftarrow \hat{\mathbf{t}}_{j,i}\hat{\mathbf{m}}_{i,i}^{-1}$;
    end
end
return $\hat{\mathbf{T}}$
```

If we denote as $\hat{\mathbf{t}}_{i,j}$ the signature in position (i, j), we have:

$$\hat{\mathbf{L}}^{-1} = \begin{pmatrix} \hat{1} & \hat{0} & \hat{0} & \cdots & \hat{0} \\ \hat{\mathbf{t}}_{1,0} & \hat{1} & \hat{0} & \cdots & \hat{0} \\ \hat{\mathbf{t}}_{2,0} & \hat{\mathbf{t}}_{2,1} & \hat{1} & \cdots & \hat{0} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \hat{\mathbf{t}}_{d-1,0} & \hat{\mathbf{t}}_{d-1,1} & \hat{\mathbf{t}}_{d-1,2} & \cdots & \hat{1} \end{pmatrix}, \tag{3.12}$$

$$\hat{\mathbf{U}}^{-1} = \begin{pmatrix} \hat{\mathbf{t}}_{0,0} & \hat{\mathbf{t}}_{0,1} & \hat{\mathbf{t}}_{0,2} & \cdots & \hat{\mathbf{t}}_{0,d-1} \\ \hat{0} & \hat{\mathbf{t}}_{1,1} & \hat{\mathbf{t}}_{1,2} & \cdots & \hat{\mathbf{t}}_{1,d-1} \\ \hat{0} & \hat{0} & \hat{\mathbf{t}}_{2,2} & \cdots & \hat{\mathbf{t}}_{2,d-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \hat{0} & \hat{0} & \hat{0} & \cdots & \hat{\mathbf{t}}_{d-1,d-1} \end{pmatrix}. \tag{3.13}$$

These matrix operations will be used in a cryptosystem in Chapter 4.

Chapter 4

DAGS: Key Encapsulation from Dyadic Generalized Srivastava Codes

As it was mentioned in Chapter 1, code-based cryptography is one of the main mathematical problems that are safe against quantum computers. The area is largely based on the Decoding Problem [BMvT78]. Over the years, since McEliece's seminal work [McE78], many cryptosystems have been proposed, trying to balance security and efficiency. In particular, those cryptosystems need to deal with inherent problems such as the large size of the public keys. In fact, while McEliece's original cryptosystem, which is based on binary Goppa codes, still unbroken in principle, it has a key of several hundred kilobytes, which has effectively prevented its use in many applications. Note that the original parameters $n = 1024$ and $k = 512$ were broken in 2008. [1]

There are currently two main trends to deal with this issue, and they both involve structured matrices: the first is based on "traditional" algebraic codes such as Goppa or Srivastava codes; the second suggests to use sparse matrices as in LDPC/MDPC codes. This work builds on the former approach, initiated in 2009 by Berger, Cayrel, Gaborit and Otmani [BCGO09], who proposed Quasi-Cyclic (QC) codes, and Misoczki and Barreto [MB09], suggesting Quasi-Dyadic (QD) codes instead (later generalized to Quasi-Monoidic (QM) codes [BLM11]). Both proposals feature very compact public keys due to the introduction of the extra algebraic structure, but unfortunately this also led to a vulnerability. Indeed, Faugère, Otmani, Perret and Tillich [FOPT10] devised a clever attack (known simply as FOPT) which exploits the algebraic structure to build a system of equations, which can successively be solved using Gröbner basis techniques. As a result, the QC proposal is definitely compromised, while the QD/QM approach needs to be treated with caution. In fact, for a proper choice of parameters, it is still possible to design secure schemes using for instance binary Goppa codes, or Generalized Srivastava (GS) codes as suggested by Persichetti in [Per12a].

In this chapter, we present DAGS, a Key Encapsulation Mechanism (KEM) that follows the QD approach using GS codes. The KEM achieves IND-CCA security following the

[1] This chapter is based on joint work with Paulo S.L.M. Barreto, Brice Odilon Boidje, Pierre-Louis Cayrel, Gilbert Ndollane Dione, Kris Gaj, Cheikh Thiécoumba Gueye, Richard Haeussler, Jean Belo Klamti, Ousmane N'diaye, Duc Tri Nguyen, Edoardo Persichetti, and Jefferson E. Ricardini. The results were published in Journal of Mathematical Cryptology [BBB+18] and at Code-Based Cryptography Workshop 2019 [BBB+19]. The acronym DAGS holds for Dyadic GS Codes.

recent framework by Hofheinz, Hövelmanns and Kiltz [HHK17], and features compact public keys and efficient encapsulation and decapsulation algorithms. One task is to balance parameters to achieve the most efficient scheme, while at the same time avoiding the FOPT attack. DAGS tries to find a balance between security and small parameters.

First, we will present the protocol giving specification about the design. Second, we will show the security of the scheme and the behavior against the known attacks in the literature at the time of the NIST submission. The proofs and details about the theoretical security of DAGS are left out here since my coauthors worked on them. They can be found in [BBB+18]. Third, we will present details about the implementation showing choices that were made in the project and the implications of those choices and the limitations and advantages of using DAGS.

After submission to the NIST competition, DAGS, the system this chapter is about, got attacked and tweaked several times. This chapter follows the three phases of DAGS chronologically, to give proper credit to the attacks, and includes the attacks and countermeasures that were derived in the later phases.

4.1 — Protocol Specification

4.1.1 – Design Rationale. In this section we introduce the three algorithms that form DAGS. System parameters are the code length n and dimension k, the values s and t which define a GS code as seen in Subsection 2.2.4, the cardinality of the base field q and the degree of the field extension m. In addition, we have $k = k' + k''$, where k' is arbitrary and is set to be “small". In practice, the value of k' depends on the base field and is such that a random vector of length k' over $\mathbb{F}_q$ provides at least 256 bits of entropy.

Let us select randomly non-zero distinct h_0 and h_j for $j = 2^l, l = 0, \dots, \lceil \log_2 n \rceil - 1$. The key generation process uses the following fundamental equation

$$\frac{1}{h_{i \oplus j}} = \frac{1}{h_i} + \frac{1}{h_j} + \frac{1}{h_0} \tag{4.1}$$

to build the vector $\mathbf{h} = (h_0, \dots, h_{n-1})$ of elements of $\mathbb{F}_{q^m}$. This is then used to form $\mathbf{u}$ and $\mathbf{v}$, where $\mathbf{u} = (u_0, \dots, u_{s-1})$ and $\mathbf{v} = (v_0, \dots, v_{n-1})$. The construction of u_i and v_i are the following: $u_i = \frac{1}{h_i} + \omega$ and $v_j = \frac{1}{h_j} + \frac{1}{h_0} + \omega$ for a random $\omega \in \mathbb{F}_{q^m}$. The vectors $\mathbf{u}$ and $\mathbf{v}$ then are used to build a matrix $C(\mathbf{u}, \mathbf{v})$ with components $C_{ij} = \frac{1}{u_i + v_j}$ which we call Cauchy matrix. Each element in the Cauchy matrix is then successively powered forming several blocks which are superimposed and then multiplied by a random diagonal matrix. Finally, the resulting matrix is projected onto the base field and row-reduced to systematic form. The overall process is described below.

4.1.1.1 – Key Generation.

a) Generate $\mathbf{h}$. To do this:
   a) Choose random non-zero distinct h_0 and h_j for $j = 2^l, l = 0, \dots, \lceil \log_2 n \rceil - 1$.
   b) Form the remaining n elements using Equation 4.1.
   c) Return s blocks up to length n excluding any block containing zero as entry.
b) Build the Cauchy support. To do this:
   a) Choose a random[2] offset $\omega \xleftarrow{\$} \mathbb{F}_{q^m}$.

[2] See Subsection 4.1.2 for restrictions about the choice of the offset.

b) Compute $u_i = \dfrac{1}{h_i} + \omega$ for $i = 0, \dots, s-1$.

c) Compute $v_j = \dfrac{1}{h_j} + \dfrac{1}{h_0} + \omega$ for $j = 0, \dots, n-1$.

d) Set $\mathbf{u} = (u_0, \dots, u_{s-1})$ and $\mathbf{v} = (v_0, \dots, v_{n-1})$.

c) Form the Cauchy matrix $\hat{H}_1 = C(\mathbf{u}, \mathbf{v}) = \begin{pmatrix} \frac{1}{(u_0+v_0)} & \cdots & \frac{1}{(u_0+v_{n-1})} \\ \frac{1}{(u_1+v_0)} & \cdots & \frac{1}{(u_1+v_{n-1})} \\ \vdots & \vdots & \vdots \\ \frac{1}{(u_{s-1}+v_1)} & \cdots & \frac{1}{(u_{s-1}+v_{n-1})} \end{pmatrix}$.

d) Build $\hat{H}_i$, $i = 2, \dots t$, by raising each element of $\hat{H}_1$ to the power of i.

e) Superimpose blocks $\hat{H}_i$, for $i = 2, \dots t$ to form matrix $\hat{H}$ as:

$$\hat{H} = \begin{pmatrix} \hat{H}_1 \\ \hat{H}_2 \\ \vdots \\ \hat{H}_t \end{pmatrix}.$$

f) Generate vector $\mathbf{z}$ by picking $\lceil \frac{n}{s} \rceil$ random elements $z_{is} \in \mathbb{F}_{q^m}, i = 0, \dots, \lceil \frac{n}{s} \rceil - 1$ and put $z_{is+j} = z_{is}$ for $i = 0, \dots, \lceil \frac{n}{s} \rceil - 1$, $j = 0, \dots, s-1$.

g) Set $y_j = \left(\dfrac{z_j}{\displaystyle\prod_{i=0}^{s-1} (u_i - v_j)^t} \right)$ for $j = 0, \dots, n-1$ and $\mathbf{y} = (y_0, \dots, y_{n-1})$.

h) Form $H = \hat{H} \cdot \text{Diag}(\mathbf{z})$.

i) Project H onto $\mathbb{F}_q$ using the co-trace function: call this H_{base}.

j) Write H_{base} in systematic form $(M \mid I_{n-k})$.

k) The public key is the generator matrix $G = (I_k \mid M^T)$.

l) The private key is the pair $(\mathbf{v}, \mathbf{y})$.

The encapsulation and decapsulation algorithms make use of two hash functions $\mathcal{G} : \mathbb{F}_q^{k'} \to \mathbb{F}_q^k$ and $\mathcal{H} : \mathbb{F}_q^{k'} \to \mathbb{F}_q^{k'}$, the former with the task of generating randomness for the scheme, the latter to provide “plaintext confirmation" as in [HHK17]. The shared symmetric key is obtained via another hash function $\mathcal{K} : \mathbb{F}_q^{k'} \to \{0, 1\}^\ell$, where ℓ is the desired key length.

4.1.1.2 – Encapsulation.

a) Choose $\mathbf{m} \xleftarrow{\$} \mathbb{F}_q^{k'}$.

b) Compute $\mathbf{r} = \mathcal{G}(\mathbf{m})$ and $\mathbf{d} = \mathcal{H}(\mathbf{m})$.

c) Parse $\mathbf{r}$ as $(\boldsymbol{\rho} \parallel \boldsymbol{\sigma})$ then set $\boldsymbol{\mu} = (\boldsymbol{\rho} \parallel \mathbf{m})$.

d) Generate error vector $\mathbf{e}$ of length n and weight $w = \frac{st}{2}$ from $\boldsymbol{\sigma}$.

e) Compute $\mathbf{c} = \boldsymbol{\mu} G + \mathbf{e}$.

f) Compute $\mathbf{s} = \mathcal{K}(\mathbf{m})$.

g) Output ciphertext $(\mathbf{c}, \mathbf{d})$; the encapsulated key is $\mathbf{s}$.

The decapsulation algorithm consists mainly of decoding the noisy codeword received as part of the ciphertext. This is done using the alternant decoding algorithm described in Algorithm 2 and requires the parity-check matrix to be in alternant form.

4.1.1.3 – Decapsulation.

a) Get parity-check matrix H' in alternant form from private key[3].
b) Use H' to decode $\mathbf{c}$ and obtain codeword $\mathbf{\mu}'G$ and error $\mathbf{e}'$.
c) Output $\perp$ if decoding fails or $\mathrm{wt}(\mathbf{e}') \neq w$.
d) Recover $\mathbf{\mu}'$ and parse it as $(\mathbf{\rho}' \parallel \mathbf{m}')$.
e) Compute $\mathbf{r}' = \mathcal{G}(\mathbf{m}')$ and $\mathbf{d}' = \mathcal{H}(\mathbf{m}')$.
f) Parse $\mathbf{r}'$ as $(\mathbf{\rho}'' \parallel \mathbf{\sigma}')$.
g) Generate error vector $\mathbf{e}''$ of length n and weight w from $\mathbf{\sigma}'$.
h) If $\mathbf{e}' \neq \mathbf{e}'' \vee \mathbf{\rho}' \neq \mathbf{\rho}'' \vee \mathbf{d} \neq \mathbf{d}'$ output $\perp$.
i) Else compute $\mathbf{s} = \mathcal{K}(\mathbf{m}')$.
j) The decapsulated key is $\mathbf{s}$.

Persichetti [Per12b, Chapter 2] shows that the code just defined is a Generalized Srivastava code. DAGS is built upon the McEliece encryption framework using GS Codes as shown in [Per12b, Chapter 4], with a notable exception. In fact, we incorporate the “randomized" version of McEliece by [NIKM08] into our scheme. This is extremely beneficial for two distinct aspects: first of all, it allows us to use a much shorter vector $\mathbf{m}$ to generate the remaining components of the scheme, which greatly improves efficiency. Secondly, it allows us to get tighter security bounds. In fact, a shorter input makes all the hash functions easy to compute, and minimizes the overhead due to the IND-CCA2 security in the QROM. Note that our protocol differs slightly from the paradigm presented in [HHK17], in the fact that we do not perform a full re-encryption in the decapsulation algorithm. Instead, we simply re-generate the randomness and compare it with the one obtained after decoding. This is possible since, unlike a generic PKE, McEliece decryption reveals the randomness used, in our case $\mathbf{e}$ (and $\mathbf{\rho}$). It is clear that if the re-generated randomness is equal to the retrieved one, the resulting encryption will also be equal. This allows us to further decrease computation time.

4.1.2 – Choice of ω. As discussed in the text above, in our scheme we use a standard alternant decoder. After computing the syndrome of the word to be decoded, the next step is to recover the Error Locator Polynomial (ELP), by means of the Euclidean algorithm for polynomial division; the algorithm then proceeds by finding the roots of the ELP. There is a 1-1 correspondence between these roots and the error positions: in fact, there is an error in position i if and only if the inverse of x_i is a root of the ELP.

Now, the generation of the error vector is random, hence we can assume the probability of having an error in position i to be around $st/2n$; since the codes can correct the maximum number of errors when mst is close to $n/2$, we can estimate this probability as $1/4m$, which is reasonably low for any nontrivial choice of m; however, we still argue that the code is not fully decodable and we now explain how to adapt the key generation algorithm to ensure that all the x_i’s are nonzero.

As part of the key generation algorithm we assign to each x_i the value v_i, hence it is enough to restrict the possible choices for ω to the set $\{\alpha \in \mathbb{F}_{q^m} | \alpha \neq 1/h_i + 1/h_0, i = 0, \ldots, n-1\}$. In doing so, we considerably restrict the possible choices for ω but we ensure that the decoding algorithm works properly.

[3] See Algorithm 4.3.3.1.

4.2 — Known Attacks and Parameters

We start by briefly presenting the hard problem on which DAGS is based, and then discuss the main attacks on the scheme and related security concerns.

4.2.1 – Hard Problems from Coding Theory. Most of the code-based cryptographic constructions are based on the hardness of the following problem, known as the (q-ary) *Syndrome Decoding Problem (SDP)*.

Problem 1. Given an $(n-k) \times n$ full-rank matrix H over $\mathbb{F}_q$, a vector $\mathbf{s} \in \mathbb{F}_q^{n-k}$, and a non-negative integer w, find a vector $\boldsymbol{e} \in \mathbb{F}_q^n$ of weight w such that $H\boldsymbol{e}^T = \mathbf{s}$ if such $\boldsymbol{e}$ exist.

The corresponding decision problem was proved to be NP-complete in 1978 [BMvT78], but only for binary codes. In 1994, it was proved that this result holds for codes over all finite fields ([Bar94], in Russian, and [Bar97, Theorem 4.1]).

In addition, many schemes (including the original McEliece proposal) require the following computational assumption.

Assumption 4.1. The public matrix output by the key generation algorithm is computationally indistinguishable from a uniformly chosen matrix of the same size.

The assumption above is historically believed to be true, except for very particular cases. For instance, there exists a distinguisher (Faugère, Gauthier, Otmani, Perret and Tillich [FGO+13]) for cryptographic protocols that make use of high-rate Goppa codes (like the CFS signature scheme [CFS01]). Moreover, it is worth mentioning that the “classical" methods for obtaining an indistinguishable public matrix, such as the use of scrambling matrices S and P, are rather outdated and unpractical and can introduce vulnerabilities to the scheme as per the work in [Str10, STM+08].

4.2.2 – Decoding Attacks. The main approach for solving the SDP is a technique known as Information Set Decoding (ISD), first introduced by Prange [Pra62]. Among several variants and generalizations, Peters showed [Pet10] that it is possible to apply Prange's approach to generic q-ary codes. Other approaches such as Statistical Decoding [Al-01, Nie11] are usually considered less efficient. Thus, when choosing parameters, we will focus mainly on defeating attacks on the ISD family.

Peters [Pet11] provides an inital study about non-asymptotic complexity for the ISD in the binary case, which later on Hamdaoui and Sendrier in [HS13] provides a similar study. For codes over $\mathbb{F}_q$, instead, a bound is given in [NPC+17], which extends the work of Peters. For a practical evaluation of the ISD running times and corresponding security levels, we used Peters' ISDFQ script [Pet] which analyzes the random choices in the ISD algorithm and computes the number of iterations.

Quantum Speedup. Bernstein in [Ber10] shows that Grover's algorithm applies to ISD-like algorithms, effectively halving the asymptotic exponent in the complexity estimates. Later, it was proven in [KT17] that several variants of ISD have the potential to achieve a better exponent, however the improvement was disappointingly away from the factor of 2 that could be expected. In fact, the number of required iterations for running a quantum version of Stern's algorithm reduce the factor of square root provide by Grover's algorithm as [Ber10] when one tried to combine an advanced quantum information-set decoding. To be on the safe side, we consider that the best attack quantum attack to take at least the square root of the time computed with the ISDFQ script.

4.2.3 – Algebraic Attacks. While, as we discussed above, recovering a private matrix from a public one can in general be a very difficult problem, the presence of extra structure in the code properties can have a considerable effect in lowering this difficulty.

A very effective structural attack was introduced by Faugère, Otmani, Perret and Tillich in [FOPT10]. The attack (for convenience simply called FOPT) relies on the property $H \cdot G^T = 0$ to build an algebraic system, using then Gröbner basis techniques to solve it and recover the private key. Note that this property is valid for every linear code, but it is the special properties of structured alternant codes that make the system solvable, as they contribute to considerably reduce the number of variables.

The attack was originally aimed at two variants of McEliece, introduced respectively in [BCGO09] and [MB09]. The first variant, using quasi-cyclic codes, was easily broken in all proposed parameters. The second variant, instead, only considered quasi-dyadic Goppa codes. In this case, most of the parameters proposed have also been broken, except for the binary case (i.e. base field $\mathbb{F}_2$). This was, in truth, not connected to the base field per se, but rather depended on the fact that, with a smaller base field, the authors provided a much higher extension degree m, as they were keeping constant the value $q^m = 2^{16}$. As it turns out, the extension degree m plays a key role in evaluating the complexity of the attack.

4.2.3.1 – Attack Complexity for DAGS. Following up on their own work, the authors in [FOPT10] analyze the attack in detail with the aim of evaluating its complexity at least somewhat rigorously. At the core of the attack, there is an affine bilinear system, which is derived from the initial system of equations by applying various algebraic relations due to the quasi-dyadic structure. This bilinear system has $n_{X'} + n_{Y'}$ variables, where these are, respectively, the number of X and Y “free" variables (after applying the relations) of an alternant parity-check matrix H with $H_{ij} = Y_j X_j^i$. Moreover, the *degree of regularity* (i.e. the maximal degree of the polynomials appearing during the computation) is bounded from above by $1 + \min(n_{X'}, n_{Y'})$. It is shown that this number dominates computation time, and so it is crucial to correctly evaluate it in our case. In fact, for the original proposal based on Goppa codes [MB09], we have $n_{X'} = n_0 - 2 + \log_2(\ell)$, where ℓ is the dyadic order and $n_0 = n/\ell$ is the number of dyadic blocks, and $n_{Y'} = m - 1$. We report an excerpt of some numbers from the paper in Table 4.1 below.

It is possible to observe several facts. For instance, in every set of parameters, $n_{X'} \gg n_{Y'}$ and so $n_{Y'} = m - 1$ is the most important number here. In other words, the degree of the extension field is crucial in evaluating the complexity of the attack, as we mentioned above. As a confirmation, it is easy to notice that all parameters were broken very easily when this is extremely small (1 in most cases), while the running time scales accordingly when m grows. In fact, the attack could not be performed in practice on the first set of parameters (hence the N/A).

The first three groups of parameters are taken from, respectively, Table 2, Table 3 and Table 5 of the preliminary (unpublished) version of [MB09], while the last group consists of some *ad hoc* parameters generated by the FOPT authors. The absence of parameters from Table 4 of [MB09] stands out: in fact, all of these parameters used $\mathbb{F}_2$ as base field

Table 4.1: Details of FOPT applied to Quasi-Dyadic Goppa codes [FOPT10].

q	m	n	k	n_0	ℓ	$n_{X'}$	$n_{Y'}$	Time/Operations
2	16	3584	1536	56	2^6	60	15	N/A
2^2	8	3584	1536	56	2^6	60	7	1,776.3 sec / $2^{34.2}$ op
2^4	4	2048	1024	32	2^6	36	3	0.5 sec / $2^{22.1}$ op
2^8	2	1280	768	20	2^6	24	1	0.03 sec / $2^{16.7}$ op
2^8	2	640	512	10	2^6	14	1	0.03 sec / $2^{15.9}$ op
2^8	2	768	512	6	2^7	11	1	0.02 sec / $2^{15.4}$ op
2^8	2	1024	512	4	2^8	10	1	0.11 sec / $2^{19.2}$ op
2^8	2	512	256	4	2^7	9	1	0.06 sec / $2^{17.7}$ op
2^8	2	640	384	5	2^7	10	1	0.02 sec / $2^{14.5}$ op
2^8	2	768	512	6	2^7	11	1	0.01 sec / $2^{16.6}$ op
2^8	2	1280	768	5	2^8	11	1	0.05 sec / $2^{17.5}$ op
2^8	2	1536	1024	6	2^8	12	1	0.06 sec / $2^{17.8}$ op
2^4	4	4096	3584	32	2^7	37	3	7.1 sec / $2^{26.1}$ op
2^8	2	3072	2048	6	2^9	13	1	0.15 sec / $2^{19.7}$ op

and thus could not be broken (at least not without very long computations), just like for the case of the first set. As a result, an updated version of [MB09] was produced for publication, in which the insecure parameters are removed and only the binary sets (those of Table 4) appear.

Towards the end of [FOPT10], the authors present a bound on the theoretical complexity of computing a Gröbner basis of the affine bilinear system which is at the core of the attack. They then evaluate this bound and compare it with the number of operations required in practice (last column of Table 4.1). The bound is given by

$$T_{\text{theo}} \approx \left(\sum_{\substack{d_1+d_2=D \\ 1\leqslant d_1,d_2\leqslant D-1}} \Big(\dim R_{d_1,d_2} - [t_1^{d_1} t_2^{d_2}]HS(t_1,t_2)\Big) \dim R_{d_1,d_2} \right) \tag{4.2}$$

where D is the degree of regularity of the system, $\dim R_{d_1,d_2} = \binom{d_1+n_{X'}}{d_1}\binom{d_2+n_{X'}}{d_2}$ and $[t_1^{d_1} t_2^{d_2}]HS(t_1,t_2)$ indicates the coefficient of the term $[t_1^{d_1} t_2^{d_2}]$ in the Hilbert bi-series $HS(t_1,t_2)$, as defined in Appendix A of [FOPT10].

As it turns out this bound is quite loose, being sometimes above and sometimes below the experimental results, depending on which set of parameters is considered. As such, it is to be read as an approximate indication of the expected complexity of a parameter set, and it only allows to have a rough idea of the security provided for each set. Nevertheless, since we are able to compute the bound for all DAGS proposed parameters, we will keep this number in mind when proposing parameters (Section 4.2.5), to make sure our choices are at least not obviously insecure.

As a bottomline, it is clear that the complexity of the attack scales somewhat proportionally to the value $m-1$ which defines the dimension of the solution space. The FOPT authors point out that any scheme for which this dimension is less or equal to 20 should be within the scope of the attack.

Since GS codes are also alternant codes, the attack can be applied to our proposal as well. There is, however, one very important difference to keep in mind. In fact, it is shown in [Per12a] that, thanks to the particular structure of GS codes, the dimension of the solution space is defined by $mt - 1$, rather than $m - 1$. This provides greater flexibility when selecting parameters for the code, and it allows, in particular, to "rest the weight" of the attack on two shoulders rather than just one. Thus we are able to balance the parameters and keep the extension degree m small while still achieving a large dimension for the solution space. We will discuss parameter selection in detail in Section 4.2.5 as already mentioned.

4.2.3.2 – Folded Codes. In 2016, an extension of the Faugère, Otmani, Perret and Tillich (FOPT) attack appeared in [FOP+16]. The authors introduce a new technique called "folding", and show that it is possible to reduce the complexity of the FOPT attack to the complexity of attacking a smaller code (the "folded" code), thanks to the strong properties of the automorphism group of the alternant codes in use. The attack turns out to be very efficient against Goppa codes, as it is possible to recover a folded code which is also a Goppa code. As a consequence, it is possible to tweak the attack to solve a different, augmented system of equations (named $\mathsf{G}_{X,Y'}$), rather than the "basic" one which is aimed at generic alternant codes (called $\mathsf{A}_{X,Y'}$). Moreover, this can be further refined in the case of Quasi-Dyadic Goppa codes, leading to a third system of equations referred to as $\mathsf{McE}_{X,Y'}$. In parallel, the authors present a new method called "structural elimination" that manages to eliminate a considerable number of variables, at the price of an increased degree in the equations considered. Solving the "eliminated" systems (called respectively $\mathsf{elimA}_{X',Y'}$, $\mathsf{elimG}_{X',Y'}$ and $\mathsf{elimMcE}_{X',Y'}$) often proves a more efficient choice, but the authors do occasionally use the non-eliminated systems when it is more convenient to do so.

The paper concentrates on attacking several parameters that were proposed for signature schemes and encryption schemes in various follow-ups of [BCGO09] and [MB09]. The latter includes, among others, some of the parameters presented in Table 4.1. While codes designed to work for signature schemes turn out to be very easy to attack (due to their particular nature), the situation for encryption is more complex. The authors are able to obtain a speedup in the attack times for previously investigated parameters, but some of the parameters could still not be solved in practice. We report the results below, where we indicate the type of system chosen to be solved, and we keep some of the previously-shown parameters for ease of comparison.

The authors of [FOPT10] do not report timings for codes that were already broken with FOPT in negligible time (like all of those where $m = 2$). Also, we have excluded from our table parameters that are not relevant to DAGS, such as the quasi-monoidic codes of [BLM11] (where q is not a power of 2).

This table confirms our intuition that high values of m result in a high number of operations, and that complexity increases somewhat proportionally to this value. Note that the last 5 sets of parameters were not broken in practice and the estimated complexity is always quite high: it is not clear what the authors mean by $\leqslant$, but it is reasonable to assume that the actual complexity would not be dramatically smaller than the indicated value, and thus at least 2^{80} in all cases. Consequently, the claim that parameters with $m - 1 \leqslant 20$ are "within the scope of the attack" looks now perhaps a bit optimistic.

The fourth set of parameters seem to contradict our intuition, since it was broken in

Table 4.2: Details of folding attack applied to Quasi-Dyadic Goppa codes [FOP+16].

q	m	n	k	n_0	ℓ	System	Folding	FOPT
2^4	4	2048	1024	32	2^6	$\mathsf{elimA}_{X',Y'}$	0.01 sec	0.5 sec
2^4	4	4096	3584	32	2^7	$\mathsf{elimA}_{X',Y'}$	0.01 sec	7.1 sec
2^2	8	3584	1536	56	2^6	$\mathsf{elimA}_{X',Y'}$	0.04 sec	1776.3 sec
2	16	4864	4352	152	2^5	$\mathsf{elimMcE}_{X',Y'}$	18 sec	N/A
2	12	3200	1664	25	2^7	$\mathsf{elimMcE}_{X',Y'}$	$\leqslant 2^{83.5}$ op	N/A
2	14	5376	3584	42	2^7	$\mathsf{elimMcE}_{X',Y'}$	$\leqslant 2^{96.1}$ op	N/A
2	15	11264	3584	22	2^9	$\mathsf{elimMcE}_{X',Y'}$	$\leqslant 2^{146}$ op	N/A
2	16	6912	2816	27	2^8	$\mathsf{elimMcE}_{X',Y'}$	$\leqslant 2^{168}$ op	N/A
2	16	8192	4096	32	2^8	$\mathsf{elimMcE}_{X',Y'}$	$\leqslant 2^{157}$ op	N/A

practice with relative ease even though $m = 16$. However, it is possible to see that this is a code with a ridiculously high rate (k/n is very close to 1) and in particular, the very large number of blocks n_0 clearly stands out. We remark that this set of parameters was chosen by the attack authors to demonstrate the efficiency of the attack and in practice such a poor choice of parameters would never be considered. Nevertheless, it gives us the confirmation (if needed) that high-rate codes are a bad choice not only for ISD-like attacks, but for structural attacks also.

The authors of [FOPT10] did not present any explicit result against GS codes and, in particular, it is not known whether a folded GS code is still a GS code. Thus, the attack in this case is limited to solving the generic system $\mathsf{A}_{X,Y'}$ (or $\mathsf{elimA}_{X',Y'}$) and does not benefit from the speedups which are specific to (binary) Goppa codes. For these reasons, and until an accurate complexity analysis is available, we choose to attain to the latest measurable guidelines and choose our parameters such that the dimension of the solution space for the algebraic system is strictly greater than 20. We then compute the bound given by Equation (4.2) and report it as an additional indication of the expected complexity of the attack.

4.2.4 – The Barelli-Couvreur Attack. The attack makes use of a novel construction called *Norm-Trace Codes* [BC18]. As the name suggests, these codes are the result of the application of both the Trace and the Norm operation to a certain support vector, and they are alternant codes. In particular, they are subfield subcodes of Reed-Solomon codes. The construction of these codes is given explicitly only for the specific case $m = 2$ (as is the case in all DAGS parameters), i.e. the support vector has components in $\mathbb{F}_{q^2}$, in which case the norm-trace code is defined as

$$\mathcal{NT}(\mathbf{x}) = \langle \mathbf{1}, \mathrm{Tr}(\mathbf{x}), \mathrm{Tr}(\alpha\mathbf{x}), \mathrm{N}(\mathbf{x}) \rangle,$$

where α is an element of trace 1.

The main idea is that there exists a specific norm-trace code that is the $\mathtt{conductor}$ of the secret subcode into the public code. By “conductor" the authors refer to the largest code for which the Schur product (i.e. the component-wise product of all codewords, denoted by $\star$) is entirely contained in the target, i.e.

$$\mathtt{Cond}(\mathcal{D}, \mathcal{C}) = \{\mathbf{u} \in \mathbb{F}_q^n : \mathbf{u} \star \mathcal{D} \subseteq \mathcal{C}\}.$$

The authors present two strategies to determine the secret subcode. The first strategy is essentially an exhaustive search over all the codes of the proper co-dimension. This co-dimension is given by $2q/s$, since s is the size of the permutation group of the code, which is non-trivial in our case due to the code being quasi-dyadic. While such a brute force in principle would be too expensive, the authors present a few refinements that make it feasible, which include an observation on the code rate of the codes in use, and the use of shortened codes. The second strategy, instead, consists of solving a bilinear system, which is obtained using the parity-check matrix of the public code and treating as unknowns the elements of a generator matrix for the secret code (as well as the support vector $\mathbf{x}$). This system is solved using Gröbner bases techniques, and benefits from a reduction in the number of variables similar to the one performed in FOPT, as well as the refinements mentioned above (shortened codes).

In both cases, it is easy to deduce that the two parameters q and s are crucial in determining the cost of running this step of the attack, which dominates the overall cost. In fact, the authors are able to provide an accurate complexity analysis for the first strategy which confirms this intuition. The average number of iterations of the brute force search is given by q^{2c}, where c is exactly the co-dimension described above, i.e. $c = 2q/s$. In addition, it is shown that the cost of computing Schur products is $2n^3$ operations in the base field. Thus, the overall cost of the recovery step is $2n^3q^{4q/s}$ operations in $\mathbb{F}_q$. The authors then argue that wrapping up the attack has negligible cost, and that q-ary operations can be done in constant time (using tables) when q is not too big. All this leads to a complexity which is below the desired security level for all of the DAGS parameters that had been proposed in the DAGS submission to NIST. We report these numbers in Table 4.3.

Table 4.3: Early DAGS parameters.

Security Level	q	m	n	k	s	t	w	Attack cost in [BC18]
1	2^5	2	832	416	2^4	13	104	2^{70}
3	2^6	2	1216	512	2^5	11	176	2^{80}
5	2^6	2	2112	704	2^6	11	352	2^{55}

As it is possible to observe, the attack complexity is especially low for the last set of parameters since the dyadic order s was chosen to be 2^6, and this is probably too large to provide security against this attack. Still, we point out that, at the time these parameters were proposed, there was no indication this was the case, since this attack is using an entirely new technique, and it is unrelated to the FOPT and folding attacks that we described in Section 4.2.3.2.

While the attack performs very well against the original DAGS parameter sets, it is overall not as critical as it appears. In fact, it is shown in Section 5.3 of [BBB+18] how this can be defeated even by modifying a single parameter, namely the size of the base field q. This is the case for DAGS_3, where changing this value from 2^6 to 2^8 is enough to push the attack complexity beyond the claimed security level. Updated parameters were introduced in [BBB+18], and we report them below.

Note that, for DAGS_5, the dyadic order s needed to be amended, too, and the rest of the code parameters modified accordingly to respect the requirements on code length, dimension etc. The case of DAGS_1 is a little peculiar. In fact, the theoretical complexity

Table 4.4: DAGS parameters secure against Barelli-Couvreur Attack [BC18].

Security Level	q	m	n	k	s	t	w	Attack cost in [BC18]
1	2^6	2	832	416	2^4	13	104	$\approx 2^{128}$
3	2^8	2	1216	512	2^5	11	176	$\approx 2^{288}$
5	2^8	2	1600	896	2^5	11	176	$\approx 2^{289}$

of the first attack approach can be made large enough by simply switching from $q = 2^5$ to $q = 2^6$, similarly to what was done for DAGS_3. With this in mind, and for the sake of simplicity, [BBB+18] featured this choice of parameters for DAGS_1, as reported in Table 4.4.

Unfortunately, the attack authors were not able to provide a security analysis for the second strategy (bilinear system). This is due to the fact that the attack uses Gröbner based techniques, and it is very hard to evaluate the cost in this case (similarly to what happened for FOPT). In this case, the only evidence the authors provide is experimental, and based on running the attack in practice on all the parameters. The authors report running times around 15 minutes for the first set and less than a minute for the last, while they admit they were not able to complete the execution in the middle case. This seems to match the evidence from the complexity results obtained for the first strategy, and suggests a speedup proportional to those. The attack fails to run in practice for the middle set and it gives the confidence to believe that updated parameters make the attack infeasible.

4.2.5 – Parameter Selection. To choose our parameters, we keep in mind all the remarks from the previous sections about decoding attacks and structural attacks. As we have just seen, we need to respect the condition $mt \geqslant 21$ to guarantee security against FOPT. At the same time, to prevent the BC attack q has to be chosen large enough and s cannot be too big. Finally, for ISD to be computationally intensive we require a sufficiently large number w of errors to decode: this is given by $st/2$ according to the minimum distance of GS codes.

In addition, we tune our parameters to optimize the performance of the implementation. In this regard, the best results are obtained when the extension degree m is as small as possible. This, however, requires the base field to be large enough to accommodate sufficiently big codes (against ISD attacks), since the maximum size for the code length n is capped by $q^m - s$. Realistically, this means we want q^m to be at least 2^{12}, and the optimal choice in this sense seems to be $q = 2^8$ (see Section 4.3). Finally, note that s is constrained to be a power of 2, and that odd values of t seem to offer best performance.

Putting all the pieces together, we are able to present three set of parameters, in Table 4.4. These correspond to three of the security levels indicated by NIST (first column), which are related to the hardness of performing a key search attack on three different variants of a block cipher, such as AES (with key-length respectively 128, 192 and 256). As far as quantum attacks are concerned, we claim that ISD with Grover (see above) will usually require more resources than a Grover search attack on AES for the circuit depths suggested by NIST (parameter MAXDEPTH). Thus, classical security bits are the bottleneck in our case, and as such we choose our parameters to provide 128, 192 and 256 bits of classical security for security levels 1, 3 and 5 respectively. Furthermore, we show in Table 4.6 the size of the public key, private key and ciphertext in bytes. Section 4.6 shows new parameters for DAGS that are the current suggested paramaters secure against the known attacks.

Table 4.5: Storage requirements in bytes for the parameters in Table 4.4.

Parameter Set	Public Key	Private Key	Ciphertext
DAGS_1	8112	2496	656
DAGS_3	11264	4864	1248
DAGS_5	19712	6400	1632

For practical reasons, during the rest of this thesis we will refer to these parameters respectively as DAGS_1, DAGS_3 and DAGS_5.

For the above parameters, it is easy to observe that the value $n_{Y'}$ is always greater or equal to 21 (it is in fact 25 for DAGS_1), which keeps us clear of FOPT. With respect to the BC attack, the complexity analysis provided by the authors results in a value of $\approx 2^{126}$ for DAGS_1 and more than 2^{288} for the other two sets. Finally, the running cost of ISD (using Peters' script) is estimated at 2^{128}, 2^{192} and 2^{256} respectively, as desired.

4.3 — Implementation and Performance Analysis

4.3.1 – Components. For DAGS_1, the finite field $\mathbb{F}_{2^6}$ is built using the polynomial x^6+x+1 and then extended to $\mathbb{F}_{2^{12}}$ using the quadratic irreducible polynomial $x^2+\alpha x+\alpha$, where α is a primitive element of $\mathbb{F}_{2^6}$. In particular, we choose $\alpha = \gamma^{65}$ where γ is a primitive element of $\mathbb{F}_{2^{12}}$. This particular choice allows for more efficient arithmetic using a *conversion matrix* to switch between $\mathbb{F}_{2^6}$ and $\mathbb{F}_{2^{12}}$ and vice-versa. Similarly, for DAGS_3 and DAGS_5, we build the base field using $x^8 + x^4 + x^3 + x^2 + 1$ and the extension field $\mathbb{F}_{2^{16}}$ is obtained via $x^2 + \beta^{50}x + \beta$, where β is a primitive element of $\mathbb{F}_{2^8}$. In Section 4.3.4, we will show the basic operations such as multiplication, inversion and tower field operations. The three main functions from DAGS are defined as:

Key generation: the key generation algorithm key_gen is composed of five main functions: *build_dyadic_signature*, *build_cauchy_matrix*, *build_trapdoor*, *project_H_on_F_q* and *generate_public_key*. The first three first functions are in charge of generating the signature, the Cauchy matrix and generating the private key respectively. The fourth function consists in the projection of the trapdoor matrix generated in *build_trapdoor* from $\mathbb{F}_{q^m}$ to $\mathbb{F}_q$. The last function, that is *generate_public_key*, computes Gauss elimination and removes the redundancy part of the matrix hence generating the public key.

Encapsulation: the encapsulation algorithm is essentially composed of the function *encapsulation* in the file *encapsulation.c*, where it computes the expansion of the message, generation of the error vector and the McEliece-like encryption. In the end, the function computes the hash function $\mathcal{K}$ to get the shared secret.

Decapsulation: the decapsulation algorithm consists mainly of the function *decapsulation* in the file *decapsulation.c*, where we essentially run the decoding algorithm, the computation of the error vector, few comparisons and computation of hash functions. In the end, we compute the hash function $\mathcal{K}$ to get the shared secret.

4.3.2 – Randomness Generation. The randomness used in our implementation is provided by the NIST API. It uses AES as a PRNG, where NIST chooses the seed in order to have a controlled environment for tests. We use this random generator to obtain our input message $\mathbf{m}$, after which we calculate $\mathcal{G}(\mathbf{m})$ and $\mathcal{H}(\mathbf{m})$, where $\mathcal{G}$ is an expansion function and $\mathcal{H}$ is a hash function. In practice, we compute both using the *SHAKE256* function [Div15] from the Keccak family. To generate a low-weight error vector, we take part of $\mathcal{G}(\mathbf{m})$ as a seed $\boldsymbol{\sigma}$. We use again *SHAKE256* to expand the seed into a string of length n, then transform the latter into a fixed-weight string using a deterministic function.

4.3.3 – Efficient Private Key Reconstruction and Decoding. As mentioned in Section 4.1.1, in our scheme we use a standard alternant decoder (Step 2 of Algorithm 4.1.1.3), which requires the input to be a matrix in alternant form, i.e. $H'_{ij} = y_j x_j^i$ for $i = 0, \dots, st-1$ and $j = 0, \dots, n-1$. The first step consists of computing the syndrome of the received word, $H'\mathbf{c}^T$. Now, defining the whole alternant matrix H' as private key would require storing stn elements of $\mathbb{F}_{q^m}$, leading to huge key sizes. It would be possible to store as private key just the defining vectors $\mathbf{u}, \mathbf{v}$ and $\mathbf{z}$, and then compute the alternant form during decapsulation. Doing so would drastically reduce the private key size, but would also significantly slow down the decapsulation algorithm. Thus we implemented the following hybrid approach. We use $\mathbf{u}, \mathbf{v}$ and $\mathbf{z}$ to compute the vector $\mathbf{y}$ during key generation and store $(\mathbf{v}, \mathbf{y})$ as private key, which still results in a compact size. Then, we complete the computation of the alternant form in the decapsulation algorithm. To avoid unnecessary overhead, we incorporate this computation together with the syndrome computation. The process is detailed as follows.

4.3.3.1 – Alternant-Syndrome Computation.

a) Input received word $\mathbf{c}$ to be decoded.
b) Compute the vector $\mathbf{s} = \mathrm{Diag}(\mathbf{y}) \cdot \mathbf{c}^T$.
c) Form intermediate matrix $\tilde{H}$. To do this:

    a) Set first row equal to $\mathbf{s}$.
    b) Obtain row i, for $i = 1, \dots, st-1$, by multiplying the j-th element of row $i-1$ by v_j, for $j = 0, \dots, n-1$.

d) Sum elements in each row and output resulting vector.

4.3.4 – Operations in $\mathbb{F}_{q^m}$.

4.3.4.1 – Arithmetic operations. In the first version of DAGS the base field multiplications were done using tables, i.e., log and anti-log tables as shown in [DHF+18]. However, those types of operations can be exploited by side-channel attacks such as cache, timing or fault attacks. It is possible to avoid that by computing the multiplication every time that it is required. Listing 4.1 shows how to compute the multiplication of two elements in $\mathbb{F}_{2^8}$ and perform the reduction by $f(x) = x^8 + x^4 + x^3 + x^2 + 1$, for a larger q, such as $q = 2^{16}$, it is possible to use the same technique as well. For other fields one needs to just take care of using all the bits necessary and changing the irreducible polynomial. However, for big fields such as $\mathbb{F}_{2^{163}}$ the multiplication and reduction can be quite costly, see [BCP18] for more details.

```
#include <stdint.h>
typedef uint16_t gf;
gf gf_q_m_mult(gf in0, gf in1) {
```

```
gf reduction = 0;
gf tmp = 0;
gf t0 = in0, t1 = in1;
int i;
//Multiplication
tmp = 0;
for (i = 0; i < 8; i++){
  tmp ^= (t0*(t1 & (1 << i)));
}
//first and second steps of reduction
for (i=0; i < 2; i++){
  reduction = (tmp >> 8) & 0x7F; //this grabs bits 8–14
  tmp = tmp & 0xFF; //this grabs bits 0–7
  tmp ^= reduction;
  tmp ^= reduction << 2;
  tmp ^= reduction << 3;
  tmp ^= reduction << 4;
}

return tmp;
}
```

Listing 4.1: Multiplication of two elements in $\mathbb{F}_2[x]/(x^8+x^4+x^3+x^2+1)$

As showed in Section 4.1.1 in the overview of the protocol, it is possible to notice that one basic and important operation for DAGS is *inversion* since the vectors $\mathbf{u}, \mathbf{v}, \mathbf{y}$ and $\mathbf{h}$ are inverses in $\mathbb{F}_{q^m}$. Fortunately, since we are in $\mathbb{F}_{q^m}$ we can easily compute the inverse of an element $a \in \mathbb{F}_{q^m}$ as $a^{-1} = a^{q^m-2}$. The inverse operation can be implemented as squarings and multiplications as shown in Listing 4.2 where we perform the inverse of elements in $\mathbb{F}_{2^8}$, a similar approach can be applied for $\mathbb{F}_{2^6}$, $\mathbb{F}_{2^{12}}$ and $\mathbb{F}_{2^{16}}$.

```
#include <stdint.h>
typedef uint16_t gf;
gf gf_inv(gf in) {
  gf tmp_11;
  gf tmp_111;
  gf tmp_1111;

  gf out = in;
  out = gf_sqr(out); //a^2
  tmp_111 = out;//a^2
  tmp_11 = gf_mult(out, in);//a^2a = a^3

  out = gf_sqr(tmp_11);//(a^3)^2 = a^6
  out = gf_sqr(out);// (a^6)^2 = a^12
  tmp_111 = gf_mult(out, tmp_11);//a^12a^3 = a^15

  out = gf_sqr(tmp_111);//(a^15)^2 = a^30
  out = gf_sqr(out);//(a^30)^2 = a^60

  tmp_1111 = gf_mult(out, tmp_111);//(a^60) a^3 = a^63

  out = gf_sqr(tmp_1111);//(a^63)^2 = a^126
  out = gf_mult(out, in);//(a^126) a = a^127
  out = gf_sqr(out);//(a^127)^2 = a^254
  return out;

}
```

Listing 4.2: Inversion of one element in $\mathbb{F}_{2^8}$

4.3.4.2 – Representation operations. Two important operations that occur on key generation and decapsulation on DAGS, are the projection of elements from $\mathbb{F}_{q^m}$ to elements in $\mathbb{F}_q$ and vice-versa. In the implementation, we called those operations as *relative field representation* and *absolute field representation* where *relative field representation* means $\mathbb{F}_{q^m}$ to elements in $\mathbb{F}_q$ and *absolute field representation* means $\mathbb{F}_q$ to elements in $\mathbb{F}_{q^m}$. Mathematically, those operations are cheap since they just change the representation of the basis by a linear mapping. However, we remind the reader that we need to perform this operations in constant-time and fast since this operations can leak sensitive information. For solving this, we present Listings 4.3 and 4.4 for the example of $\mathbb{F}_q = \mathbb{F}_{2^8}$ and $\mathbb{F}_{2^{16}} \equiv \mathbb{F}_{2^8}[x]/(x^2 + x + \alpha)$.

```
#include <stdint.h>
typedef uint16_t gf;
gf relative_field_representation(gf a, int k) {
  gf x[extension] = {0};
  uint8_t b_0_t = a & 0x1;
  uint8_t b_1_t = (a & 0x2) >> 1;
  uint8_t b_2_t = (a & 0x4) >> 2;
  uint8_t b_3_t = (a & 0x8) >> 3;
  uint8_t b_4_t = (a & 0x10) >> 4;
  uint8_t b_5_t = (a & 0x20) >> 5;
  uint8_t b_6_t = (a & 0x40) >> 6;
  uint8_t b_7_t = (a & 0x80) >> 7;
  uint8_t b_8_t = (a & 0x100) >> 8;
  uint8_t b_9_t = (a & 0x200) >> 9;
  uint8_t b_10_t = (a & 0x400) >> 10;
  uint8_t b_11_t = (a & 0x800) >> 11;
  uint8_t b_12_t = (a & 0x1000) >> 12;
  uint8_t b_13_t = (a & 0x2000) >> 13;
  uint8_t b_14_t = (a & 0x4000) >> 14;
  uint8_t b_15_t = (a & 0x8000) >> 15;


  uint8_t A = b_5_t ^ b_9_t ^ b_11_t ^ b_12_t ^ b_13_t ^ b_15_t;
  uint8_t B = b_2_t ^ b_3_t ^ b_4_t ^ b_5_t;
  uint8_t C = b_6_t ^ b_8_t;
  uint8_t D = b_7_t ^ b_9_t;
  uint8_t E = b_10_t ^ b_11_t;

  uint8_t b_4  = C ^ D ^ b_10_t; // 4
  uint8_t b_11 = b_5_t ^ b_6_t ^ E ^ b_15_t; // 4
  uint8_t b_9  = b_3_t ^ b_8_t ^ A; // 7
  uint8_t b_5  = b_4_t ^ b_4 ^ A; // 8
  uint8_t b_6  = b_4 ^ b_12_t ^ b_14_t; // 6
  uint8_t b_15 = b_4 ^ b_6_t ^ b_11_t ^ b_15_t; // 5
  uint8_t b_3  = b_3_t ^ b_6_t ^ b_8_t ^ b_9_t ^ b_11 ; // 6
  uint8_t b_13 = b_4 ^ E ^ b_5_t ^ b_13_t; // 6
  uint8_t b_10 = b_2_t ^ b_4_t ^ b_11_t ^ b_14_t ^ b_15_t; // 4
  uint8_t b_12 = B ^ b_4 ^ b_4_t^ b_8_t ^ b_12_t ^ b_10; //  9
  uint8_t b_1  = B ^ b_8_t ^ b_11_t ^ b_13_t ^ b_14_t; // 7
  uint8_t b_0  = E ^ b_0_t ^ b_8_t ^ b_9_t ^ b_12_t; // 5
  uint8_t b_2  = C ^ b_4_t ^ b_11_t ^ b_13_t ^ b_14_t; // 5
  uint8_t b_14 = C ^ b_4_t ^ b_7_t ^ b_12_t ^ b_13_t ^ b_14_t ^
    b_15_t; // 7
  uint8_t b_7  = D ^ b_5_t ^  b_8_t ^ b_13_t ^ b_14_t ^ b_15_t;
    // 6
  uint8_t b_8  = B ^ b_5_t ^ b_1_t ^ b_7_t ^ b_10_t ^ b_12_t ^
    b_13_t ^ b_15_t; // 8

  x[0] = (b_0) | (b_1 << 1) | (b_2 << 2) | (b_3 << 3) | (b_4 <<
    4) | (b_5 << 5) | (b_6 << 6) | (b_7 << 7);
  x[1] = (b_8) | (b_9 << 1) | (b_10 << 2) | (b_11 << 3) | (b_12
    << 4) | (b_13 << 5) | (b_14 << 6) | (b_15 << 7);
  return x[k];
}
```

Listing 4.3: Relative field representation from $\mathbb{F}_{2^{16}}$ to $\mathbb{F}_{2^8}$

```
#include <stdint.h>
typedef uint16_t gf;
gf absolut_field_representation(gf *element) {
  gf beta = 788;
  gf tmp1 = 0, tmp2 = 0, in0 = element[0], in1 = element[1];
  uint16_t ctl = 0;
  for (int i = 0; i < 8; i++) {
    gf power = gf_pow_f_q_m(beta, i);
    ctl = constat_time_is_not_equal_zero(in0 & (1 << i));
    constant_time_mux(ctl, tmp1 ^ power, tmp1);

  }
  for (int i = 0; i < 8; i++) {
    gf power = gf_pow_f_q_m(beta, i);
    ctl = constat_time_is_not_equal_zero(in1 & (1 << i));
```

```
        constant_time_mux(ctl, tmp2 ^ power, tmp2);
    }

    gf tmp_3 = gf_q_m_mult(1, tmp1);
    gf tmp_4 = gf_q_m_mult(2, tmp2);

    gf result = tmp_3 ^ tmp_4;
    return result;

}
```

Listing 4.4: absolute field representation from $\mathbb{F}_{2^8}$ to $\mathbb{F}_{2^{16}}$

4.3.5 – Time and Space Requirements. The implementation is in ANSI C, as requested by the NIST “call of proposals” for the reference implementation. For the measurements we used a processor x64 Intel core i5-5300U@2.30GHz with 16GiB of RAM compiled with GCC version 8.2.120181127 without any optimization and running on Arch Linux.

We start by considering space requirements. In Figure 4.1 we recall the flow between two parties **Maria** and **João** in a standard Key Exchange protocol derived from a KEM.

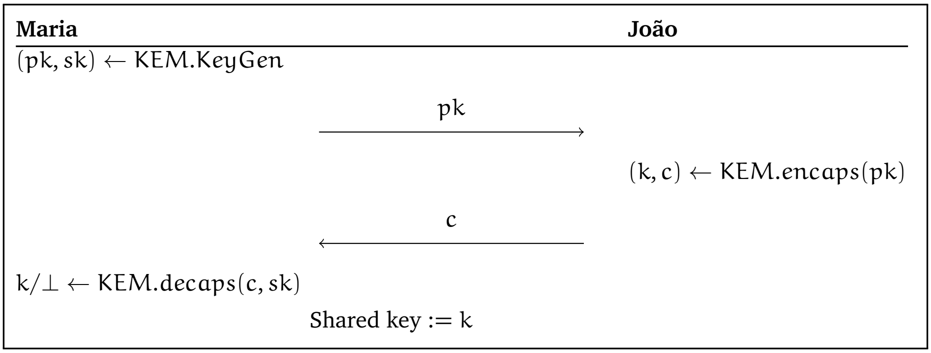

Maria
João
$(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KEM.KeyGen}$
pk
$(\mathsf{k}, \mathsf{c}) \leftarrow \mathsf{KEM.encaps}(\mathsf{pk})$
c
$\mathsf{k}/\perp \leftarrow \mathsf{KEM.decaps}(\mathsf{c}, \mathsf{sk})$
Shared key $:= \mathsf{k}$

Figure 4.1: KEM-based Key Encapsulation flow

When instantiated with DAGS, the public key is given by the generator matrix G. The non-identity block M^T is $k \times (n-k) = k \times mst$ and is dyadic of order s, thus requires only $kmst/s = kmt$ elements of the base field for storage. The private key is simply the pair (v, y), consisting of $2n$ elements of $\mathbb{F}_{q^m}$. Finally, the ciphertext is the pair $(\mathbf{c}, \mathbf{d})$, that is, a q-ary vector of length n plus 256 bits. This leads to the measurements (in bytes) in Table 4.6.

Note that in our reference code, which is not optimized, we currently allocate a full byte for each element of $\mathbb{F}_{2^6}$ and two bytes for each element of $\mathbb{F}_{2^{12}}$ thus effectively wasting some memory. However, we expect to be able to represent elements more efficiently, namely using three bytes to store either four elements of $\mathbb{F}_{2^6}$ or two elements of $\mathbb{F}_{2^{12}}$. The measurements in Tables 4.6 and 4.7, above, are taken with respect to the latter method. This is not a problem for DAGS_3 and DAGS_5, obviously.

Table 4.6: Storage Requirements in bytes

Parameter Set	Public Key	Private Key	Ciphertext
DAGS_1	8112	2496	656
DAGS_3	11264	4864	1248
DAGS_5	19712	6400	1632

Table 4.7: Communication Bandwidth in bytes.

Message Flow	Transmitted Message	Size		
		DAGS_1	DAGS_3	DAGS_5
$P_1 \to P_2$	G	8112	11264	19712
$P_2 \to P_1$	$(\mathbf{c}, \mathbf{d})$	656	1248	1632

We now move on to analyzing time measurements. We are using the x64 architecture and our measurements used the "cpucycles.h" from SUPERCOP[4]. Table 4.8 gives the results of our measurements represented by the average after running the code 50 times.

Table 4.8: Timings.

Algorithm	Cycles		
	DAGS_1	DAGS_3	DAGS_5
Key Generation	2,540,311,986	4,320,206,006	7,371,897,084
Encapsulation	12,108,373	26,048,972	96,929,832
Decapsulation	215,710,551	463,849,016	1,150,831,538

4.4 — Advantages and Limitations

We presented DAGS, a Key Encapsulation Mechanism based on Quasi-Dyadic Generalized Srivastava codes. As shown in [BBB+18], DAGS is IND-CCA secure in the Random Oracle Model, and in the Quantum Random Oracle Model. Thanks to this feature, it is possible to employ DAGS not only as a key-exchange protocol (for which IND-CPA would be a sufficient requirement), but also in other contexts such as Hybrid Encryption, where IND-CCA is of paramount importance.

Like any scheme based on structured algebraic codes, DAGS is susceptible to algebraic attacks (FOPT etc.); this can be seen as a limitation of the scheme. In fact, to defeat the attacks, we need to respect stringent conditions on the minimal choices of values for the scheme, in particular the size of the fields in use (both the base field q and the extension degree m) and the values t and s. We remark that in many cases an accurate complexity

[4] https://bench.cr.yp.to/supercop.html

analysis of the attack is not available. This forces us to choose conservative parameters, and this can also been seen as a disadvantage of the scheme.

Nevertheless, DAGS is competitive and compares well with other other code-based schemes. These include the classic McEliece approach [BCL$^+$], as well as more recent proposals such as BIKE [ABB$^+$] and BIG QUAKE [BBB$^+$]. The "Classic McEliece" project is an evolution of the well-known McBits [BCS13](based on the work of Persichetti [Per13]), and benefits from a well-understood security assessment [McE78, Nie86, Pra62] but suffers from the usual public key size issue. BIG QUAKE continues the line of work of [BCGO09], and proposes to use quasi-cyclic Goppa codes. Due to the particular nature of the algebraic attacks, it seems harder to provide security with this approach, and the protocol has to use very large parameters in order to do so. Finally, BIKE, a protocol based on QC-MDPC codes, is the result of a merge between two independently published works with similar background, namely CAKE [BGG$^+$17] and Ouroboros [DGZ17]. The scheme possesses some very nice features like compact keys and an easy implementation approach, but has currently some potential drawbacks. In fact, the QC-MDPC encryption scheme on which it is based is susceptible to a reaction attack by Guo, Johansson and Stankovski (GJS) [GJS16], and thus the protocol is forced to employ ephemeral keys. Moreover, due to its non-trivial Decoding Failure Rate (DFR), achieving IND-CCA security is currently infeasible, so that the BIKE protocol only claims to be IND-CPA secure.

Indeed, another advantage of our proposal is that it does not involve any decoding error. This is particularly favorable in a comparison with some lattice-based schemes like [BCNS15], [ADPS16] and [BCD$^+$16], as well as BIKE. No decoding error allows for a simpler formulation and better security bounds in the IND-CCA security proof.

Our public key size is considerably smaller than in Classic McEliece and BIG QUAKE, and similar to that of BIKE. With regard to the latter, we point out that while, for the same security level, DAGS public keys are indeed bigger, our ciphertexts are a lot smaller. This is because DAGS uses much shorter codes than BIKE, and the size of ciphertexts is a direct consequence of this fact. Thus, in the end, the total communication bandwidth is of the same order of magnitude and smaller if keys are reused.
All the objects involved in the scheme are vectors of finite field elements, which in turn are represented as binary strings; thus computations are very fast. The cost of computing the hash functions is minimized thanks to the parameter choice that makes sure the input $\mathtt{frm}-\mathtt{em}$ is only 256 bits. As a result, we expect that it will be possible to implement our scheme efficiently on multiple platforms.

Finally, we would like to highlight that a DAGS-based Key Exchange features an "asymmetric" structure, where the bandwidth cost and computational effort of the two parties are considerably different. In particular, in the flow described in Figure 4.1, **João** benefits from a much smaller message and faster computation (the encapsulation operation), whereas **Maria** has to perform a key generation and a decapsulation (which includes a run of the decoding algorithm), and transmit a larger message (the public matrix). This is suitable for traditional client-server applications where the server side is usually expected to respond to a large number of requests and thus benefits from a lighter computational load. On the other hand, it is easy to imagine an instantiation, with reversed roles, which could be suitable for example in Internet-of-Things (IoT) applications, where it would be beneficial to reduce the burden on the client side, due to its typical processing, memory and energy constraints.

All in all, DAGS offers great flexibility in key exchange applications, which is not the case for traditional key exchange protocols like Diffie-Hellman.

4.5 — SimpleDAGS

The DAGS algorithms, as detailed in the original proposal [BBB$^+$18], follow the "Randomized McEliece" paradigm presented in [NIKM08], which is built upon the McEliece cryptosystem. The fact that this version of McEliece is proved to be IND-CPA secure makes it so that the resulting KEM conversion achieves IND-CCA security tightly, as detailed in [HHK17]. However, to apply the conversion correctly, it is necessary to use multiple random oracles. These are needed to produce the additional randomness required by the paradigm, as well as to convert McEliece into a deterministic scheme (by generating a low-weight error vector from a random seed) and to obtain an additional hash output for the purpose of plaintext confirmation. Even though, in practice, such random oracles are realized using the same hash function (the *SHAKE256* function [Div15] from the Keccak family), the protocol's description ends up being quite cumbersome and hard to follow.

A simpler protocol can be obtained, although, as we will see, not without consequences, using the Niederreiter cryptosystem. We report the new description below. In the description, we follow the same conventions used in the original DAGS specification, using variables n, k, r to denote, respectively, code length, dimension and co-dimension. All vectors are written in boldface, and, for ease of notation, treated as column vectors.

4.5.1 – Algorithm 1. Key Generation

Key generation follows closely the process described in the original DAGS Key Generation. We present here a compact version, and we refer the reader to the description in Section 4.1.1 for further details.

a) Generate dyadic signature $\mathbf{h}$.

b) Build the Cauchy support $(\mathbf{u}, \mathbf{v})$.

c) Form Cauchy matrix $\hat{H}_1 = C(\mathbf{u}, \mathbf{v})$.

d) Build $\hat{H}_i$, $i = 2, \dots t$, by raising each element of $\hat{H}_1$ to the power of i.

e) Superimpose blocks $\hat{H}_i$, for $i = 2, \dots t$ to form matrix $\hat{H}$ as

$$\hat{H} = \begin{pmatrix} \hat{H}_1 \\ \hat{H}_2 \\ \vdots \\ \hat{H}_t \end{pmatrix}.$$

f) Generate vector $\mathbf{z}$ by picking $\lceil \frac{n}{s} \rceil$ random elements $z_{is} \in \mathbb{F}_{q^m}, i = 0, \dots, \lceil \frac{n}{s} \rceil - 1$ and put $z_{is+j} = z_{is}$ for $i = 0, \dots, \lceil \frac{n}{s} \rceil - 1, j = 0, \dots, s-1$.

g) Form $H = \hat{H} \cdot \mathrm{Diag}(\mathbf{z})$.

h) Project H onto $\mathbb{F}_q$ using the co-trace function: call this H_{base}.

i) Write H_{base} as $(B \mid A)$, where A is $r \times r$.

j) Get systematic form $(M \mid I_r) = A^{-1} H_{base}$: call this $\tilde{H}$.

k) Sample a uniform random string $\mathbf{r} \in \mathbb{F}_q^n$.

l) The public key is the matrix $\tilde{H}$.

m) The private key consists of $(\mathbf{u}, A, \mathbf{r})$ and $\tilde{H}$.

The main differences are as follows. First of all, the public key consists of the systematic parity-check matrix $\tilde{H} = (M \mid I_r)$, rather than the generator matrix $G = (I_k \mid M^T)$. Also, the private key only stores $\mathbf{u}$ instead of $\mathbf{v}$ and $\mathbf{y}$, but it includes additional elements, namely the random string $\mathbf{r}$, the submatrix A and $\tilde{H}$ itself[5].

4.5.2 – Algorithm 2. Encapsulation

Encapsulation uses a hash function $\mathcal{H} : \{0,1\}^* \to \{0,1\}^\ell$ to extract the desired symmetric key, ℓ being the desired bit length (commonly 256). Note that inputs to $\mathcal{H}$ are automatically cast to bitstrings. The function is also used to provide plaintext confirmation by appending an additional hash value, as detailed below.

a) Sample $\boldsymbol{e} \xleftarrow{\$} \mathbb{F}_q^n$ of weight w.

b) Set $\mathbf{c} = (\mathbf{c}_0, \mathbf{c}_1)$ where $\mathbf{c}_0 = \tilde{H}\boldsymbol{e}$ and $\mathbf{c}_1 = \mathcal{H}(2, \boldsymbol{e})$.

c) Compute $\mathbf{k} = \mathcal{H}(1, \boldsymbol{e}, \mathbf{c})$.

d) Output ciphertext $\mathbf{c}$; the encapsulated key is $\mathbf{k}$.

4.5.3 – Algorithm 3. Decapsulation

As in every code-based scheme, the decapsulation algorithm consists mainly of decoding; in this case, like in the original DAGS version, we call upon the alternant decoding algorithm (see for example [MS77]).

a) Get syndrome $\mathbf{c}_0'$ corresponding to matrix[6] H' from private key[7].

b) Decode $\mathbf{c}_0$ and obtain $\boldsymbol{e}'$.

c) If decoding fails or $\mathrm{wt}(\boldsymbol{e}') \neq w$, set $b = 0$ and $\boldsymbol{\eta} = \mathbf{r}$.

d) Check that $\tilde{H}\boldsymbol{e}' = \mathbf{c}_0$ and $\mathcal{H}(2, \boldsymbol{e}') = \mathbf{c}_1$. If so, set $b = 1$ and $\boldsymbol{\eta} = \boldsymbol{e}'$.

e) Otherwise, set $b = 0$ and $\boldsymbol{\eta} = \mathbf{r}$.

f) The decapsulated key is $\mathbf{k} = \mathcal{H}(b, \boldsymbol{\eta}, \mathbf{c})$.

The description we just presented follows the guidelines detailed by the “SimpleKEM" construction of [BP18], hence our choice to call this new version “SimpleDAGS". This is one of two aspects in which this variant diverges substantially from the original; we will discuss advantages (and disadvantages) of this new paradigm in the next section. The other different aspect is that using Niederreiter requires a different strategy for decoding, which we describe below.

4.5.4 – Decoding from a Syndrome. In the original version of DAGS, the input to the decoding algorithm is, as is commonly the case is coding theory, a noisy codeword. The alternant decoding algorithm consists of three distinct steps. First, it is necessary to compute the syndrome of the received word, with respect to the alternant parity-check matrix; this is represented as a polynomial $S(z)$. Then, the algorithm uses the syndrome to compute the *error locator polynomial* $\sigma(z)$ and the *error evaluator polynomial* $\omega(z)$, by solving the *key equation* $\omega(z)/\sigma(z) = S(z) \mod z^r$. Finally, finding the roots of the two polynomials reveals, respectivelly, the locations and values (if the code is not binary) of

[5] This is mostly a formal difference, since $\tilde{H}$ is in fact the public key.

[6] In alternant form.

[7] See below for details.

the errors. Actually, as shown in Section 4.1.1, it is possible to speed up decapsulation by incorporating the first step of the decoding algorithm in the reconstruction of the alternant matrix, i.e. the syndrome is computed *on the fly*, while the alternant matrix is built.

We now explain how to perform alternant decoding when the input is a syndrome, rather than a noisy codeword, as in Algorithm 3 above. In this case, we do not need to reconstruct the alternant matrix itself, but rather to transform the received syndrome to the syndrome corresponding to the alternant matrix. This consists of two steps. First, remember that the public key $\tilde{H}$ is the systematic form of the matrix H_{base}. This is obtained from the quasi-dyadic parity-check matrix H, whose entries are in $\mathbb{F}_{q^m}$, by projecting it onto the base field $\mathbb{F}_q$. The projection is performed using the co-trace function and a basis for the extension field, say $\{\beta_1, \dots, \beta_m\}$. Recall from Section 2.2, that the co-trace function works similarly to the trace function, by writing each element of $\mathbb{F}_{q^m}$ as a vector whose components are the coefficients with respect to the basis $\{\beta_1, \dots, \beta_m\}$. However, instead of writing the components on m successive rows, the co-trace function distributes them over the rows at regular intervals, r at a time. More precisely, if $\mathbf{a} = (a_1, a_2, \dots, a_r)^T$ is a column of H, the corresponding column $\mathbf{a}' = (a'_1, a'_2, \dots, a'_{rm})^T$ of H_{base} will be formed by writing the components of each a_i in positions $a'_i, a'_{r+i}, \dots, a'_{r(m-1)+i}$, for all $i = 1, \dots, m$.

The first step consists of transforming the received syndrome $\mathbf{c}_0 = \tilde{H}\mathbf{e}$ into $H\mathbf{e}$. For this, we need to multiply the syndrome by A to obtain $A\tilde{H}\mathbf{e} = AA^{-1}H_{base}\mathbf{e} = H_{base}\mathbf{e}$. Then we reverse the projection process and "bring back" the syndrome on the extension field. This is immediate when operating directly on the matrices, but a little less intuitive when starting from a syndrome. It turns out that it is still possible to do that, by using again the basis $\{\beta_1, \dots, \beta_m\}$ and summing up the results. Namely, it is enough to collect all the components $s_i, s_{r+i}, \dots, s_{r(m-1)+i}$ of the syndrome $\mathbf{s} = H_{base}\mathbf{e}$ and multiply the resulting vector with the vector $(\beta_1, \dots, \beta_m)$. This maps the vector of components back to its corresponding element in $\mathbb{F}_{q^m}$ and it is immediate to check that this process yields $H\mathbf{e}$.

The second step consists of relating the newly-obtained syndrome to the alternant parity-check matrix H'. Since this is just another parity-check for the same code, it is possible to obtain one from the other via an invertible matrix. In particular, for GS codes we have $H = CH'$, where the $r \times r$ matrix C can be obtained using $\mathbf{u}$. Namely, the r rows of C correspond to the coefficients of the polynomials $g_1(x), \dots, g_r(x)$, where we have

$$g_{(l-1)t+i} = \frac{\prod_{j=1}^{s}(x-u_j)^t}{(x-u_l)^i}$$

for $l = 1, \dots, s$ and $i = 1, \dots, t$. To complete the second step, it is enough to compute C and then $C^{-1}H\mathbf{e}$. The resulting syndrome is ready to be decoded.

4.5.5 – Consequences. There are some notable consequences to keep in mind when switching to the SimpleDAGS variant. First of all, the change in the KEM conversion not only makes the protocol simpler, but has additional advantages. The reduction is tight in the ROM, and the introduction of the plaintext confirmation step provides an extra layer of defense, at the cost of just one additional hash value. This is similar to what is done in the Classic McEliece submission [BCL+]. Moreover, the use of *implicit rejection* and a “quiet" KEM (i.e. such that the output is always a session key) further simplifies the API, and is an incentive to design constant-time algorithms, without needing extra machinery or stronger assumptions, as explained in Sections 14 and 15 of [BP18].

On the other hand, using Niederreiter has a negative impact on the overall performance of the scheme. The cost of the first step of decoding, detailed above, is comparable to that of reconstructing H' (and computing the syndrome) in the original DAGS, but there is an additional cost in the multiplication by A. Moreover, inverting the matrix C in the second step is expensive, and would slow down decapsulation considerably. Alternatively, one could delegate some computation time to the key generation algorithm, and store C^{-1} as private key; this would preserve the efficiency of the decapsulation but noticeably increase the size of the private key. Either way, there is a clear a tradeoff at hand, sacrificing performance and efficiency in favor of a simpler description and tighter security. It therefore falls to the user's discretion whether original DAGS or SimpleDAGS is the best variant to be employed for the purpose.

4.6 — Improved Resistance

It is natural to think that introducing additional algebraic structure like QD in a scheme based on algebraic codes (such as Goppa or GS) can give an adversary more power to perform a structural attack. This is the case of the well-known FOPT attack [FOPT10], and successive variants [FOP+16], which exploit this algebraic structure to solve a multivariate system of equations and reconstruct an alternant matrix which is equivalent to the private key. A detailed analysis of such attacks, and countermeasures, is given in the original DAGS paper [BBB+18]. In 2018, Barelli and Couvreur presented a structural attack aimed precisely at DAGS [BC18], which is very successful against the original parameters as presented in Section 4.2.4. Moreover, early in 2019 another attack against DAGS_1 was presented in [BBCO19] and thanks to this analysis, it is now possible to see that these parameters are particularly vulnerable to the second attack approach. Table 4.9 shows the cost for the attack presented in [BBCO19]. In what follows, we will briefly explain the reason for this, and present a new choice of parameters for DAGS_1.

Table 4.9: First set of parameters for DAGS.

Security Level	q	m	n	k	s	t	w	Attack cost in [BBCO19]
1	2^5	2	832	416	2^4	13	104	2^{44}
3	2^6	2	1216	512	2^5	11	176	2^{44}
5	2^6	2	2112	704	2^6	11	352	2^{33}

As mentioned in Section 4.2.4, the success of the attack strongly depends on the dimension of the invariant code $\mathcal{D}$, which is given by $k_0 - c$, where $k_0 = k/s$ is the number of row blocks and $c = 2q/s$ was defined above. For the parameters in question, we have $k_0 = 26$ and $c = 8$ and therefore this dimension is 18. This leads to an imbalance in the ratio of the number of equations to the number of variables. The former are given by $(k_0 - c)(n_0 - k_0 - 1)$, where $n_0 = n/s$ is the number of column blocks, while the latter consists of the $(k_0 - c)c$ variables of the **U** type and the $n_0 - k_0 + c + \log s - 3$ variables of the **V** type that define the bilinear system. Therefore we obtain 450 equations in 179 total variables, and this ratio is about 2.5. The authors then show how the system can be solved by specializing the **U** variables to obtain linear equations, for a total cost of approximately 2^{111} operations, which is below the claimed security level. Actually, this cost can be further reduced following a hybrid approach that combines exhaustive search and Gröbner bases, to a total of 2^{83}. The crucial point is that a ratio of 2.5 is quite high, and this is what makes the attack feasible. In contrast, the updated DAGS_5 parameters produce a ratio of 1.1 which is too low for the attack to work (the system has too many variables) while the situation for DAGS_3 is even more extreme, since in this case $c = k_0$ and therefore $\mathcal{D}$ does not even exist. In this case, the authors suggest to use the dual code instead, therefore replacing k_0 with $n_0 - k_0$ in all the above formulas. In principle, this makes the attack applicable, but the parameters yield a ratio of 0.7 which is again too low to be of any use. We insist on this crucial point to select our next choice of parameters for DAGS_1 (where “N.A." stands for “not applicable").

Table 4.10: New DAGS Parameters.

Security Level	q	m	n	k	s	t	w	Attack cost in [BC18]	Attack cost in [BBCO19]
1	2^8	2	704	352	2^4	11	88	$\approx 2^{542}$	N.A.
3	2^8	2	1216	512	2^5	11	176	$\approx 2^{288}$	N.A.
5	2^8	2	1600	896	2^5	11	176	$\approx 2^{289}$	N.A.

Table 4.11: New storage requirements in bytes for the revised version of DAGS with the parameters from Table 4.10.

Parameter Set	Public Key	Private Key	Ciphertext
DAGS_1	7744	2816	736
DAGS_3	11264	4864	1248
DAGS_5	19712	6400	1632

Note that we have only changed the parameters for DAGS_1, but we have chosen to report the other two sets too, in order to provide a complete view. With this new choice, we have $k_0 = 22$ and $c = 32$ and therefore $\mathcal{D}$ does not exist; in fact, not even its dual exists, since in this case $k_0 = n_0 - k_0$. This completely defeats the second attack approach, while the first approach would produce a ridiculously large complexity ($\approx 2^{542}$, see above), and we therefore feel comfortable claiming that DAGS_1 is now safe against all known attacks.

In the end, we can add the Barelli-Couvreur attack(s) to the set of constraints on the selection of parameters, and we are very thankful to the authors of [BC18] and [BBCO19] for the detailed and careful analysis of the attack techniques.

4.6.0.1 – Binary DAGS. Parameters in schemes based on QD-GS codes are a carefully balanced machine, needing to satisfy many constraints. First of all, we would like the code dimension $k = n - mst$ to be approximately $n/2$, since rate close to $1/2$ is an optimal choice in many aspects (for instance, against ISD). Secondly, the dyadic order s, which has to be a power of 2, should be as big as possible, to obtain the most reduction in key size (but not too big, to avoid the Barelli-Couvreur attack). On the other hand, the extension degree m and the number of blocks t need to be large enough to have $mt > 21$, in order to avoid FOPT. Of course, m, s and t cannot all be large at the same time otherwise the dimension k would become trivial. Moreover, it is possible to observe that the best outcome is obtained when m and t are of opposite magnitude (one big and one small) rather than both of “medium" size. Now, since s and t also determine the number of correctable errors, t cannot be too small either, while a small m is helpful to avoid having to work on very large extension fields. Note that q^m still needs to be at least as big as the code length n (since the support elements are required to be distinct). After all these considerations, the result is that, in previous literature [Per12a, CHP12], the choice of parameters was oriented towards large base field q and small $m = 2$, with s ranging from 2^4 to 2^6, and t chosen accordingly. We now investigate the consequences of choosing parameters in the opposite way.

Choosing large m and small t allows q to be reduced to the minimum, and more precisely q could be even 2 itself, meaning binary codes are obtained. Binary codes were already considered in the original QD Goppa proposal by Misoczki and Barreto [MB09], where they ended up being the only safe choice. The reason for this is that larger base fields mean m can be chosen smaller (and in fact, must, in order to avoid working on prohibitively large extension fields). This in turn means FOPT is very effective (remember that there is no parameter t for Goppa codes), so in order to guarantee security one had to choose m as big as possible (at least 16) and consequently $q = 2$. Now in our case, if t is small, s must be bigger (for error-correction purposes), and this pushes n and k up accordingly. We present below our binary parameters (Table 4.12). Table 4.13 refers to the memory requirements in bytes, the names refer to DAGS-Binary and the length and dimension of the code.

Table 4.12: DAGS-Bin Parameters.

Security Level	q	m	n	k	s	t	w	Attack cost in [BC18]	Attack cost in [BBCO19]
1	2	13	6400	3072	2^7	2	128	N.A.	N.A.
3	2	14	11520	4352	2^8	2	256	N.A.	N.A.
5	2	14	14080	6912	2^8	2	256	N.A.	N.A.

Table 4.13: Memory Requirements for Binary DAGS (bytes).

Parameter Set	Public Key	Private Key	Ciphertext
DAGS-Bin64003072	9984	20800	832
DAGS-Bin115204352	15232	40320	1472
DAGS-Bin140806912	24192	49280	1792

The parameters are chosen to stay well clear of the algebraic attacks such as FOPT. In particular, using binary parameters should entirely prevent the latest attack by Barelli and Couvreur. In this case we have $m \gg 2$, and it is not yet clear whether the attack is applicable in the first place. However, even if this was the case, the complexity of the attack, which currently depends on the quantity q/s, should depend instead on mq^{m-1}/s. It is obvious that, with our choice of parameters, the attack would be completely infeasible in practice.

Note that, in order to be able to select binary parameters, it is necessary to choose longer codes (as explained above), which end up in slightly larger public keys: these are about 1.3 times larger than those of the original (non-binary) DAGS. On the other hand, the binary base field should bring clear advantages in terms of arithmetic, and result in a much more efficient implementation. All things considered, this variant should be seen as yet another tradeoff, in this case sacrificing public key size in favor of increased security and efficient implementation.

4.7 — Revised Implementation Results

In this section we present the results obtained in our revised implementation, i.e., using the parameters in Table 4.10. Our efforts focused on several aspects of the code, with the ultimate goal of providing faster algorithms, but which are also clearer and more accessible. Moreover, we incorporate a dedicated version of the Karatsuba multiplication algorithm as detailed in Chapter 3, applied to the quasi-dyadic case, which further boosts the efficiency of encapsulation (where all objects are quasi-dyadic). Finally, we “cleaned up" and polished our C code, to ensure it is easier to understand for external auditors. Below, we report timings obtained for our revised implementation (Table 4.15), as well as the measurements previously obtained for the reference code (Table 4.14), for ease of comparison. We remark that all these numbers refer to the updated DAGS parameters (i.e. those presented in Table 4.10). The timings were acquired running the code 100 times and taking the average. We used CLANG compiler version 8.0.0 and the compilation flags -O3 -g3 -Wall -march=native -mtune=native -fomit-frame-pointer -ffast-math. Moreover, we used the processor Intel(R) Core(TM) i5-5300U CPU @ 2.30GHz. The code in C is available at `https://git.dags-project.org/dags/dags` and a toy example in SAGE is available at `https://git.dags-project.org/gustavo/DAGS_ref_sage`.

Table 4.14: Timings for the reference code submitted to NIST.

Algorithm	Cycles		
	DAGS_1	DAGS_3	DAGS_5
Key Generation	2,540,311,986	4,320,206,006	7,371,897,084
Encapsulation	12,108,373	26,048,972	96,929,832
Decapsulation	215,710,551	463,849,016	1,150,831,538

Table 4.15: Timings for revised implementation.

Algorithm	Cycles		
	DAGS_1	DAGS_3	DAGS_5
Key Generation	408,342,881	1,368,126,687	2,061,117,168
Encapsulation	5,061,697	14,405,500	35,655,468
Decapsulation	192,083,862	392,435,142	388,316,593

Chapter 5

Root Finding over $\mathbb{F}_{2^m}$

As was mentioned in Chapter 2, finding roots of the Error Locator Polynomial (ELP) is an important step in the decryption of McEliece-like cryptosystems. One of the requirements for those proposals is that they are resistant to all known cryptanalysis methods. However, even if a scheme is immune to such attacks, it may be subject to attacks related to its implementation. In particular, implementations need to avoid side-channel attacks.[1]

There are different ways to apply side-channel attacks to a cryptosystem. As an example, an attacker can measure the execution time of the operations performed by an algorithm and, based on these measures, estimate some secret information of the scheme. This approach is thriving even in a data communication network environment. Bernstein, for instance, demonstrated how to recover AES keys by doing remote timing attacks on the cache "access speed" [Ber05].

In code-based cryptography, timing attacks on the decryption process are mostly done during the computation and factorization of the ELP as shown by [SSMS09]. The attack is usually done during the polynomial evaluation process, while computing the roots of the ELP. This attack was demonstrated first in [SSMS09] and later in an improved version in [BCDR17].

[Str12] demonstrates algorithms to efficiently find roots in code-based cryptosystems. However, the author shows only timings in different types of implementations and selects the one that has the least timing variability. In other words, the author does not present an algorithm to find the roots in constant time and eliminate a remote timing attack as remarked in Section 6 of [Str13]. In our work, we use strategies to make the execution time of those algorithms constant. The first and most important one is to write the algorithms iteratively, eliminating all variable-length recursions. We also use permutations and simulated operations to uncouple possible measurements of the side effects of the data being measured. The implementation for finding roots in [Cho17] uses the Fast Fourier Transform (FFT), which is efficient, but is built and optimized for $\mathbb{F}_{2^{13}}$. In this chapter, we aim at developing a more generic implementation that does not require specific optimization in the finite-field arithmetic.

In summary, we show how to perform a remote timing attack on a code-based key encapsulation mechanism called BIGQUAKE which was submitted to NIST [BBB$^+$]. The

[1] This chapter is based on a paper accepted to Latincrypt 2019 in a joint work with Douglas Martins and Ricardo Custódio [MBC19]. Part of this work was done while Gustavo was at CryptoExperts for an internship.

attack uses information on timing of finding roots of a polynomial over $\mathbb{F}_{2^m}$. The original implementation submitted to NIST uses a variation of the Berlekamp Trace Algorithm (BTA) to find roots in the ELP. We provide other methods to find roots showing mathematically that is possible to avoid timing attacks. At the end, we make a comparison between the methods, showing the number of CPU cycles required for each of our implementations.

5.1 — BIGQUAKE Key Encapsulation Mechanism & Attack

BIGQUAKE (BInary Goppa QUAsi-cyclic Key Encapsulation) [BBB+] uses binary Quasi-cyclic (QC) Goppa codes in order to accomplish a KEM between two distinct parties. Instead of using binary Goppa codes as showed in Section 2.2, BIGQUAKE uses QC Goppa codes, which have the same properties as Goppa codes but allow smaller keys. Furthermore, BIGQUAKE aims to be IND-CCA [BP18], which makes the attack scenario in Section 5.1.1 meaningful.

Let us suppose that Alice and Bob (A and B respectively) want to share a session secret key K using BIGQUAKE. Then Bob has published his public key and Alice needs to follow the encapsulation mechanism. The function $\mathcal{F}$ takes an arbitrary binary string as input and returns a word of weight t, i.e $\mathcal{F} : \{0,1\}^* \rightarrow \{x \in \mathbb{F}_2^n | w_h(x) = t\}$. The detailed construction of the function $\mathcal{F}$ can be found in subsection 3.4.4 in [BBB+]. $\mathcal{H} : \{0,1\}^k \rightarrow \{0,1\}^s$ is a hash function. The function $\mathcal{H}$ in the original implementation is SHA-3. The encapsulation mechanism can be described as:

a) A generates a random $m \in \mathbb{F}_2^s$;
b) Generate $e \leftarrow \mathcal{F}(m)$;
c) A sends $c \leftarrow (m \oplus \mathcal{H}(e), H \cdot e^T, \mathcal{H}(m))$ to B;
d) The session key is defined as: $K \leftarrow \mathcal{H}(m, c)$.

After Bob receives c from Alice, he initiates the decapsulation process:

a) B receives $c = (c_1, c_2, c_3)$;
b) Using the secret key, Bob decodes c_2 to e' with $w_h(e') \leqslant t$ such that $c_2 = H \cdot e'^T$;
c) B computes $m' \leftarrow c_1 \oplus \mathcal{H}(e')$;
d) B computes $e'' \leftarrow \mathcal{F}(m')$;
e) If $e'' \neq e'$ or $\mathcal{H}(m') \neq c_3$ then B aborts.
f) Else, B computes the session key: $K \leftarrow \mathcal{H}(m', c)$.

After Bob executes the decapsulation process successfully, both parties of the protocol agree on the same session secret key K.

5.1.1 – Attack Description. In [SSMS09], the attack exploits the fact that flipping a bit of the error e changes the Hamming weight w and per consequence the timing for its decryption. If we flip a position that contains an error ($e_i = 1$) then the error will be removed and the time of computation will be shorter. However, if we flip a bit in a wrong position ($e_i = 0$) then it will add another error, which will increase the decryption time or make decryption fail. The attack described in [BCDR17] exploits the root finding in the polynomial ELP. It takes advantage of sending ciphertexts with fewer errors than expected, which generate an ELP with degree less than t, resulting in less time for finding roots. We explore both ideas applied to the implementation of BIGQUAKE that the decryption will fail ultimately as $e' \neq e$ due to the bitflip. However, the timing variation is observable earlier.

Algorithm 15 is the direct implementation of the attack proposed in [SSMS09]. We reused the attack presented to show that the attack still works in current implementations such as BIGQUAKE when the root finding procedure is vulnerable to remote timing attacks.

```
Algorithm 15: Attack on ELP.
Data: $n$-bit ciphertext $c$, $t$ as the number of errors and precision parameter $M$
Result: Attempt to obtain an error vector $e$ hidden in $c$.
$e \leftarrow [0, \dots, 0]$;
for $i \leftarrow 0$ to $n - 1$ do
    $T \leftarrow 0$;
    $c' \leftarrow c \oplus \text{setBit}(n, i)$;
    $time_m \leftarrow 0$;
    for $j \leftarrow 0$ to $M$ do
        $time_s \leftarrow time()$;
        $\text{decrypt}(c')$;
        $time_e \leftarrow time()$;
        $time_m \leftarrow time_m + (time_e - time_s)$;
    end
    $T \leftarrow time_m / M$;
    $L \leftarrow (T, i)$;
end
Sort $L$ in descending order of $T$;
for $k \leftarrow 0$ to $t - 1$ do
    $index \leftarrow L[k].i$;
    $e[index] \leftarrow 1$;
end
return $e$;
```

After finding the positions of the errors, one needs to verify if the error e' found is the correct one, and then recover the message m. In order to verify for correctness, one can check e' by computing $c_1 \oplus \mathcal{H}(e') \oplus m = m'$ and if c_3 is equal to $\mathcal{H}(m')$ and if $e' = \mathcal{F}(m')$ as in decryption. As mentioned in Subsection 5.1, the ciphertext is composed of $c = (m \oplus \mathcal{H}(e), H \cdot e^T, \mathcal{H}(m))$ or $c = (c_1, c_2, c_3)$.

5.1.1.1 – Notes about BIGQUAKE implementation and attack. In our tests, that is, when we ran Algorithm 15 we noticed that BIGQUAKE fails after inserting one additional error. In the file "kem.c" there is the encryption and decryption process. We extract the part that causes the failure from that file. Listing 5.1 shows initialiazion of the variable "error" with a random value of the memory since it is used the function "malloc" and if the message is not the correct one the function "m2error" does not override the values in "error" and it breaks the function "decrypt_nied". One could exploit this idea using the attack proposed in [VDvT02]. However, we did not consider this scenario since BIGQUAKE performs a CCA transform.

We recall that the function "malloc" is a memory allocation function and according to the C reference code the function returns a pointer to the lowest (first) byte in the allocated memory block that is suitably aligned for any object type with fundamental alignment, for more details see https://en.cppreference.com/w/c/memory/malloc.

```c
//Construct an error e from m
int *error = (int *) malloc(NB_ERRORS*sizeof(int));
m2error(m, error);

//Encrypt e (Niederreiter)
unsigned char *syndrome = malloc(SYNDROME_BYTES*sizeof(unsigned char));
decrypt_nied(syndrome, error, (unsigned char *) sk);
```

Listing 5.1: Code snippet of BIGQUAKE decryption process.

The problem has an easy solution as we show in Listing 5.2. The solution is to initialize the variable with zeros. In this way, if the function ‘m2error”does not override the values in “error” then “‘error” will contains zeros and it does not cause failure in the function “decrypt_nied”.

```c
//Construct an error e from m
int error[NB_ERRORS] = { 0 };
m2error(m, error);

//Encrypt e (Niederreiter)
unsigned char *syndrome = malloc(SYNDROME_BYTES*sizeof(unsigned char));
decrypt_nied(syndrome, error, (unsigned char *) sk);
```

Listing 5.2: Code snippet of BIGQUAKE decryption process with the fix.

In our attack, we selected the precision parameter M as 500 since it was the one that showed the more precise results without taking a long time of computation. In our tests, it took $\approx$ 17 minutes for recovering one correct message m. We used an Intel® Core(TM) i5-5300U CPU @ 2.30GHz and running everything locally. The implementation of the attack is written in C and it is avaiable at: https://git.dags-project.org/gustavo/roots_finding/src/master/attack_bigquake.

5.1.2 – Constant-time $\mathbb{F}_{2^m}$ operations. In our analysis, we noticed that the original implementation of BIGQUAKE uses log and antilog tables for computing multiplications and inversions. These look-up tables give a speed up in those operations. However, this approach is subject to cache attacks in a variation of [BHLY16] where the attacker tries to produce cache misses and recover some secret data from the operations.

We want to avoid the use of look-up tables; we made a constant time implementation for multiplication and inversion. We used a similar approach as [Cho17] for finite field operations. Listing 5.3 shows the multiplication in constant time between two elements over $\mathbf{F}_{2^{12}}$ followed by the reduction of the result by the irreducible polynomial $f(x) = x^{12}+x^6+x^4+x+1$. The inversion in finite fields can be computed by raising an element a to the power 2^{m-2} as it is shown in Listing 5.4. The squarings are implemented following the implementation provided in [Cho17].

```c
#include <stdint.h>
```

```c
typedef uint16_t gf;
gf gf_q_m_mult(gf in0, gf in1) {
  uint64_t i, tmp, t0 = in0, t1 = in1;
  //Multiplication
  tmp = t0 * (t1 & 1);
  for (i = 1; i < 13; i++)
    tmp ^= (t0 * (t1 & (1 << i)));
  //reduction
  tmp = tmp & 0x7FFFFF;
  //first step of reduction
  gf reduction = (tmp >> 12);
  tmp = tmp & 0xFFF;
  tmp = tmp ^ (reduction << 6);
  tmp = tmp ^ (reduction << 4);
  tmp = tmp ^ reduction << 1;
  tmp = tmp ^ reduction;
  //second step of reduction
  reduction = (tmp >> 12);
  tmp = tmp ^ (reduction << 6);
  tmp = tmp ^ (reduction << 4);
  tmp = tmp ^ reduction << 1;
  tmp = tmp ^ reduction;
  tmp = tmp & 0xFFF;
  return tmp;
}
```

Listing 5.3: Multiplication of two elements in $\mathbb{F}_{2^{12}}$

```c
#include <stdint.h>
typedef uint16_t gf;
gf gf_inv(gf in) {
  gf tmp_11 = 0;
  gf tmp_1111 = 0;
  gf out = in;
  out = gf_sq(out); //a^2
  tmp_11 = gf_mult(out, in); //a^2*a = a^3
  out = gf_sq(tmp_11); //(a^3)^2 = a^6
  out = gf_sq(out); // (a^6)^2 = a^12
  tmp_1111 = gf_mult(out, tmp_11); //a^12*a^3 = a^15
  out = gf_sq(tmp_1111); //(a^15)^2 = a^30
  out = gf_sq(out); //(a^30)^2 = a^60
  out = gf_sq(out); //(a^60)^2 = a^120
  out = gf_sq(out); //(a^120)^2 = a^240
  out = gf_mult(out, tmp_1111); //a^240*a^15 = a^255
  out = gf_sq(out); // (a^255)^2 = 510
  out = gf_sq(out); //(a^510)^2 =  1020
  out = gf_mult(out, tmp_11); //a^1020*a^3 = 1023
  out = gf_sq(out); //(a^1023)^2 = 2046
  out = gf_mult(out, in); //a^2046*a = 2047
  out = gf_sq(out); //(a^2047)^2 = 4094
  return out;
}
```

Listing 5.4: Inversion of an element in $\mathbb{F}_{2^{12}}$

5.2 — Root Finding Methods

As argued, the leading cause of information leakage in the decoding algorithm is the process of finding the roots of the ELP. In general, the time needed to find these roots varies, often depending on the roots themselves. Thus, an attacker who has access to the decoding time can infer these roots, and hence get the vector of errors e. Next, we propose modifications in four of these algorithms to avoid the attack presented in Subsection 5.1.1.

Strenzke [Str12] presents an algorithm analysis for fast and secure root finding for code-based cryptosystems. He uses as a basis for his results the implementation of "Hymes" [BS08]. We rewrote the operation in Hymes without tables and analyzed each line of code from the original implementations, taking care of modifying them in order to eliminate processing that could indicate root-dependent execution time. The adjustments were made in the following algorithms to find roots: exhaustive search, linearized polynomials, Berlekamp trace algorithm (BTA), and successive resultant algorithm (SRA).

In this work, we use the following notation: given a univariate polynomial p, of degree d and coefficients over a finite field, one needs to find its roots. In our case, we are concerned about binary fields, i.e., $\mathbb{F}_{2^m}$. Additionally, we assume that all the factors of p are linear and distinct.

5.2.1 – Exhaustive search. Exhaustive search is a direct method, in which the evaluation of p for all the elements in $\mathbb{F}_{2^m}$ is performed. A root is found whenever the evaluation result is zero. This method is acceptable for small fields and can be made efficient with a parallel implementation. Algorithm 16 describes this method.

As can be seen in Algorithm 16, this method leaks information. This is because whenever a root is found, i.e., $dummy = 0$, an extra operation is performed. In this way, the attacker can infer from this additional time that a root was found, thus providing ways to obtain data that should be secret.

One solution to avoid this leakage is to permute the elements of vector A. Using this technique, an attacker can identify the extra operation, but without learning any secret information. In our case, we use the Fisher-Yates shuffle [Bla05] for shuffling the elements of vector A. In [WSN18], the authors show an implementation of the shuffling algorithm safe against timing attacks. Algorithm 17 shows the permutation of the elements and the computation of the roots.

Using this approach, we add one extra step to the algorithm. However, this permutation blurs the sensitive information of the algorithm, making the usage of Algorithm 17 slightly harder for the attacker to acquire timing leakage.

The main costs for Algorithm 16 and Algorithm 17 are the polynomial evaluation C_{pol_eval}. Since we need to evaluate each element in A, it is safe to assume that the total cost is:

$$C_{exh} = n \cdot C_{pol\_eval}. \tag{5.1}$$

Algorithm 16: Exhaustive search algorithm for finding roots of a univariate polynomial over $\mathbb{F}_{2^m}$.

```
Data: $p(x)$ as univariate polynomial over $\mathbb{F}_{2^m}$ with $d$ roots, $A = [a_0, \ldots, a_{n-1}]$
      as the support of the code in $\mathbb{F}_{2^m}$, $n$ as the length of the code.
Result: $R$ as a set of roots of $p(x)$.
R ← ∅;
for i ← 0 to n − 1 do
    dummy ← p(A[i]);
    if dummy == 0 then
        R.add(A[i]);
    end
end
return R;
```

Algorithm 17: Exhaustive search algorithm with a countermeasure for finding roots of an univariate polynomial over $\mathbb{F}_{2^m}$.

```
Data: $p(x)$ as univariate polynomial over $\mathbb{F}_{2^m}$ with $d$ roots, $A = [a_0, \ldots, a_{n-1}]$
      as the support of the code in $\mathbb{F}_{2^m}$, $n$ as the length of the code.
Result: $R$ as a set of roots of $p(x)$.
permute(A);
R ← ∅;
for i ← 0 to n − 1 do
    dummy ← p(A[i]);
    if dummy == 0 then
        R.add(A[i]);
    end
end
return R;
```

We can go further and express the cost for one polynomial evaluation by the number of operations in finite fields. In our implementation[2] the cost is determined by the degree d of the polynomial and basic finite-field operations such as addition and multiplication. As a result, the cost for one polynomial evaluation is:

$$C_{pol\_eval} = d(C_{gf\_add} + C_{gf\_mul}). \tag{5.2}$$

5.2.2 – Linearized polynomials. The second countermeasure proposed is based on linearized polynomials. The authors in [FT02] propose a method to compute the roots of a polynomial over $\mathbb{F}_{2^m}$, using a particular class of polynomials, called linearized polynomials. In [Str12], this approach is a recursive algorithm which the author calls “dcmp-rf”. In our solution, however, we present an iterative algorithm. Linearized polynomials can be defined as follows:

[2] available in https://git.dags-project.org/gustavo/roots_finding

Definition 5.1. A polynomial $\ell(y)$ over $\mathbb{F}_{2^m}$ is called a linearized polynomial if

$$\ell(y) = \sum_i c_i y^{2^i}, \tag{5.3}$$

where $c_i \in \mathbb{F}_{2^m}$.

In addition, from [TJR01], we have Lemma 5.2 that describes the main property of linearized polynomials for finding roots.

Lemma 5.2. Let $y \in \mathbb{F}_{2^m}$ and let $\alpha^0, \alpha^1, \ldots, \alpha^{m-1}$ be a standard basis over $\mathbb{F}_2$. If

$$y = \sum_{k=0}^{m-1} y_k \alpha^k, y_k \in \mathbb{F}_2 \tag{5.4}$$

and $\ell(y) = \sum_j c_j y^{2^j}$, then

$$\ell(y) = \sum_{k=0}^{m-1} y_k \ell(\alpha^k). \tag{5.5}$$

We call $A(y)$ over $\mathbb{F}_{2^m}$ an affine polynomial if $A(y) = \ell(y) + \beta$ for $\beta \in \mathbb{F}_{2^m}$, where $\ell(y)$ is a linearized polynomial.

We can illustrate a toy example to understand the idea behind finding roots using linearized polynomials.

Example 5.2.1. Let us consider the polynomial $f(y) = y^2 + (\alpha^2+1)y + (\alpha^2+\alpha+1)y^0$ over $\mathbb{F}_{2^3}$ and α represented as elements in $\mathbb{F}_2[x]/(x^3+x^2+1)$. Since we are trying to find roots, we can write $f(y)$ as

$$y^2 + (\alpha^2+1)y + (\alpha^2+\alpha+1)y^0 = 0$$

or

$$y^2 + (\alpha^2+1)y = (\alpha^2+\alpha+1)y^0 \tag{5.6}$$

We can point that on the left-hand side of Equation 5.6, $\ell(y) = y^2 + (\alpha^2+1)y$ is a linearized polynomial over $\mathbb{F}_{2^3}$ and Equation 5.6 can be expressed just as

$$\ell(y) = \alpha^2 + \alpha + 1 \tag{5.7}$$

If $y = y_2\alpha^2 + y_1\alpha + y_0 \in \mathbb{F}_{2^3}$ then, according to Lemma 5.2, Equation 5.7 becomes

$$y_2\ell(\alpha^2) + y_1\ell(\alpha) + y_0\ell(\alpha^0) = \alpha^2 + \alpha + 1 \tag{5.8}$$

We can compute $\ell(\alpha^0), \ell(\alpha)$ and $\ell(\alpha^2)$ using the left hand side of Equation 5.6 and we have the following values

$$\begin{aligned} \ell(\alpha^0) &= (\alpha^0)^2 + (\alpha^2+1)(\alpha^0) = \alpha^2 + 1 + 1 = \alpha^2 \\ \ell(\alpha) &= (\alpha)^2 + (\alpha^2+1)(\alpha) = \alpha^2 + \alpha^2 + \alpha + 1 = \alpha + 1 \\ \ell(\alpha^2) &= (\alpha^2)^2 + (\alpha^2+1)(\alpha^2) = \alpha^4 + \alpha^4 + \alpha^2 = \alpha^2. \end{aligned} \tag{5.9}$$

A substitution of Equation 5.9 into Equation 5.8 gives us

$$(y_2 + y_0)\alpha^2 + (y_1)\alpha + (y_1)\alpha^0 = \alpha^2 + \alpha + 1. \tag{5.10}$$

Equation 5.10 can be expressed as a matrix in the form

$$\begin{pmatrix} y_2 & y_1 & y_0 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 \end{pmatrix}. \tag{5.11}$$

If one solves simultaneously the linear system in Equation 5.11 then the results are the roots of the polynomial given in Equation 5.6. From Equation 5.10, one observes that the solutions are $y = 110$ and $y = 011$, which can be translated to $\alpha^2 + \alpha$ and $\alpha + 1$.

The authors in [FT02] provide a generic decomposition for finding affine polynomials. They show that each polynomial in the form $F(y) = \sum_{j=0}^{t} f_j y^j$ for $f_j \in \mathbb{F}_{2^m}$ can be represented as

$$F(y) = f_3 y^3 + \sum_{i=0}^{\lceil (t-4)/5 \rceil} y^{5i}(f_{5i} + \sum_{j=0}^{3} f_{5i+2^j} y^{2^j}). \tag{5.12}$$

After that, we summarize all the steps as Algorithm 18. The function "generate(m)" refers to the generation of the elements in $\mathbb{F}_{2^m}$ using Gray codes, see [Sav97] for more details about Gray codes.

Algorithm 18 presents a countermeasure in the last steps of the algorithm, i.e., we added a dummy operation for blinding if X[j] is a root of polynomial $F(x)$. Using Algorithm 18, the predominant cost for its implementation is:

$$C_{lin} = m(C_{gf\_pow} + C_{pol\_eval}) + 2^m(C_{gf\_pow} + 2C_{gf\_mul}). \tag{5.13}$$

The linearized polynomials methods precompute the polynomials to later on perform a brute force search running in all elements of the field but it decreases the number of operations performed in the search.

Algorithm 18: Linearized polynomials for finding roots over $\mathbb{F}_{2^m}$.

```
Data: $p(x)$ as a univariate polynomial over $\mathbb{F}_{2^m}$ with degree $d$ and $m$ as the
      extension field degree.
Result: $R$ as a set of roots of $p(x)$.
for $k \leftarrow 0$ to $m - 1$ do
    $T[k] \leftarrow []$;
end
$Q \leftarrow []$;
for $j \leftarrow 0$ to $2^m - 1$ do
    $A[j] \leftarrow []$;
end
$R \leftarrow []$;
$dummy \leftarrow []$;
if $f_0 == 0$ then
    $R.append(0)$;
end
for $i \leftarrow 0$ to $\lceil (d-4)/5 \rceil$ do
    $\ell_i(x) \leftarrow 0$;
    for $j \leftarrow 0$ to $3$ do
        $\ell_i(x) \leftarrow \ell_i(x) + f_{5i+2^j} x^{2^j}$;
    end
    $Q[i] \leftarrow \ell_i(x)$;
end
for $k \leftarrow 0$ to $m - 1$ do
    for $i \leftarrow 0$ to $\lceil (d-4)/5 \rceil$ do
        $T[k][i] \leftarrow Q[\alpha^k]$;
    end
end
$A_i^0 \leftarrow []$;
for $i \leftarrow 0$ to $\lceil (d-4)/5 \rceil$ do
    $A_i^0 \leftarrow f_{5i}$;
end
$X \leftarrow$ generate$(m)$;
for $j \leftarrow 1$ to $2^m - 1$ do
    for $i \leftarrow 0$ to $\lceil (d-4)/5 \rceil$ do
        $temp \leftarrow A[j-1][i]$;
        $temp \leftarrow temp + T[\delta(X[j], X[j-1])][i]$;
        $A[j][i] \leftarrow temp$;
    end
end
for $j \leftarrow 1$ to $2^m - 1$ do
    $result \leftarrow 0$;
    for $i \leftarrow 0$ to $\lceil (d-4)/5 \rceil$ do
        $result = result + (X[j])^{5i} A[j][i]$;
    end
    $eval = result + f_3(X[j])^3$;
    if $eval == 0$ then
        $R.append(X[j])$;
    else
        $dummy.append(X[j])$;
    end
end
return $R$;
```

5.2.3 – Berlekamp Trace Algorithm – BTA. In [Ber70], Berlekamp presents an efficient algorithm to factor a polynomial, which can be used to find its roots. We call this algorithm *Berlekamp trace algorithm* since it works with a trace function defined as $\mathrm{Tr}(x) = x + x^2 + x^{2^2} + \cdots + x^{2^{m-1}}$. It is possible to change BTA for finding roots of a polynomial $p(x)$ using $\beta = \{\beta_1, \beta_2, \ldots, \beta_m\}$ as a standard basis of $\mathbb{F}_{2^m}$, and then computing the greatest common divisor between $p(x)$ and $\mathrm{Tr}(\beta_1 \cdot x)$. After that, it starts a recursion where BTA performs two recursive calls; one with the result of gcd algorithm and the other with the remainder of the division $p(x)/\gcd(p(x), \mathrm{Tr}(\beta_i \cdot x))$. The base case is when the degree of the input polynomial is constant. In this case, BTA returns the root, by getting the constant term of the polynomial. In summary, the BTA is a divide and conquer like algorithm since it splits the task of computing the roots of a polynomial $p(x)$ into computing the roots of two smaller polynomials. The description of the BTA algorithm is presented in Algorithm 19.

Algorithm 19: Berlekamp Trace Algorithm [Str12] – $\mathrm{BTA}(p(x), i)$-rf.

```
Data: $p(x)$ as a univariate polynomial over $\mathbb{F}_{2^m}$ and i.
Result: The set of roots of $p(x)$.
if $\deg(p(x)) \leqslant 1$ then
    return root of $p(x)$;
end
$p_0(x) \leftarrow \mathrm{gcd}(p(x), \mathrm{Tr}(\beta_i \cdot x))$;
$p_1(x) \leftarrow p(x)/p_0(x)$ ;
return $\mathrm{BTA}(p_0(x), i+1) \cup \mathrm{BTA}(p_1(x), i+1)$;
```

As we can see, a direct implementation of Algorithm 19 has non-constant execution time. The recursive behavior may leak information about the characteristics of roots in a side-channel attack. Additionally, in our experiments, we noted that the behavior of the gcd with the trace function may result in a polynomial with the same degree. Therefore, BTA will divide this input polynomial in a future call with a different basis. Consequently, there is no guarantee of a constant number of executions.

In order to avoid the nonconstant number of executions, here referred as $\mathrm{BTA\text{-}it}$, we propose an iterative implementation of Algorithm 19. In this way, our proposal iterates in a fixed number of iterations instead of calling itself until the base case. The main idea is not changed; we still divide the task of computing the roots of a polynomial $p(x)$ into two smaller instances. However, we change the approach of the division of the polynomial. Since we want to compute the same number of operations independent of the degree of the polynomial, we perform the gcd with a trace function for all elements in β, and choose a division that results in two new polynomials with almost the same degree.

This new approach allows us to define a fixed number of iterations for our version of BTA. Since we always divide into two small instances, we need $t-1$ iterations to split a polynomial of degree t into t polynomials of degree 1. Algorithm 20 presents this approach.

Algorithm 20 extracts a root of the polynomial when the variable $\mathrm{current}$ has a polynomial with degree equal to one. If this degree is greater than one, then the algorithm needs to continue dividing the polynomial until it finds a root. The algorithm does that by adding the polynomial in a stack and reusing this polynomial in a division.

Algorithm 20: Iterative Berlekamp Trace Algorithm – $\mathtt{BTA}(p(x))\text{-}\mathtt{it}$.

```
Data: $p(x)$ as an univariate polynomial over $\mathbb{F}_{2^m}$, $d = deg(p(x))$ as number of
      expected roots, $\beta$ as a standard basis of $\mathbb{F}_{2^m}$.
Result: The set of roots of $p(x)$.
$g \leftarrow \{p(x)\}$; // The set of polynomials to be computed
for $k \leftarrow 0$ to $d$ do
    $current = g.pop()$;
    for $j \leftarrow 1$ to $m$ do
        $candidates[j-1] \leftarrow gcd(current, Tr(\beta_j \cdot x))$;
    end
    Select $p_0 \in candidates$ such as $p_0$.degree $\simeq \frac{current.degree}{2}$;
    $p_1(x) \leftarrow current/p_0(x)$ ;
    if $p_0$.degree $== 1$ then
        $R.add$(root of $p_0$)
    end
    else
        $g.add(p_0)$;
    end
    if $p_1$.degree $== 1$ then
        $R.add$(root of $p_1$)
    end
    else
        $g.add(p_1)$;
    end
end
return $R$
```

The overall cost of Algorithm 20 is:

$$C_{BTA-it} = t(mC_{gcd} + C_{QuoRem}). \tag{5.14}$$

where C_{gcd} is the cost to compute the gcd of two polynomials, d be the largest degree of the two polynomials. In our implementation, the cost of C_{gcd} is:

$$C_{gcd} = d(C_{gf\_inv} + 3C_{gf\_mul}), \tag{5.15}$$

and C_{QuoRem} is the cost for computing the quotient and remainder between two polynomials. The cost for this computation is:

$$C_{QuoRem} = d(C_{gf\_inv} + (d+1)C_{gf\_mul} + C_{gf\_add}). \tag{5.16}$$

5.2.4 – Successive Resultant Algorithm. In [Pet14], the author presents an alternative method for finding roots in $\mathbb{F}_{p^m}$. Later on, the authors explain the method better in [DPP16]. The Successive Resultant Algorithm (SRA) relies on the fact that it is possible to find roots exploiting properties of an ordered set of rational mappings.

Given a polynomial p of degree d and a sequence of rational maps $K_1, \ldots, K_t$, the algorithm computes finite sequences of length $j \leqslant t+1$ obtained by successively transforming the roots of p by applying the rational maps. The algorithm can be explained as follows: Let $\{v_1, \dots, v_m\}$ be an arbitrary basis of $\mathbb{F}_{p^m}$ over $\mathbb{F}_p$, then it is possible to define $m+1$ functions $\ell_0, \ell_1, \dots, \ell_m$ from $\mathbb{F}_{p^m}$ to $\mathbb{F}_{p^m}$ such that

$$\begin{cases} \ell_0(z) = z \\ \ell_1(z) = \prod_{i \in \mathbb{F}_p} \ell_0(z - iv_1) \\ \ell_2(z) = \prod_{i \in \mathbb{F}_p} \ell_1(z - iv_2) \\ \dots \\ \ell_m(z) = \prod_{i \in \mathbb{F}_p} \ell_{m-1}(z - iv_1). \end{cases}$$

The functions ℓ_j are examples of linearized polynomials, as previously defined in Section 5.2.2.

Petit [Pet14] sets up the following polynomial system:

$$\begin{cases} f(x_1) = 0 \\ x_j^p = a_j x_j = x_{j+1} \quad j = 1, \dots, m-1 \\ x_m^p - a_m x_m = 0 \end{cases} \tag{5.17}$$

where the $a_i \in \mathbb{F}_{p^n}$. Any solution of this system provides us with a root of $p(x)$ by the first equation, and the m last equations together imply this root belongs to $\mathbb{F}_{p^m}$. From this system of equations, [Pet14] derives Theorem 5.3.

Theorem 5.3. Let $(x_1, x_2, \dots, x_m)$ be a solution of the equations in Equation 5.17. Then $x_1 \in \mathbb{F}_{p^m}$ is a solution of $p(x)$. Conversely, given a solution $x_1 \in \mathbb{F}_{p^m}$ of $p(x)$, we can reconstruct a solution of all equations in Equation 5.17 by setting $x_2 = x_1^p - a_1 x_1$, etc.

In [Pet14], the author presents an algorithm for solving the system in Equation 5.17 using resultants. The solutions of the system are the roots of polynomial $p(x)$. We implemented the method presented in [Pet14] using SAGE Math [Th19] due to the lack of libraries in C that work with multivariate polynomials over finite fields. It is worth remarking that this algorithm is almost constant-time and hence we just need to protect the branches presented on it. The countermeasure adopted was to add dummy operations, similar to Section 5.2.2.

5.3 — Comparison

In this section, we presented the results of the execution of each of the methods presented in Section 5.2. We used an Intel® Core(TM) i5-5300U CPU @ 2.30GHz. The code was compiled with GCC version 8.3.0 and the following compilation flags "*-O3 -g3 -Wall -march=native -mtune=native -fomit-frame-pointer -ffast-math*". We ran 500 times the code and got the average number of cycles. Table 5.1 shows the number of cycles of root finding methods without countermeasures, while Table 5.2 shows the number of cycles when there is a countermeasure. In both cases, we used $d \in \{55, 65, 100\}$ where d is the number of roots. We remark that the operations in the tables are over $\mathbb{F}_{2^{12}}$ and $\mathbb{F}_{2^{16}}$. We used two different finite fields for showing the generality of our implementations and the costs for a small field and a larger field.

Figure 5.1 shows the number of cycles for random polynomials with degree 55, 65 and 100 and all the operations are over $\mathbb{F}_{2^{16}}$ and we ran 600 times with random polynomials with $d \in \{55, 65, 100\}$. Figure 5.1a shows a time variation in the execution time of the exhaustive method, as expected, the average time was increased. Figure 5.1b shows a

Table 5.1: Number of cycles divided by 10^8 for each method of finding roots without countermeasures.

Nr. Roots	Field	Exhaustive Search	Linearized polynomials	BTA-rf	SRA
55	$\mathbb{F}_{2^{12}}$	10.152	11.116	9.801	23.016
	$\mathbb{F}_{2^{16}}$	30.336	31.100	37.169	23.335
65	$\mathbb{F}_{2^{12}}$	12.103	15.512	11.933	27.113
	$\mathbb{F}_{2^{16}}$	37.331	38.709	38.308	27.828
100	$\mathbb{F}_{2^{12}}$	18.994	18.316	17.322	35.552
	$\mathbb{F}_{2^{16}}$	48.561	63.792	57.707	37.359

Table 5.2: Number of cycles divided by 10^8 for each method of finding roots with countermeasures.

Nr. Roots	Field	Exhaustive Search	Linearized polynomials	BTA-it	SRA
55	$\mathbb{F}_{2^{12}}$	11.311	12.546	10.718	24.104
	$\mathbb{F}_{2^{16}}$	32.578	43.364	36.471	26.600
65	$\mathbb{F}_{2^{12}}$	13.941	17.590	16.774	28.558
	$\mathbb{F}_{2^{16}}$	39.161	40.394	39.081	29.296
100	$\mathbb{F}_{2^{12}}$	19.711	19.921	18.081	42.114
	$\mathbb{F}_{2^{16}}$	50.120	65.331	68.636	42.124

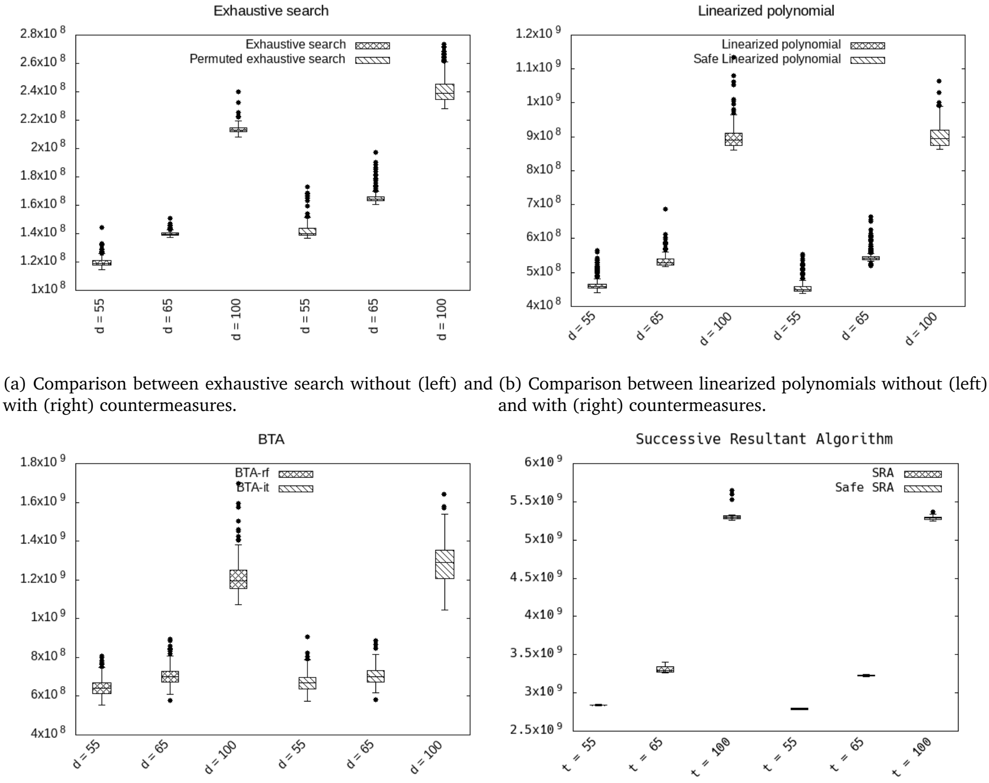
variation of time when we did not add the countermeasures, but when we add them, we see a more constant behavior. Figure 5.1c, it is possible to see a nonconstant behavior of BTA-rf. However, this is different for BTA-it, which shows a constant behavior.

The main focus of our proposal was to find alternatives to compute roots of ELP that have constant execution time. Figure 5.2 presents an overview between the original implementations and the implementations with countermeasures. It is possible that when a countermeasure is present on Linearized and on BTA approach, the number of cycles increases. However, the variance of time decreases. We remark that the "points" out of range can be ignored since we did not run the code under a separated environment, and as such it could be that some process in our environment influenced the result.

In our study, we demonstrated countermeasures that can be used to avoid remote timing attacks. In our empirical analysis, i.e, the results in Table 5.2, BTA-it shows an increase in the number of cycles but it shows a more constant behaviour and it is one of the safer choices. However, the exhaustive search with shuffling shows the smallest variation of time, which can be an alternative for usage. Still, the problem for this method is that if the field is large, then it becomes costly to shuffle and iterate through all elements.

5.3.1 – Open problems. We bring to the attention of the reader that we did not use any optimization in our implementations, i.e., we did not use vectorization or bit slicing techniques or any specific instructions such as Intel® IPP Cryptography for finite field arithmetic in our code. Therefore, these techniques and instructions can improve the finite fields operations and reduce the number of cycles of our implementations.

We remark that for achieving a safer implementation, one needs to improve the security analysis, by removing conditional memory access and protecting memory access of instructions. Moreover, one can analyze the security of the implementations, by considering



(a) Comparison between exhaustive search without (left) and with (right) countermeasures.

(b) Comparison between linearized polynomials without (left) and with (right) countermeasures.

(c) Comparison between BTA-rf (left) and BTA-it (right) executions.

(d) Comparison between SRA (left) and Safe SRA (right) executions.

Figure 5.1: Plots of measurements cycles for methods presented in Section 5.2. Our evaluation of SRA was made using a Python implementation and cycles measurement with C. In our tests, the drawback of calling a Python module from C has behavior bordering to constant.

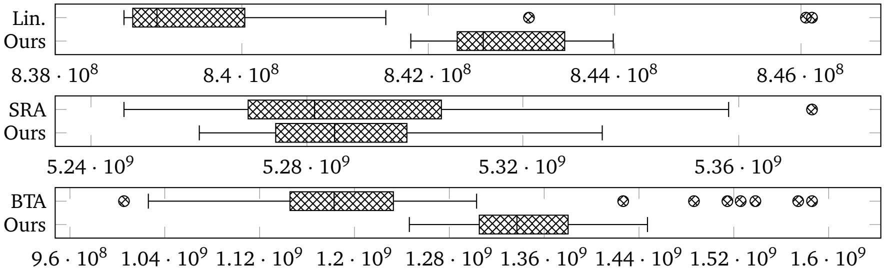


Figure 5.2: Comparison of original implementation and our proposal for Linearized, Successive resultant algorithm and Berlekamp trace algorithm with $d = 100$.

different attack scenarios and performing an in-depth analysis of hardware side-channel attacks.

Chapter 6

A Reaction Attack on LRPC Codes

Currently, there are two major trends to construct code-based schemes with small keys. The first one makes use of codes defined by very sparse parity-check matrices, such as LDPC and MDPC codes [BCG06, MTSB13], while the second one is based on *rank metric* codes [Gab85]. In this chapter, we focus on the latter, and in particular, we consider the case of LRPC codes, which are in a sense a point of contact between the two. In fact, LRPC stands for Low-Rank Parity-Check, and this class of codes is characterized by a "sparse" (in the rank metric sense) parity-check matrix, and therefore it can effectively be seen as a rank-metric equivalent of LDPC/MDPC codes. As we will see, and as it is often the case for rank-metric schemes, LRPC codes share many of the aspects of their Hamming metric counterpart, including vulnerabilities.[1]

In what follows, we describe McNie [KKG+18] – a first round candidate [NIS17] to the NIST PQ crypto standardization process[2]. We will use McNie to showcase our reaction attack in Section 6.4.

McNie follows a "hybrid" framework using both McEliece and Niederreiter in the encryption process. The scheme employs quasi-cyclic codes with low-weight parity-check matrices of the form $(\mathbf{H}_1\ \mathbf{H}_2\ \mathbf{H}_3)$ and $\left(\begin{smallmatrix}\mathbf{H}_1 & \mathbf{H}_2 & \mathbf{H}_3 & \mathbf{H}_4\\ \mathbf{H}_5 & \mathbf{H}_6 & \mathbf{H}_7 & \mathbf{H}_8\end{smallmatrix}\right)$ where $\mathbf{H}_i$ are circulant matrices. The authors refer to these codes as 3- and 4-Quasi-Cyclic codes. The key generation, encryption and decryption of McNie are summarized in Figure 6.1.

Remark. According to the protocol specifications [GKK+17], in the general description of the scheme, the authors suggest the possibility to further use a permutation matrix $\mathbf{P}$ to form $\mathbf{F}$ as $\mathbf{F} = \mathbf{G}'\mathbf{P}^{-1}\mathbf{H}^\top\mathbf{S}$. However, in the actual proposal, this matrix is set to the identity matrix, so it is never used. Therefore, we do not see a reason to use it and burden the description.

There exist some variations of LRPC codes that can be used in the same framework for cryptosystems. For example, it is possible to use double circulant LRPC (DC-LRPC) where the parity-check matrix $\mathbf{H}$ is a double-circulant matrix of rank d, i.e., a concatenation of two cyclic matrices, as shown in [GRSZ14]. Another variation used is Quasi-Cyclic

[1] This chapter is based on a paper accepted at Latincrypt 2019 and it is joint work with Edoardo Persichetti, Simona Samardjiska and Paolo Santini [SSPB19].

[2] https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization

1. **Key generation:** Choose a random 3 or 4 generator QC LRPC code over $\mathbb{F}_{q^m}$ of low rank d, $(n-k)\times n$ parity check matrix H over $\mathbb{F}_{q^m}$ and generator matrix $\mathbf{G}$ over $\mathbb{F}_{q^m}$. Further, choose a random invertible $(n-k)\times(n-k)$ matrix $\mathbf{S}$ over $\mathbb{F}_{q^m}$ and a random $l \times n$ matrix $\mathbf{G}'$ over $\mathbb{F}_{q^m}$.
   Secret Key: The low rank matrix H, and the masking matrix $\mathbf{S}$.
   Public Key: The matrices $\mathbf{F} = \mathbf{G}'H^\top\mathbf{S}$ and $\mathbf{G}'$.
2. **Encryption:** To encrypt a message $\mathbf{m} \in \mathbb{F}_{q^m}$, generate a random $\boldsymbol{e} \in \mathbb{F}_{q^m}^n$ of rank r. Compute $\mathbf{c}_1 = \mathbf{m}\mathbf{G}' + \boldsymbol{e}$ and $\mathbf{c}_2 = \mathbf{m}\mathbf{F}$. The ciphertext is $(\mathbf{c}_1, \mathbf{c}_2)$.
3. **Decryption:** Compute syndrome $\mathbf{s}' = \mathbf{c}_1 H^\top - \mathbf{c}_2\mathbf{S}^{-1} = \boldsymbol{e}H^\top$. Recover the error vector $\boldsymbol{e}$ by decoding the LRPC code, then compute $\mathbf{m}\mathbf{G}' = \mathbf{c}_1 - \boldsymbol{e}$ and obtain $\mathbf{m}$ by solving the obtained system.

Figure 6.1: The McNie cryptosystem [GKK+17, KKG+18].

LRPC codes where the parity-check matrix H is a quasi-cyclic matrix of low rank d as in [KKG+18]. In all those variations as the one presented in Figure 2.1, the secret matrix H has *low rank*.

6.1 — A Reaction Attack

We are now ready to describe the details of our attack. The main idea is to exploit decoding failures caused by the syndrome $\mathbf{s}$ not generating the whole space $\langle FE \rangle$. Thus, for ease of exposition, in this section we will assume that this is the case. Later we will show that the influence of other types of decoding failures to the success of our attack is negligible, thus justifying the current assumption.

Suppose that an adversary $\mathcal{A}$ interacts with a decryption oracle $\mathcal{D}$ of an LRPC cryptosystem. He continuously sends encrypted messages to $\mathcal{D}$ and waits for the reaction from the oracle. If $\mathcal{D}$ returns failure, $\mathcal{A}$ records the error $\mathbf{e}$ that he used in the encryption of the message. $\mathcal{A}$ collects a total of t error vectors, where t is chosen appropriately. We will discuss this choice later in this section.

Let $\mathbf{e}$ be an error vector that $\mathcal{A}$ collected during his interaction with $\mathcal{D}$. We now show how to use this information to recover the secret matrix $\mathbf{H}$.

Recall from Section 2.2.7, Equation 2.26, that we can express the syndrome equation over the base field as $\mathbf{s}' = \mathbf{A}_H \boldsymbol{e}'^\top$, where $\mathbf{s}'$ contains the coefficients of the syndrome in the basis $\{F_iE_j\}_{\substack{1\leqslant i\leqslant d\\ 1\leqslant j\leqslant r}}$. Alternatively, directly from equation 2.24, the syndrome equation can be written in a matrix form: $\mathbf{s}$ can be written as the product between the vector of basis elements $(F_1E_1, F_1E_2, \ldots, F_dE_r)$ and a matrix $\bar{\mathbf{A}}_{\mathbf{H},\mathbf{e}} \in \mathbb{F}_q^{rd\times(n-k)}$:

$$\mathbf{s} = (F_1E_1, F_1E_2, \ldots, F_dE_r)\cdot \bar{\mathbf{A}}_{\mathbf{H},\mathbf{e}}. \tag{6.1}$$

The key observation in our attack is that a decoding failure occurs when the matrix $\bar{\mathbf{A}}_{\mathbf{H},\mathbf{e}}$ is not of full rank – in other words, the left kernel of the matrix $\bar{\mathbf{A}}_{\mathbf{H},\mathbf{e}}$ is non-trivial. This means that there must exist (at least) one nonzero vector $\mathbf{v}_\mathbf{e} \in \mathbb{F}_q^{rd}$, $\mathbf{v}_\mathbf{e} \neq 0_{1\times rd}$, such that

$$\mathbf{v}_\mathbf{e} \cdot \bar{\mathbf{A}}_{\mathbf{H},\mathbf{e}} = 0_{1\times n-k}. \tag{6.2}$$

Now consider our attack scenario. The adversary $\mathcal{A}$ knows the error $\mathbf{e}$ that caused the matrix $\bar{\mathbf{A}}_{\mathbf{H},\mathbf{e}}$ to be of non-full rank. He, however, does not know the coefficients $\mathbf{h} = \{h_{i,j,l}\}_{\substack{1\leqslant l\leqslant d\\ 1\leqslant i\leqslant n-k\\ 1\leqslant j\leqslant n}}$ of the matrix $\mathbf{H}$, and therefore he does not know the kernel vector $\mathbf{v}_\mathbf{e}$. Setting $\bar{\mathbf{A}}_\mathbf{e}(\mathbf{h}) = \bar{\mathbf{A}}_{\mathbf{H},\mathbf{e}}$ to emphasize the unknown coefficients $\mathbf{h}$, we can rewrite Equation 6.2 as

$$\mathbf{v}_\mathbf{e} \cdot \bar{\mathbf{A}}_\mathbf{e}(\mathbf{h}) = 0_{1\times n-k}. \tag{6.3}$$

The main step of our attack now boils down to finding the solutions to Equation 6.3 in the unknown coefficients $\mathbf{h}$ of $\mathbf{H}$ and the unknown kernel vector $\mathbf{v}_\mathbf{e}$.

Observe that we can actually use several errors $\mathbf{e}_1, \dots, \mathbf{e}_t$ to form equations similar to Equation 6.3. In these equations for each error $\mathbf{e}_i$ we introduce a new appropriate kernel element $\mathbf{v}_{\mathbf{e}_i}$. However, they all share the same unknown coefficients of the matrix $\mathbf{H}$. Thus, we can form the following system:

$$\begin{cases} \mathbf{v}_{\mathbf{e}_1} \cdot \bar{\mathbf{A}}_{\mathbf{e}_1}(\mathbf{h}) = 0_{1\times n-k} \\ \mathbf{v}_{\mathbf{e}_2} \cdot \bar{\mathbf{A}}_{\mathbf{e}_2}(\mathbf{h}) = 0_{1\times n-k} \\ \dots \\ \mathbf{v}_{\mathbf{e}_t} \cdot \bar{\mathbf{A}}_{\mathbf{e}_t}(\mathbf{h}) = 0_{1\times n-k} \end{cases} \tag{6.4}$$

The right value for t depends on several factors: the method of solving, the parameters of the system, but most notably the nature of the system. In principle, t should be big enough such that solving the system unambiguously gives the coefficients of $\mathbf{H}$. We will discuss the choice of t in the next section.

Suppose that t is chosen appropriately. Observe that system 6.4 is a system of bilinear equations reminiscent of the equations obtained in the MinRank problem [BFS99]. We can thus try solve this system using similar strategies as in at least three different methods for solving MinRank – the Kernel method [GC00], Kipnis-Shamir method [KS99] and the minors method [Cou01]. The main differences are that first, there are several polynomial matrices whose non-trivial kernel needs to be found and second, all of these matrices are polynomial matrices in the same variables. At first sight, this situation bears similarities to Simultaneous MinRank [BFP13, FGP+15] which is commonly encountered in $\mathcal{MQ}$ cryptography based on multivariate systems. However, since the different errors $\mathbf{e}_1, \mathbf{e}_2, \dots, \mathbf{e}_t$ produce different matrices $\bar{\mathbf{A}}_{\mathbf{e}_1}(\mathbf{h}), \bar{\mathbf{A}}_{\mathbf{e}_2}(\mathbf{h}), \dots, \bar{\mathbf{A}}_{\mathbf{e}_t}(\mathbf{h})$, it is not clear how to use the common techniques that significantly speed up the attack on those $\mathcal{MQ}$ cryptosystems.

In the next subsection, we will describe in detail a Kernel-method–like approach to solving system 6.4. A straightforward application of the other algebraic methods results in a significantly larger complexity. Therefore a deeper insight into the properties of system 6.4 is necessary in order to apply these efficiently.

6.1.1 – Solving System 6.4 by Kernel Guessing. Suppose that the adversary $\mathcal{A}$ has collected t error vectors $\mathbf{e}_1, \mathbf{e}_2, \dots, \mathbf{e}_t$ that cause decryption failures. As before, we assume that the corresponding matrices $\bar{\mathbf{A}}_{\mathbf{e}_1}(\mathbf{h}), \bar{\mathbf{A}}_{\mathbf{e}_2}(\mathbf{h}), \dots, \bar{\mathbf{A}}_{\mathbf{e}_t}(\mathbf{h})$ are singular. This means that sizes of the kernels of these matrices are at least 1. The adversary now tries to guess the vectors $\mathbf{v}_{\mathbf{e}_i}$, such that $\mathbf{v}_{\mathbf{e}_i}$ belongs to the kernel of $\bar{\mathbf{A}}_{\mathbf{e}_i}(\mathbf{h})$ for each $i \in [1; t]$. If all vectors $\mathbf{v}_{\mathbf{e}_i}$ are correctly guessed, what remains is to solve the obtained linear system in the unknown coefficients $\mathbf{h}$. In general, in order to obtain a unique solution we need to collect at least the same number of equations as variables. For each $\mathbf{v}_{\mathbf{e}_i}$ we can form $n-k$

equations, so we need $t(n-k)$ to be at least the number of variables. For a random LRPC code of rank d, the number of unknown coefficients of the matrix $\mathbf{H}$ is $n(n-k)d$. Hence we need:

$$t \geqslant nd.$$

If, in addition, the code is quasi-cyclic and its parity-check matrix is made of circulant matrices of size p, the number of unknown coefficients is $n(n-k)d/p$. In this case

$$t \geqslant \frac{nd}{p}. \tag{6.5}$$

Let us denote the probability of correctly guessing a kernel vector corresponding to an error $\mathbf{e}_i$ by $P_{\mathbf{e}_i}$. This probability clearly depends on the dimension of the kernel of $\bar{\mathbf{A}}_{\mathbf{e}_i}(\mathbf{h})$. Let us denote this dimension as $K_{\mathbf{e}_i} \geqslant 1$: then, we know that $|\mathrm{Ker}(\bar{\mathbf{A}}_{\mathbf{e}_i}(\mathbf{h}))| = q^{K_{\mathbf{e}_i}}$. Then,

$$P_{\mathbf{e}_i} = \frac{q^{K_{\mathbf{e}_i}}}{q^{rd}} = q^{-(rd-K_{\mathbf{e}_i})}. \tag{6.6}$$

Clearly, a larger kernel of some of the matrices would make the attack faster. However, there is no way to detect whether a matrix $\bar{\mathbf{A}}_{\mathbf{e}_i}(\mathbf{h})$ associated to an error $\mathbf{e}_i$ would have a larger kernel. Therefore we must assume the worst case, i.e. a kernel of dimension 1. It remains open whether it is possible to devise a strategy to generate error vectors that induce matrices of larger kernels.

Let us denote the probability of all vectors $\mathbf{v}_{\mathbf{e}_i}$ being correctly guessed by P_t. Then

$$P_t = \prod_{i=1}^{t} P_{\mathbf{e}_i} \geqslant q^{-(rd-1)t}. \tag{6.7}$$

After the kernel vectors have been guessed, system 6.4 becomes an over-determined linear system over $\mathbb{F}_q$. Solving it gives the coefficients of $\mathbf{H}$. However, the basis F is still unknown. Luckily, knowing the coefficients of $\mathbf{H}$ turns out to be enough to find the basis F. It is possible to obtain the basis F from sufficiently many message-ciphertext pairs and the syndrome equation. A high level description of the attack is given by Algorithm 21.

In Algorithm 21, through the procedure CollectErrors, the adversary interacts with the decryption oracle $\mathcal{D}$, by sending it encrypted messages and waiting for decryption failures. Each time there is a failure, the adversary saves the error it used, until enough errors that cause decryption failures are collected. The procedure CollectMEC adds an extra cost to the adversary of one encryption.

Once the errors have been obtained the main part of the attack can begin. Note that the collection of the errors can be done only once, and we need only a handful of errors unlike in the Hamming metric (see Section 6.1.2 for a more detailed discussion).

Next, the function SolveH denotes the procedure for solving system 6.4 for some guessed kernel elements corresponding to the obtained errors. If system 6.4 has a solution, then SolveH will return this solution, otherwise it will return $\perp$. This solution is then used in SolveF together with ℓ valid triplets $(\mathbf{m}_i, \mathbf{e}_i, \mathbf{c}_i)$ of messages, errors and ciphertexts generated in the procedure CollectMEC. SolveF is a procedure whose main goal is to find the basis F. However depending on the scheme, there might be other parts of the secret key sk that can be found in this procedure. The value of ℓ is also dependent on the scheme. We present an instantiation of SolveF for McNie [GKK+17] in Section 6.4.

Algorithm 21: Reaction attack on LRPC codes.

```
Data: $d, t, \ell \in \mathbb{Z}$
Result: Matrix $\mathbf{H}$ of rank $d$
$\mathbf{e}_1, \mathbf{e}_2, \dots, \mathbf{e}_t \leftarrow \mathsf{CollectErrors}(\mathsf{pk}, \mathcal{D}(\mathsf{sk}))$ ;    // Collect errors from decryption failures
repeat
    $\mathbf{v}_{\mathbf{e}_1}, \mathbf{v}_{\mathbf{e}_2}, \dots, \mathbf{v}_{\mathbf{e}_t} \leftarrow_R \mathbb{F}_q^{rd}$;    // Guess kernel vectors
    $\mathbf{h} \leftarrow \mathsf{SolveH}(\mathbf{v}_{\mathbf{e}_1}, \mathbf{v}_{\mathbf{e}_2}, \dots, \mathbf{v}_{\mathbf{e}_t}, \mathbf{e}_1, \mathbf{e}_2, \dots, \mathbf{e}_t)$ ;    // Solve system 6.4
    if $\mathbf{h} \neq \perp$ then
        $\{(\mathbf{m}_i, \mathbf{e}_i, \mathbf{c}_i)\}_{i=1}^{\ell} \leftarrow \mathsf{CollectMEC}(\mathsf{pk})$ ;    // Collect messages, errors, ciphertexts
        $F, \mathsf{success} \leftarrow \mathsf{SolveF}(\mathbf{h}, \{(\mathbf{m}_i, \mathbf{e}_i, \mathbf{c}_i)\}_{i=1}^{\ell})$ ;    // Find basis F
    else
        $\mathsf{success} \leftarrow \perp$;
    end
until success;
$\mathbf{H} \leftarrow \mathsf{Reconstruct}(\mathbf{h}, F)$;    // Reconstruct the matrix H
return $\mathbf{H}$;
```

We are now ready to state the total complexity of our attack. It is

$$\begin{aligned}\mathsf{Cost}(\mathsf{React}) = & P_3^{-1}(\mathsf{Cost}(\mathsf{Enc} \wedge \mathsf{Dec})) + \\ & + P_t^{-1}(\mathsf{Cost}(\mathsf{SolveH}) + \ell\mathsf{Cost}(\mathsf{Enc}) + \mathsf{Cost}(\mathsf{SolveF}))\end{aligned} \tag{6.8}$$

where $P_3 = \frac{1}{q^{n-k+1-rd}}$ is the failure rate of the scheme (see Section 2.2.7.1), $P_t = q^{-(rd-1)t}$ for a rank-d LRPC code and errors of rank r. In the case of random LRPC codes $t = nd$ and $\mathsf{Cost}(\mathsf{SolveH}) = n^3(n-k)^3d^3$. When the code is quasi-cyclic and uses circulant matrices of size p, $t = \frac{nd}{p}$ and $\mathsf{Cost}(\mathsf{SolveH}) = \frac{n^3(n-k)^3d^3}{p^3}$. As said earlier, $\mathsf{Cost}(\mathsf{SolveF})$ depends on the scheme, but usually, $\mathsf{Cost}(\mathsf{SolveF}) < \mathsf{Cost}(\mathsf{SolveH})$. See Section 6.4 for more details about this.

6.1.2 – Analogies and Differences with the Hamming Metric. The cryptanalysis procedure we have described in this section resembles the one proposed by Guo, Johansson and Stankovski in [GJS16], tailored for the McEliece cryptosystems using QC Moderate-Density Parity-Check codes [MTSB13], decoded in the Hamming metric. Essentially, these codes are a special case of Low-Density Parity-Check codes [Gal63], i.e., codes which are described by a parity-check matrix that contains a low number of nonzero entries. These codes admit efficient decoding algorithms, like the Bit Flipping (BF) decoder or some of its variants, which are all based on the sparsity of the parity-check matrix. In any case, the most popular decoders have a common structure, that is, they consist in iterative procedures, in which at each iteration the error vector is partially guessed and the syndrome gets coherently updated. This procedure goes on until a null syndrome is obtained (decoding success), or a maximum number of iterations is reached (decoding failure).

As originally observed in [GJS16], the decoding failure probability somehow depends on geometrical relations between the error vector and the secret parity-check matrix. Then, reaction attacks can be mounted, by means of statistical tests on the decoding outcomes of a large number of decryption queries. In particular, these tests are used to guess the number of overlapping ones between columns in the secret key [SBCB19]; clearly, in order to achieve statistical reliability, the number of observed decryption instances (i.e., the number of queries) needs to be sufficiently large. For instance, we can consider the empirical results for the parameters that were broken in [GJS16]: the authors used, in all successful attacks, more than 10^8 decryption queries. Considering a decoding failure probability approximately equal to 10^{-4}, this leads to a number of observed events of decoding failures in the order of 10^4.

There are clear differences between reaction attacks in the Hamming metric and the one we propose in this work. First of all, in the rank metric case, no statistical test is needed: events of decoding failures are due (with overwhelming probability) to some rank deficiency in the syndrome, and this fact is used to establish algebraic relations like those in Equation 6.3. This difference is emphasized by the fact that the number of failure events that an adversary needs to collect is significantly lower than the one that is needed for the Hamming metric case.

Additionally, in the Hamming metric case, the feasibility of reaction attacks is somehow related to the chosen decoder and to its setting [NJS19], in the sense that modifications in the decoding procedure and/or slight variations in its setting might lead to significant differences in the attack outcome. From this difference arises a question on the existence of alternative LRPC decoding techniques, and on their eventual effect on reaction attacks.

Another crucial difference is represented by the fact that, for Low-Density Parity-Check codes, only few parity-check matrices can be used to efficiently perform decoding on a given corrupted codeword. Indeed, for a given parity-check matrix $\mathbf{H}$, each matrix $\mathbf{H}' = \mathbf{W}\mathbf{H}$, with $\mathbf{W}$ being non-singular, is again a valid parity-check matrix, but $\mathbf{W}$ preserves the density only if it is a permutation matrix. If $\mathbf{W}$ is not a permutation matrix, in fact, rows of $\mathbf{H}'$ correspond to linear combinations of rows of $\mathbf{H}$: thus, their density is, with overwhelming probability, larger than that of $\mathbf{H}$. This means that only the actual $\mathbf{H}$, or a row-permuted version of it, guarantees efficient decoding of intercepted ciphertexts. Then, when mounting a reaction attack, the adversary's goal is that of reconstructing exactly one of these matrices. In the rank metric case the number of parity-check matrices that allow for efficient decoding techniques is significantly larger – we show in the next section that any matrix of the form $\mathbf{W}\mathbf{H}$ can be used to efficiently decode. In such a case, we speak of *equivalent keys*: this fact, as we describe in the next section, allows for significant reductions in the attack complexity.

6.2 — Equivalent Keys in LRPC Cryptosystems

In the previous section we described the basic attack that makes use of decryption failures. Now, we dig a little deeper, and show that due to existence of particular equivalent keys, it is possible to speed up the attack by an exponential factor.

We start with a well known property of weight preservation in the rank metric. For completeness we include a proof that will be useful later on.

Proposition 6.1. Let $\mathbf{b} \in \mathbb{F}_{q^m}^n$, and let $\mathbf{W} \in GL_n(\mathbb{F}_q)$. Then:

$$\mathrm{wt}(\mathbf{b}) = \mathrm{wt}(\mathbf{b} \cdot \mathbf{W}).$$

In other words, weight is preserved under multiplication by non-singular matrices over $\mathbb{F}_q$. Recall that in Hamming metric, weight is preserved under multiplication by permutation matrices.

Proof. Let $\mathrm{wt}(\mathbf{b}) = d$. This means that $\mathbf{b}$ can be represented as $\mathbf{b} = \mathbf{F} \cdot \bar{\mathbf{B}}$, where $\mathbf{F} = (F_1, \dots, F_d)$, and $\langle F_1, \dots, F_d \rangle$ is a basis of some d-dimensional subspace of $\mathbb{F}_{q^m}^n$, and $\bar{\mathbf{B}} \in \mathcal{M}_{d,n}(\mathbb{F}_q)$ is the (full rank) matrix representation of $\mathbf{b}$. Now

$$\mathbf{b} \cdot \mathbf{W} = \mathbf{F} \cdot \bar{\mathbf{B}} \cdot \mathbf{W} = \mathbf{F} \cdot (\bar{\mathbf{B}} \cdot \mathbf{W}).$$

Since $\mathbf{W}$ is invertible, $\bar{\mathbf{B}} \cdot \mathbf{W}$ has the same rank as $\bar{\mathbf{B}}$, i.e., $\mathrm{wt}(\mathbf{b} \cdot \mathbf{W}) = d$. □

As a direct consequence, we have the following:

Proposition 6.2. Let $\mathbf{H} \in \mathcal{M}_{n-k,n}(\mathbb{F}_{q^m})$ be the parity check matrix of an LRPC code $\mathcal{C}$ of rank d. Let $\mathbf{W} \in GL_{n-k}(\mathbb{F}_q)$ be arbitrary. Then $\mathbf{WH}$ is a parity check matrix for the same code $\mathcal{C}$ hence the same rank d.

Proof. Follows directly from the previous proposition, by considering the columns of $\mathbf{H}$ as vectors of weight d. □

Definition 6.3. Let $P = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ be an LRPC cryptosystem with a secret key $\mathsf{sk} = (\mathbf{H}, \cdot)$. We say that P has an equivalent key $\mathsf{sk}' = (\mathbf{H}', \cdot')$, if $\mathsf{sk}' \neq \mathsf{sk}$ and sk' can be used as a secret key for P with the same efficiency as sk. In particular, $\mathbf{H}'$ is of the same rank as $\mathbf{H}$ and can be used in the decoding procedure with the same efficiency as $\mathbf{H}$.

With some abuse of this definition, we will also say that $\mathbf{H}'$ is an equivalent key of $\mathbf{H}$.

As a direct consequence of Proposition 6.2, we have:

Corollary 6.4. Let $P = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ be an LRPC cryptosystem with a secret key $\mathsf{sk} = (\mathbf{H}, \cdot)$ where $\mathbf{H} \in \mathcal{M}_{n-k,n}(\mathbb{F}_{q^m})$. Let $\mathbf{W} \in GL_{n-k}(\mathbb{F}_q)$ be arbitrary. Then, $\mathsf{sk}' = (\mathbf{WH}, \cdot)$ is an equivalent key for P.

A particular equivalent key is of our interest; later we present a key recovery attack that recovers exactly such a key.

Let $\mathbf{H} \in \mathcal{M}_{n-k,n}(\mathbb{F}_{q^m})$ be the parity check matrix of an LRPC code $\mathcal{C}$ of rank d. We rewrite $\mathbf{H}$ as:

$$\mathbf{H} = \begin{pmatrix} \sum_{i=1}^{d} h_{1,1,i}F_i & \sum_{i=1}^{d} h_{1,2,i}F_i & \cdots & \sum_{i=1}^{d} h_{1,n,i}F_i \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{i=1}^{d} h_{n-k,1,i}F_i & \sum_{i=1}^{d} h_{n-k,2,i}F_i & \cdots & \sum_{i=1}^{d} h_{n-k,n,i}F_i \end{pmatrix} =$$

$$= \sum_{i=1}^{d} \left(\begin{pmatrix} h_{1,1,i} & h_{1,2,i} & \cdots & h_{1,n,i} \\ \vdots & \vdots & \ddots & \vdots \\ h_{n-k,1,i} & h_{n-k,2,i} & \cdots & h_{n-k,n,i} \end{pmatrix} \right) F_i$$

i.e as

$$\sum_{i=1}^{d} \hat{\mathbf{H}}_i \cdot F_i = \sum_{i=1}^{d} (\hat{\mathbf{H}}_{i1} | \hat{\mathbf{H}}_{i2}) \cdot F_i \tag{6.9}$$

where $\hat{\mathbf{H}}_i$ is the matrix of coefficients corresponding to the basis element F_i and $\hat{\mathbf{H}}_{i1}$ are $(n-k) \times (n-k)$ matrices and $\hat{\mathbf{H}}_{i2}$ are $(n-k) \times k$ matrices.

Assume: $\hat{\mathbf{H}}_1 = (\hat{\mathbf{H}}_{11} | \hat{\mathbf{H}}_{12})$ where $\hat{\mathbf{H}}_{11} \in GL_{n-k}(\mathbb{F}_q)$, see Section 6.3.1.2 for more details about the existence of invertible matrix. Then $\mathbf{H}' = \hat{\mathbf{H}}_{11}^{-1} \cdot \mathbf{H}$ is an equivalent key and can be written as:

$$\mathbf{H}' = (\mathbf{I}_{n-k} | \hat{\mathbf{H}}'_{12}) \cdot F_1 + \sum_{i=2}^{d} (\hat{\mathbf{H}}'_{i1} | \hat{\mathbf{H}}'_{i2}) \cdot F_i \tag{6.10}$$

where $\hat{\mathbf{H}}'_{i1} = \hat{\mathbf{H}}_{11}^{-1} \cdot \hat{\mathbf{H}}_{i1}$ and $\hat{\mathbf{H}}'_{i2} = \hat{\mathbf{H}}_{11}^{-1} \cdot \hat{\mathbf{H}}_{i2}$.

A crucial observation to make is that in the general case, the equivalent key $\mathbf{H}'$ is determined by $n(n-k)d - (n-k)^2$ coefficients, as opposed to $n(n-k)d$ coefficients for $\mathbf{H}$. This observation can be used to reduce the size of the private key by storing $\mathbf{H}'$ instead of $\mathbf{H}$. It can also be used to speed up our reaction attack if we recover $\mathbf{H}'$ instead of $\mathbf{H}$ because now we have fewer unknown coefficients, i.e., fewer variables in system 6.4. We need however to show that the collected error vectors that correspond to a secret $\mathbf{H}$ are also valid for the equivalent keys.

Proposition 6.5. For an arbitrary LRPC code $\mathcal{C}$ of length n, dimension k and rank d over $\mathbb{F}_{q^m}$, with parity check matrix $\mathbf{H}$, if an error vector $\mathbf{e}$ causes a decoding failure, then the same error vector causes decoding failure for any equivalent $\mathbf{WH}$ where $\mathbf{W} \in GL_{n-k}(\mathbb{F}_q)$.

Proof. Recall that the error vector $\mathbf{e}$ in system 6.4 causes the syndrome to be of non-maximal rank. By multiplying the syndrome equation by $\mathbf{W} \in GL_{n-k}(\mathbb{F}_q)$ we obtain $(\mathbf{WH}) \cdot \mathbf{e}_i^\top = \mathbf{W} \cdot \mathbf{s}^\top$, which from Proposition 6.1 means that the same error vectors cause syndromes of non-maximal rank for the matrix $\mathbf{WH}$. From Proposition 6.2 we know that $\mathbf{WH}$ is an equivalent key.

□

6.2.0.1 – Equivalent Keys for Quasi-cyclic codes. Note that if $\mathbf{H}$ is quasi-cyclic, then so are the matrices $\hat{\mathbf{H}}_i$ in Equation 6.9. Then $\mathbf{H}' = \hat{\mathbf{H}}_{11}^{-1} \cdot \mathbf{H}$ is an equivalent key of the form 6.10. This key $\mathbf{H}'$ is determined by $\frac{n(n-k)d-(n-k)^2}{p}$ coefficients, as opposed to $\frac{n(n-k)d}{p}$ coefficients for $\mathbf{H}$.

6.3 — Equivalent Key Attack on Quasi-Cyclic H

In the previous section we showed that under some plausible conditions, there exists an equivalent key that is determined by fewer coefficients than the original one. Therefore it makes sense to look for this key in our attack. In the case of QC codes, we need $\frac{(n-k)^2}{p}$ fewer variables, so the number of kernel elements that we need to guess instead of 6.5 becomes

$$t \geqslant \frac{nd-(n-k)}{p}. \tag{6.11}$$

The gain in the probability compared to 6.7 is a factor of $q^{(rd-1)\frac{n-k}{p}}$, so looking for an equivalent key accelerates the attack by an exponential factor.

6.3.1 – Probability of Success. The success of the attack described above depends on two conditions being satisfied. First, recall that in Section 6.1 we assumed that all collected errors are a result of the syndrome not being of full rank. The results from Section 2.2.7.1 show that this is not always the case. However, as we will show shortly, this happens with significant probability, so we can conclude that the feasibility of our attack is not affected by these other types of decryption failures.

There is however one more place where the attack may fail: If we want to recover the good equivalent key from Section 6.2, the success of the attack further depends on the probability that such an equivalent key exists. As we will see later, this probability is also big, so it is safe to assume that an appropriate equivalent key exists.

6.3.1.1 – Syndrome of non-full rank. We say that an observed decoding failure event is *useful* if it is due to $\dim(S) < rd$. This happens with probability

$$\rho = \frac{P_3}{P_1+P_2+P_3} = \frac{1}{1+\frac{P_1+P_2}{P_3}}, \tag{6.12}$$

where the above probabilities depend on the system parameters and have been defined in Section 2.2.7.1. Since we typically have $P_1, P_2 \ll P_3$, we commonly have $\rho \approx 1$. Now the the attack will be successful only if all of the t collected errors are useful, i.e.,

$$Pr_s = \rho^t. \tag{6.13}$$

6.3.1.2 – Existence of good equivalent keys. The attack presented in Section 6.3 requires that a good equivalent key exists. Recall that in Equation 6.14, we assumed that $\hat{\mathbf{H}}_{11}$ is invertible matrix of size $n-k$. Actually, note that our attack does not make a difference between the matrices $\hat{\mathbf{H}}_{l1}$, for $l \in [1;d]$, so it is enough that at least one of these is an invertible. In other words, our attack will find an equivalent key of the form:

$$\mathbf{H}' = (\mathbf{I}_{n-k}|\hat{\mathbf{H}}'_{l2}) \cdot F_l + \sum_{\substack{i=1 \\ i\neq l}}^{d} (\hat{\mathbf{H}}'_{i1}|\hat{\mathbf{H}}'_{i2}) \cdot F_i \tag{6.14}$$

for some $l \in [1;d]$.

Since we are dealing with quasi-cyclic codes, the matrices $\hat{\mathbf{H}}_{l1}$, for $l \in [1;d]$ are block matrices of circulant matrices (block-circulant). Each of these circulant matrices can be uniquely represented by a polynomial. Now considering the determinant of the block-circulant matrix as a polynomial and assuming it behaves as a random polynomial, we can use the result from Proposition 2.26. Hence, the probability that the block-circulant matrix is invertible is given by:

$$\mathrm{Pr}_c = \frac{\prod_{i=1}^{\tau}(q^{d_i\alpha_i} - q^{d_i(\alpha_i-1)})}{q^{n-k}} \tag{6.15}$$

where $x^{n-k} - 1 = p_1^{\alpha_1}(x) \cdot \dots \cdot p_\tau^{\alpha_\tau}(x)$ is the factorization over $\mathbb{F}_q$ and $d_i = \deg(p_i)$.

Now the probability that at least one of the matrices $\hat{\mathbf{H}}_{l1}$, for $l \in [1; d]$ is invertible is

$$\mathrm{Pr}_{ek} = 1 - (1 - \mathrm{Pr}_c)^d. \tag{6.16}$$

Equation 6.16 gives the probability that an equivalent key of the special form exists.

Remark. We should emphasize that forcing the matrices $\hat{\mathbf{H}}_{l1}$, for $l \in [1; d]$ to be singular in the design of the scheme does not help hinder our attack. It only requires a small modification on the equivalent key. The rest of the attack is essentially the same.

6.3.2 – A Quantum-Enhanced Attack. Since cryptosystems based on LRPC codes are considered post-quantum, it makes sense to estimate their security against quantum-enhanced attack, using the full power of quantum computers. A second look at our attack immediately shows a possibility for a quantum speed-up using Grover's algorithm [Gro96]. Recall that Grover's algorithm searches for an item in an unsorted database satisfying a given condition. In our attack, a huge part consists of searching for elements in appropriate kernels (see Equation (6.4)). The rest is just solving linear equations. It follows that it is straightforward to apply Grover's algorithm, and we can expect roughly a quadratic speed-up in the search phase, i.e., we can find a vector in the kernel with a number of trials which is about

$$T_e = O(\sqrt{q^{rd-1}}). \tag{6.17}$$

We could also think to apply a quantum algorithm for solving the linear systems like for example HHL [HHL09]. However, in our case, there is no benefit from doing so, since HHL requires a large amount of quantum memory, and is not particularly suited for the systems that we have. Therefore, we decided to simply "Groverize" our attack. For the design of the oracle, we can reuse [SW16] with a small modification. The modification is that in [SW16] the authors use multiple variables and the cost in number of gates is $2m(n^2 + 2n)$. However, we are using a linear system which means that for our attack, using an $m \times n$ matrix and a vector of length n, we have a gate complexity of mn.

6.4 — Case Study: McNie

6.4.1 – Recovering the Secret Key in McNie. Recall the generic structure of our attack from Algorithm 21. The two main procedures are SolveH and SolveF. The first recovers the coefficients of a key equivalent to $\mathbf{H}$ and is generic for LRPC cryptosystems. SolveF, instead, finds the secret basis F and the rest of the secret key. In the case of McNie the secret key is $\mathsf{sk} = (\mathbf{H}, \mathbf{S})$ where $\mathbf{S}$ is an invertible $(n-k) \times (n-k)$ matrix.

Recall that for McNie (see Figure 6.1) it is true that:

$$\mathbf{c}_1 \mathbf{H}^\top - \mathbf{c}_2 \mathbf{S}^{-1} = \boldsymbol{e}\mathbf{H}^\top.$$

Suppose that in SolveH we have recovered an equivalent key $\mathbf{H}' = \mathbf{T} \cdot \mathbf{H}$ for some $\mathbf{T} \in GL_{n-k}(\mathbb{F}_q)$. Multiplying the previous equation by $\mathbf{T}^\top$ we obtain

$$\mathbf{c}_1(\mathbf{T} \cdot \mathbf{H})^\top - \mathbf{c}_2((\mathbf{T}^\top)^{-1}\mathbf{S})^{-1} = \boldsymbol{e}(\mathbf{T} \cdot \mathbf{H})^\top$$

i.e., $\mathsf{sk}' = (\mathbf{H}', \mathbf{S}') = (\mathbf{T} \cdot \mathbf{H}, (\mathbf{T}^\top)^{-1}\mathbf{S})$ is an equivalent secret key for $\mathsf{sk} = (\mathbf{H}, \mathbf{S})$, so we can continue with recovering $\mathbf{S}'$ instead of $\mathbf{S}$. Now we can rewrite the previous equation as

$$(\mathbf{c}_1 - \boldsymbol{e})\mathbf{H}'^\top = \mathbf{c}_2\mathbf{S}'. \tag{6.18}$$

Notice that if we know a triple $(\mathbf{m}, \mathbf{e}, (\mathbf{c}_1, \mathbf{c}_2))$ of message, error and ciphertext (which of course anyone can generate from the public key), once the coefficients of $\mathbf{H}'$ are known, the remaining unknowns in Equation 6.18 are the $(n-k)^2$ coefficients of $\mathbf{S}'$ and the d basis elements of F, all in $\mathbb{F}_{q^m}$. Furthermore, seen as a system of equations over $\mathbb{F}_{q^m}$ in these unknowns, Equation 6.18 is a system of $n-k$ linear equations. Hence, by generating at least $\lceil \frac{(n-k)^2+d}{n-k} \rceil = n-k+1$ valid triples $(\mathbf{m}_i, \mathbf{e}_i, (\mathbf{c}_1, \mathbf{c}_2)_i)$ we can form an overdetermined system in $(n-k)^2 + d$ variables. Solving this system will give the remaining parts of the secret key. Thus in the case of McNie we can define SolveF as the procedure that solves this system. Its cost is $\mathsf{Cost}(\mathsf{SolveF}) = ((n-k)^2 + d)^3$ using schoolbook matrix operations. The cost of $n-k+1$ encryptions is negligible in comparison.

Based on the results from this section and Sections 6.1 and 6.3 we have estimated our attack complexity for the McNie parameters given in their NIST submission [GKK+17]. The results are given in Table 6.1. The attack column at Table 6.1 is built upon Equation 6.8 considering $\mathsf{Cost}(\mathsf{SolveF}) = ((n-k)^2 + d)^3$ and $\mathsf{Cost}(\mathsf{SolveH})$ as $\frac{n^3(n-k)^3d^3}{p^3}$ since the cost of $n-k+1$ encryptions is negligible. This gives us the total cost of

$$\frac{2(n-k+1)}{q^{n-k+1-rd}} + q^{((rd-1)t)}\left(\frac{n^3(n-k)^3d^3}{p^3} + (n-k)^2 + d^3\right).$$

We remark that the improvement provided by Grover is not directly over the entire equation but just over $q^{((rd-1)t)}$. The probability column refers to probability of success and probability that an equivalent key exists as shown in Section 6.3.1.2.

We recall that we do not exploit any specific properties of McNie as the attack in [LT18] which dramatically decrease the security of the scheme, we provide a general reaction attack against LRPC cryptosystems. We provide an implementation of the attack using SAGE Math [Th19], and verified that, under the assumption that the kernel vectors have been found, an equivalent key can be successfully found, the code is available at: https://lrpc.cryptme.in/

Table 6.1: McNie parameters proposed to the first round of the NIST competition, complexities and success probability of our proposed attack. The security and attack are in $\log_2$ scale.

n	k	d	r	q	m	Dec. Failure	Security (bits)	Attack (Classical)	Attack (Quantum)	t	Success $\mathrm{Pr}_s \cdot \mathrm{Pr}_{ek}$
93	62	3	5	2	37	2^{-17}	128	136	80	8	$0.5 \cdot 0.8$
105	70	3	5	2	37	2^{-20}	128	137	81	8	$2^{-10} \cdot 0.74$
111	74	3	7	2	41	2^{-17}	192	185	105	8	$0.08 \cdot 0.87$
123	82	3	7	2	41	2^{-20}	192	186	105	8	$2^{-15} \cdot 0.875$
111	74	3	7	2	59	2^{-17}	256	185	105	8	$1 \cdot 0.875$
141	94	3	9	2	47	2^{-20}	256	234	130	8	$2^{-22} \cdot 0.875$
60	30	3	5	2	37	2^{-16}	128	165	95	10	$0.63 \cdot 0.67$
72	36	3	5	2	37	2^{-21}	128	166	96	10	$2^{-20} \cdot 0.75$
76	38	3	7	2	41	2^{-18}	192	226	126	10	$2^{-6} \cdot 0.875$
84	42	3	7	2	41	2^{-21}	192	227	127	10	$2^{-37} \cdot 0.623$
76	38	3	7	2	53	2^{-18}	256	226	126	10	$1 \cdot 0.875$
88	44	3	8	2	47	2^{-20}	256	257	142	10	$2^{-8} \cdot 0.875$

PART II

QUANTUM CRYPTANALYSIS

Chapter 7

Background on Quantum Cryptanalysis

Before we start with the idea about quantum cryptanalysis we need to understand a little bit of quantum computing. One definition for quantum computing can be: It is the field that studies the computational power and other properties of computers built using quantum-mechanical principles. As has been mentioned before, the impact of a quantum computer in cryptography can be negative such as breaking the currently deployed cryptography and that leads us to the area of quantum cryptanalysis.

We can define quantum cryptanalysis as the study of using quantum computers to break cryptosystems. It is not only limited to breaking cryptosystems but includes also understanding the impacts of the quantum algorithms in cryptosystems helping to establish more secure and robust systems. Given this definition we can go a little bit deeper and include that one important area of quantum cryptanalysis is the estimation of resources.

One can see resources as the number of quantum gates and qubits for running a quantum algorithm. For example, finding preimages of a hash function using Grover's algorithm or running Shor's algorithm for solving the Discrete Logarithm Problem (DLP).

This part of the thesis is organized as follows: First, we will give an explanation about quantum gates and quantum circuits. Second, we will present Grover's algorithm since we will need this knowledge for the application of it. At last, we show how to use Grover's algorithm for finding preimages in parallel with low communication cost and no use of memory.

7.1 — Quantum Computation

In this section, we will briefly explain the notation of quantum computation. Quantum computation takes advantage of quantum mechanics which can be described using quantum states. In summary, the quantum state is a superposition of classical states; mathematically, it can be written as a vector of amplitudes. In order to change these amplitudes one can perform a measurement or a unitary operation. It is common to use the Dirac notation to represent a quantum state.

The superposition means that the quantum state can assume any of the classical states and by "classical" state we mean a state in which the system can be found if we observe it, i.e., if we measure the quantum state. A quantum state, or just state $|\Psi\rangle$ is a superposition of classical states and it can be written as

$$|\Psi\rangle = \alpha_0 |0\rangle + \alpha_1 |1\rangle + \cdots + \alpha_{N-1} |N-1\rangle . \tag{7.1}$$

Here α_i is a complex number and it is called the amplitude of $|i\rangle$ in $|\Psi\rangle$ and N is the number of classical states that the system can assume.

7.1.1 – Qubits. In classical computation, the smallest unit of information that a computer has is a bit, which can be 0 or 1. The quantum bit or qubit is the same thing but for a quantum computer. However, in the quantum case the qubit does not assume only 0 or 1, it can stay in a superposition between 0 and 1. Consider a system with 2 basis states and call them $|0\rangle$ and $|1\rangle$. We identify these basis states with two orthogonal vectors $\begin{pmatrix}1\\0\end{pmatrix}$ and $\begin{pmatrix}0\\1\end{pmatrix}$, respectively. A single qubit can be in any superposition

$$\alpha_0 |0\rangle + \alpha_1 |1\rangle\,, \text{ where } |\alpha_0|^2 + |\alpha_1|^2 = 1. \tag{7.2}$$

Accordingly, a single qubit belongs to a vector space $\mathbb{C}^2$.

7.2 — Quantum Circuits

A quantum circuit follows the general idea of classical circuit replacing the elementary "classical" gates such as AND, OR, XOR and NOT by elementary quantum gates. A quantum gate is a unitary transformation on qubits. However, it is hard to do interesting computations with a single qubit. Like classical computers, quantum computers use registers that are composed of multiple qubits.

7.2.1 – Quantum registers. Quantum registers are qubit strings whose length determines the amount of information that they can store. In superposition, each qubit in the register is in a superposition of $|0\rangle$ and $|1\rangle$, and consequently a register of n qubits is in a superposition of all 2^n possible bit strings that could be represented using n "classical" bits. The state space of a size-n quantum register is a linear combination of 2^n basis vectors, each of length 2^n:

$$|\psi_n\rangle = \sum_{i=0}^{2^n-1} a_i |i\rangle \tag{7.3}$$

Example 7.2.1. A three qubit register has the following expansion:

$$|\psi_2\rangle = a_0 |000\rangle + a_1 |001\rangle + a_2 |010\rangle + a_3 |011\rangle + a_4 |100\rangle + a_5 |101\rangle + a_6 |110\rangle + a_7 |111\rangle \tag{7.4}$$

or in a vector form, using the computational basis:

$$|\psi_2\rangle = a_0 \begin{bmatrix}1\\0\\0\\0\\0\\0\\0\\0\end{bmatrix} + a_1 \begin{bmatrix}0\\1\\0\\0\\0\\0\\0\\0\end{bmatrix} + a_2 \begin{bmatrix}0\\0\\1\\0\\0\\0\\0\\0\end{bmatrix} + a_3 \begin{bmatrix}0\\0\\0\\1\\0\\0\\0\\0\end{bmatrix} + a_4 \begin{bmatrix}0\\0\\0\\0\\1\\0\\0\\0\end{bmatrix} + a_5 \begin{bmatrix}0\\0\\0\\0\\0\\1\\0\\0\end{bmatrix} + a_6 \begin{bmatrix}0\\0\\0\\0\\0\\0\\1\\0\end{bmatrix} + a_7 \begin{bmatrix}0\\0\\0\\0\\0\\0\\0\\1\end{bmatrix} \tag{7.5}$$

We recall that each possible bit configuration in the quantum superposition is denoted by the tensor product of its counterpart qubits.

As with single qubits, the squared absolute value of the amplitude associated with a given bit string is the probability of observing that bit string upon collapsing the register to a classical state, that is, when the register is measured the amplitudes of all 2^n possible bit configurations of an n-bit register sum to one:

$$\sum_{i=0}^{2^n-1} |a_i|^2 = 1. \tag{7.6}$$

7.2.2 – Quantum gates. In classical computing, binary values, as stored in a register, pass through logic gates that, given a certain binary input, produce a certain binary output. Mathematically, classical logic gates are described as boolean functions. Quantum logic gates present a certain similarity with classical gates. When a quantum logic gate is applied to quantum registers it maps the current state to another state, transforming the state until it reaches a final state, i.e., the measured state.

From a mathematical point of view, quantum logic gates can be represented as transformation matrices, or linear operators, applied to a quantum register by multiplying the transformation matrix with the matrix representation of the register. All the quantum operations that correspond to quantum gates must be unitary. A complex matrix U is unitary, if $U^{-1} = U^\dagger$, where $U^\dagger$ is the conjugate transpose: $U^\dagger = \overline{U}^T$. It easy to check that $UU^\dagger = U^\dagger U = I$. Unitary operators preserve the inner product of two vectors, geometrically preserving the lengths of the vectors and the angle between them:

$$\langle u| U^\dagger U |v\rangle = \langle u| I |v\rangle = \langle u|v\rangle . \tag{7.7}$$

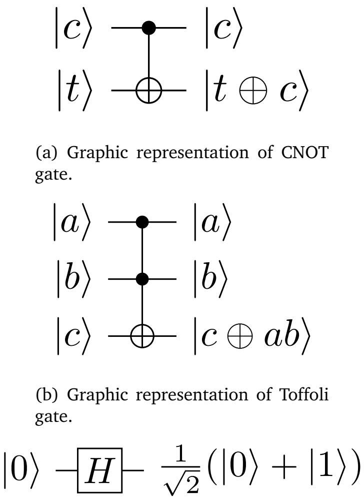
The composition of two unitary operators is also unitary. Given unitary transformation matrices U and V:

$$(UV)^\dagger = V^\dagger U^\dagger = V^{-1}U^{-1} = (UV)^{-1}. \tag{7.8}$$

Since all evolution in a quantum system can be described by unitary matrices and all unitary transformations are invertible, all quantum computation is reversible. For a computation to be reversible the output of the computation contains sufficient information to reconstruct the input, i.e. no input information is erased.

There are several quantum gates each one with a specific function. The most common ones are the bitflip gate X, phaseflip gate Z, Hadamard gate H, controlled-not (CNOT) gate and Toffoli gate. In this thesis, we will be using the Hadamard H, CNOT and Toffoli gate. Figure 7.1 shows the representation of these three gates, i.e., 7.1a is a CNOT gate, 7.1b is a Toffoli gate and 7.1c is a Hadamard gate.

The CNOT operates on two qubits. It flips the target qubit if and only if the control qubit is $|1\rangle$. Figure 7.1a shows this operation where $|t\rangle$ is flipped if $|c\rangle$ is equal to 1. The Toffoli gate operates on three qubits, its operation is similar to CNOT but we add one extra control bit. It flips the target qubit if and only if the two control qubits are $|1\rangle$. This gate is important because it is complete for classical reversible computation: any computation can be implemented by a circuit of Toffoli gates. The Hadamard gate operates only in one qubit but it is crucial for quantum computation. This gate maps the basis state $|0\rangle$ to $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ and $|1\rangle$ to $\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$, which means that we create the “superposition” of the basis states. The superposition can be described as the probability to measure 0 or 1, for more details about superposition and physics aspects of quantum computation and information see [LSP98].



(a) Graphic representation of CNOT gate.

(b) Graphic representation of Toffoli gate.

(c) Graphic representation of Hadamard gate.

Figure 7.1: Quantum gates.

7.3 — Grover's Algorithm

Grover's algorithm [Gro96] is one of the most popular quantum algorithms among cryptographers. This algorithm provides a quadratic speedup for searching an element in an unordered database. Definition 7.1 provides a definition of the search problem.

Definition 7.1. For $N = 2^n$, we are given a function $f : \{0,1\}^N \to \{0,1\}$ which assumes the value 0 for almost all entries. The goal is to find an x such that $f(x) = 1$.

In the classical setting, one needs to perform $\Theta(N)$ queries for finding x, the number of queries varies with the randomness in the search. In the quantum setting, that is using Grover's algorithm, one needs to perform $O(\sqrt{N})$ queries. Figure 7.2 gives a high level abstraction of Grover's algorithm.

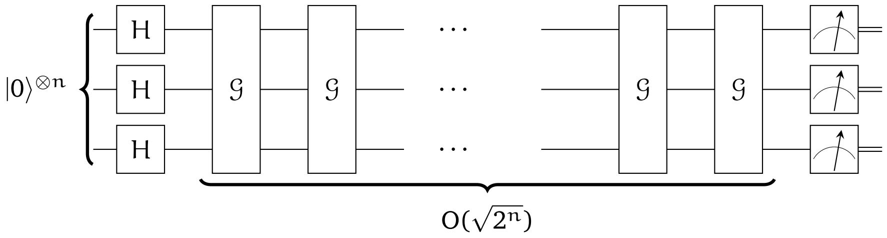


Figure 7.2: High level circuit of Grover's algorithm

In a more detailed level of the algorithm, we can summarize the steps as:

1 Let $N = 2^n$. Initialize the register with n qubits to $|\psi\rangle = |0\rangle^{\otimes n}$;

2 Put the system into an equal superposition state applying Hadamard, i.e. $|\psi\rangle = |H\rangle^{\otimes n} |0\rangle^{\otimes n}$;

3 Repeat $\frac{\pi}{4}\sqrt{N}$ times:

  a) Apply Grover step $\mathcal{G}$;

4 The answer is the measurement of the register $|\psi\rangle$.

The Grover step $\mathcal{G}$ consists of two steps which are an oracle reflection U_f and an additional reflection U_s. The oracle reflection U_f is a function that will call an Oracle $\mathcal{O}$ which returns $f(x) = 1$ for the correct answer and $f(x) = 0$ otherwise. The reflection U_f performs a flipping of the phase over the marked items and can be computed as:

$$U_f |x\rangle = (-1)^{f(x)} |x\rangle .$$

The additional reflection U_s performs inversion about the average, transforming the amplitude of each state so that it is as far above the average as it was below the average prior to the transformation, and vice versa. The U_s can be described as application of Hadamard transform, followed by a conditional phase shift that shifts every state except $|0\rangle$ by -1, followed by another Hadamard transform. It is possible to represent the phase shift by the unitary operator $2|0\rangle\langle 0| - I$, which will map to the following state when the value is 0:

$$[2|0\rangle\langle 0| - I]\,|0\rangle = 2|0\rangle\langle 0|0\rangle - I = |0\rangle \tag{7.9}$$

and it will map to the following state when the value is not 0:

$$[2|0\rangle\langle 0| - I]\,|x\rangle = 2|0\rangle\langle 0|x\rangle - I = -|x\rangle . \tag{7.10}$$

Using the notation presented in the summary $|\psi\rangle = |H\rangle^{\otimes n} |0\rangle^{\otimes n}$ and the entire diffusion transform, we have:

$$H^{\otimes n}[2|0\rangle\langle 0| - I]H^{\otimes n} = 2H^{\otimes n}|0\rangle\langle 0|H^{\otimes n} - I = 2|\psi\rangle\langle\psi| - I \tag{7.11}$$

and the entire Grover iteration can be represented as:

$$\mathcal{G} = [2|\psi\rangle\langle\psi| - I]\mathcal{O}. \tag{7.12}$$

The implementation of $\mathcal{O}$, that is $f(x)$, depends on the search problem. In fact, $f(x)$ will determine the depth of the circuit. In Chapter 9, we will show a quantum circuit for finding preimages of a hash function.

In summary, if one wants to build a quantum circuit for Grover's algorithm it is necessary to consider the costs besides the constructions of the oracle, i.e., it is necessary to consider the costs for the initial Hadamard transformation and the costs for Grover's iteration. However, for most of the cases is safe to assume that the cost for applying Grover's algorithm is $O(\sqrt{N})$.

Chapter 8

AES in a Quantum Computer

As we mentioned in Chapter 7, quantum computations need to be reversible. Furthermore, the oracle $\mathcal{O}$ present in Grover's algorithm needs to be implemented as a reversible function. One interesting case for this thesis is the implementation of AES. A reversible version of AES has already been shown in [GLRS16].

In this chapter, we will show an improvement in the number of gates to the AES implementation proposed in [GLRS16]. Our improvement is during the SubBytes function, where we propose a new circuit for squaring and a different addition chain for computing the multiplicative inverse. Since the publication of [GLRS16] two papers, [ASAM18] and [KHJ18], already provided some improvements. Our improvements are bigger than both of these. Moreover, we provide an implementation using a library called libquantum[1]. Libquantum is a C library for the simulation of quantum mechanics created by Björn Butscher and Hendrik Weimer. The library has a special focus on quantum computing. Libquantum presents an easy-to-use interface and a range of quantum gates such as CNOT, Toffoli, and Hadamard, and one can define a new gate by giving a matrix that describes the gate.

8.1 — The AES Block Cipher

We first will give a brief explanation about AES. Later on, we will give details about the classical and quantum implementation of each function. AES is a block cipher, designed by Daemen and Rijmen [DR13]. It is based on Rijndael but only provides 128-bit blocks. The key length can be 128, 192 and 256. We will be considering just AES-128, i.e., AES with a key size of 128 bits. The main reason for this consideration is that libquantum does not support more than 128 qubits.

AES has different transformations operating on an intermediate result that is called *State*. The state can be seen as an array of bytes, with four rows and four columns. The number of rounds N_r depends on the size of the key, e.g., AES-128 performs 10 rounds, AES-192 performs 12 rounds and AES-256 performs 14 rounds.

In the encryption process with AES, one needs first to perform key addition, denoted by *AddRoundKey*, followed by $N_r - 1$ executions of *Round*, and finally one application of *FinalRound*. The *Round* function is the application of 4 transformations which are

[1] http://libquantum.de/

SubBytes, *ShiftRows*, *MixColumns* and *AddRoundKey*. The *FinalRound* consists of the application of *SubBytes*, *ShiftRows* and *AddRoundKey*. The code below shows in a pseudo C language, how those rounds are put together. One advantage of AES is that one just needs to implement the transformation functions and then reuse them in the rounds.

```
AES(State, CipherKey){
  KeyExpansion(CipherKey, ExpandedKey);
  AddRoundKey(State, ExpandedKey[0]);
  for(i = 1; i < N_r; i++){
    Round(State, ExpandedKey[i]);
  }
  FinalRound(State, ExpandedKey[N_r]);
}
```

Listing 8.1: High level description of AES.

8.1.1 – SubBytes. The SubBytes operation is a non-linear byte substitution and operates on each byte of the state. In classical computation, it is common to use tables for this operation. Those substitution tables are known as S-boxes. For AES the S-Box is defined using inversion on $\mathbb{F}_{2^8}^*$ and an affine transformation $\alpha \mapsto a \cdot \alpha^{-1} + b$ for $\alpha \in \mathbb{F}_{2^8}^*$ and $0 \mapsto b$ for some $a, b \in \mathbb{F}_{2^8}^*$. For full details see [DR13].

8.1.1.1 – Quantum implementation. Unfortunately, the SubBytes operation is more complicated in a quantum setting than a classical setting. For instance, the cost for implementing and accessing tables is not exactly known. One option is the solution presented in [GLRS16] where a state byte is an element $\alpha \in \mathbb{F}_2[x]/(x^8 + x^4 + x^3 + x + 1)$. Then one first computes the multiplicative inverse of α followed by the affine transformation.

8.1.2 – ShiftRows. The ShiftRows step operates on the rows of the state, it cyclically shifts the bytes of each row. The first row is left unchanged. The second row is shifted one to the left, the third row is shifted two to the left. The last and fourth row is shifted three to the left.

8.1.2.1 – Quantum implementation. The quantum implementation of this step does not use any gate to be performed. This step is a permutation of the AES state and it does not add any new gate to the circuit. However, one needs to rearrange the position of the subsequent gates assuring that they receive the correct input.

8.1.3 – MixColumns. As the name suggests the MixColumns step operates on the columns of the state. In fact, it is a bricklayer permutation operating on the state column by column. Each column is seen as a polynomial over $\mathbb{F}_{2^8}$ and is then multiplied modulo $z^4 + 1$ with a fixed polynomial $c(z)$, for more details about the selection of $c(z)$ see [DR13]. This operation can be seen as a matrix multiplication by a vector. The matrix is generated by the fact that $c(z)$ is a fixed polynomial and it can be found in the original description of AES provided by NIST [Nat01] and in [DR13].

8.1.3.1 – Quantum implementation. In a quantum setting, this step can be implemented similar to the classical setting, i.e., the step is applied to an entire column of the state and this means that it operates on 32 qubits at a time. In [GLRS16], the authors adapt the creation of a LUP-type decomposition matrix using just CNOT gates to replicate the result of the original matrix in [DR13].

8.1.4 – AddRoundKey. In this step, an expanded key is combined with the state, i.e., the state is modified by combining it with the round key using the bitwise XOR operation. We regard Listing 8.1, where the first step is to expand the key into $N_r + 1$ round keys, a round key is denoted by $ExpandedKey[i], 0 \leqslant i \leqslant N_r$.

8.1.4.1 – Quantum implementation. The implementation in a quantum circuit of AddRoundKey can be done with a parallel execution of CNOT gates. To be more precise, for a key of 128 bits the circuit generated contains only 128 CNOT gates.

8.2 — Background on the Quantum Languages

In this chapter, we present the implementation of AES using libquantum and the translator from a C based language to openQASM [CBSG17]. However, let us first introduce the languages. We will only present usage of CNOT and Toffoli gates since the AES implementation does not use other gates.

8.2.1 – Syntax of libquantum. In Listing 8.2, we present the generation of a quantum register using the function *quantum_new_qureg* and we give two parameters. The first parameter is the value to which the register is initialized; in our case it is 6 that is represented by 0110 in binary, the second parameter is the number of qubits that the register will hold; in our example 4 qubits.

After the creation of a quantum register, we apply the CNOT gate using function *quantum_cnot*, which is similar to the description of the gate, that is, one gives the position of the control and target qubits as integers starting from 0 on the right. The third argument is the register to which the gate is applied. In our example, this appears in Line 5 where we select the qubit at position 1 as our control and qubit at position 0 as our target.

The last relevant line for the example is a call of *quantum_toffoli* which is a Toffoli gate, and it has four parameters, the first two numbers represent the control qubits followed by the target qubit and the register to which the gate is applied.

Figure 8.1 shows the result of the computation in Listing 8.2, i.e., the figure shows initial state followed by the application of CNOT which changes the qubit at position 0. After that, we have the application of the Toffoli gate, which uses the qubits at positions 1 and 2 and flips the qubit at position 0. The last quantum function that we use is “quantum_measure” which performs a measurement on the whole quantum register, the result is an integer value. At last, we call “quantum_delete_qureg” which is not a function for quantum computing. The function deletes a quantum register and frees its allocated memory.

```
1 #include <quantum.h>
2 int main(void) {
3   quantum_reg q_reg = quantum_new_qureg(6, 4);
4 
5   quantum_cnot(1, 0, &q_reg);
6 
7   quantum_toffoli(1, 2, 0, &q_reg);
8 
```

```
  int result =  quantum_measure(&reg);

  quantum_delete_qureg(q_reg);

  return 0;
}
```

Listing 8.2: Basic generation of a quantum register and usage of CNOT and Toffoli gates using libquantum.

$$\begin{array}{l}
1.000000\ +0.000000i\,|6\rangle\ (1.000000e+00)\ (|0110\rangle) \\
1.000000\ +0.000000i\,|7\rangle\ (1.000000e+00)\ (|0111\rangle) \\
1.000000\ +0.000000i\,|6\rangle\ (1.000000e+00)\ (|0110\rangle)
\end{array}$$

Figure 8.1: Result of the circuit presented in Listing 8.2.

8.2.2 – Syntax of OpenQASM. Listing 8.3 shows the same circuit as Listing 8.2 using the low level language OpenQASM. However, the reader can notice that on lines 4 and 5 there are two “extra” gates, the “x” gate is just a NOT gate used to initialize bits 1 and 2 to value 1. In an analogous way, we can say that openQASM is similar to the Assembler language for classical computers.

```
1 OPENQASM 2.0;
2 include "qelib1.inc";
3 qreg reg[4];
4 x reg[1];
5 x reg[2];
6 cx reg[1], reg[0];
7 ccx reg[1], reg[2], reg[0];
```

Listing 8.3: Basic generation of a quantum register and usage of CNOT and Toffoli gates using openQASM.

8.2.3 – openQASM translator. As it was mentioned, we first implemented AES using libquantum which is a C based language that simulates quantum computations. One advantage of libquantum is that it is a library written in C and follows the code style. If one is familiar with C standards then it is easy to write a circuit using the library. However, this library is limited in tools for analyzing a quantum circuit, i.e., it does not present any algorithm for optimization or for checking the circuit. One could use the algorithms presented in [Amy13] to optimize a circuit. Fortunately, the implementations of those algorithms are included in openQASM [CBSG17].

In order to translate the libquantum code to openQASM, we develop a small translator that is able to identify the type of gate and the qubits of the operation and translates into openQASM language.

The translator works as follows: First, it parses the file, this is done line by line and in this step the translator saves each line as a text string and it removes empty lines. After that, it starts from the beginning of the list with the generation of each gate, it uses regular expressions for finding the type of gate and generates an object that knows which type of gate it is, the qubits of control and qubit of target and saves each gate in a list of gates. In the final step, the translator gets the list of gates and writes each gate and all qubits in a valid openQASM syntax. Listing 8.4 shows the functions for getting the lines, and transforming CNOT and Toffoli gates in our intermediate representation. The code for all the functions is available at https://quantum.cryptme.in.

```
def __get_lines(self):
    line = self.fp.readline()
    cnt = 1
    lines = []
    while line:
        if line.strip():
            lines.append(line.strip())
        line = self.fp.readline()
        cnt += 1
    return lines
def __transform(self):
    cnot_present = re.find(" cnot");
    toffoli_present = re.find(" toffoli");
    if cnot_present:
        self.__solve_cnot()
    elif toffoli_present:
        self.__solve_toffoli()
    else:
        raise GateUnkownError("unknown gate: ", self.line)
def __solve_cnot(self):
    result = re.sub("[a-z_&;()]", "", self.line)
    r_split = result.strip().split(",")
    self.qbits_control.append(r_split[0])
    self.qbit_target = r_split[1]
    self.gate_type = GatesEnum.CNOT
def __solve_toffoli(self):
    result = re.sub("[a-z_&;()]", "", self.line)
    r_split = result.strip().split(",")
    self.qbits_control.append(r_split[0])
    self.qbits_control.append(r_split[1])
    self.qbit_target = r_split[2]
    self.gate_type = GatesEnum.TOF
```

Listing 8.4: Functions for getting lines and extracting information from the gates.

8.3 — Improved AES Implementation

We implemented the AES circuit presented in [GLRS16] using the language and tools presented before. We notice that the implementation could save gates during the computation of the inverse. We give more details further in the text.

In our implementation, we use a total of 15 276 gates for one round of AES, in which 7 168 are Toffoli gates, 8 044 are CNOT gates and 64 NOT gates. More specifically, we will now give the number of gates for each function of AES. As stated before ShiftRows does not require any gates.

8.3.1 – SubBytes. The first function that AES performs is SubBytes. It is possible to break this function into two and quantify the number of gates for the inverse and the affine function. The inverse can be computed using $\alpha^{-1} = \alpha^{254}$. Note that this form of computing the inverse by exponentiation means that $\alpha = 0$ need not be handled separately.

In [GLRS16], the authors compute the inverse using

$$\alpha^{-1} = \alpha^{254} = ((\alpha \cdot \alpha^2) \cdot (\alpha \cdot \alpha^2)^4 \cdot (\alpha \cdot \alpha^2)^{16} \cdot \alpha^{64})^2.$$

Using this approach and returning all ancillas clean uses 8 multiplications and 29 squarings. In our implementation, the inverse function uses a different addition chain and it requires 7 multiplications and 19 squarings. Like [GLRS16] we use 25 ancilla qubits.

In order to show where we can perform fewer squarings we first show the computation of $\alpha^{-1} = \alpha^{254}$ presented in [GLRS16]. Figure 8.2 shows the inverse used in [GLRS16], the symbols on the step column means: (*) when a multiplication between two values occurs, (^) when a squaring or multi-squaring occurs, (^ ^) when an out-of-place squaring or multi-squaring occurs which cost 8 extra CNOTS. An out-of-place squaring is the operation $(a, 0) \mapsto (a, a^2)$ while in place squaring is $a \to a^2$. A multi-squaring maps $a \mapsto a^{2^n}$ for some $n \in \mathbb{Z}$. After a verification, we can notice that there are 8 multiplications and 29 squarings to compute and uncompute the multiplicative inverse. In the original implementation, the authors claim that it is possible to perform all squarings in and out of place using 275 CNOT gates. However, we could not verify this since there are no more details about those operations other than the circuit for a single squaring and an out-of-place squaring. Furthermore, it is easily possible to improve this addition chain by swapping the order of steps 14 and step 15 and decrease 2 squarings and saving one multiplication at the expense of an out-of-order 4-th power in step 5.

Figure 8.3 shows our proposal for computing the inverse. In the end, we compute 19 squarings and 7 multiplications.

Figure 8.4 shows a circuit for squaring an element. Each squaring uses 10 CNOT gates as shown in Figure 8.4, squaring can be achieved as shown below:

Given a polynomial $p(x) = a_7x^7 + a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x^1 + a_0x^0$ with $a_i \in \mathbb{F}_2$ and $p(x) \in \mathbb{F}_2[x]/(x^8 + x^4 + x^3 + x + 1)$, we compute $p(x)^2$. At first, the polynomial will be $p(x)^2 = a_7x^{14} + a_6x^{12} + a_5x^{10} + a_4x^8 + a_3x^6 + a_2x^4 + a_1x^2 + a_0x^0$ and we need to reduce the polynomial using $x^8 + x^4 + x^3 + x + 1$. We can rewrite the elements larger than x^7 as:

$$\begin{aligned}
x^8 &\equiv x^4 + x^3 + x + 1 && \bmod x^8 + x^4 + x^3 + x + 1 \\
x^9 &\equiv x^5 + x^4 + x^2 + x && \bmod x^8 + x^4 + x^3 + x + 1 \\
x^{10} &\equiv x^6 + x^5 + x^3 + x^2 && \bmod x^8 + x^4 + x^3 + x + 1 \\
x^{12} &\equiv x^8 + x^7 + x^5 + x^4 && \bmod x^8 + x^4 + x^3 + x + 1 \\
x^{14} &\equiv x^{10} + x^9 + x^7 + x^6 && \bmod x^8 + x^4 + x^3 + x + 1
\end{aligned}$$

Qubits position Step	$0\dots7$	$8\dots15$	$16\dots23$	$24\dots31$	$32\dots39$
0	α				
1 ^ ^	α	α^2			
2 *	α	α^2	α^3		
3 ^ ^	α	α^2	α^3	α^{12}	
4 *	α	α^2	α^3	α^{12}	α^{15}
5 *	α	α^2		α^{12}	α^{15}
6 ^	α	α^2		α^{48}	α^{15}
7 *	α	α^2	α^{63}	α^{48}	α^{15}
8 ^ ^	α		α^{63}	α^{48}	α^{15}
9 ^	α^{64}		α^{63}	α^{48}	α^{15}
10 *	α^{64}	α^{127}	α^{63}	α^{48}	α^{15}
11 ^	α^{64}	α^{254}	α^{63}	α^{48}	α^{15}
12 ^	α	α^{254}	α^{63}	α^{48}	α^{15}
13 *	α	α^{254}		α^{48}	α^{15}
14 ^ ^	α	α^{254}	α^3	α^{48}	α^{15}
15 ^	α	α^{254}	α^3	α^{12}	α^{15}
16 *	α	α^{254}	α^3	α^{12}	
17 ^ ^	α	α^{254}	α^3		
18 ^ ^	α	α^{254}	α^3	α^2	
19 *	α	α^{254}		α^2	
20 ^ ^	α	α^{254}			

Figure 8.2: Reversible computation of $\alpha^{-1} = \alpha^{254}$ proposed in [GLRS16].

After a first step of reduction, it is possible to notice that some elements continue bigger than x^7, for example x^{14}, which needs one more step of reduction. We show below how x^{14} can be reduced:

$$\begin{aligned} x^{14} &\equiv x^{10} + x^9 + x^7 + x^6 && \text{mod } x^8 + x^4 + x^3 + x + 1 \\ &\equiv x^6 + x^5 + x^3 + x^2 + x^5 + x^4 + x^2 + x + x^7 + x^6 && \text{mod } x^8 + x^4 + x^3 + x + 1 \\ &\equiv x^7 + x^4 + x^3 + x && \text{mod } x^8 + x^4 + x^3 + x + 1 \end{aligned}$$

In the end we can rewrite $a_7x^{14} + a_6x^{12} + a_5x^{10} + a_4x^8 + a_3x^6 + a_2x^4 + a_1x^2 + a_0x^0$ as:

$$\begin{aligned} a_7(x^7 + x^4 + x^3 + x) + a_6(x^7 + x^5 + x^3 + x + 1) + a_5(x^6 + x^5 + x^3 + x^2) + \\ a_4(x^4 + x^3 + x + 1) + a_3x^6 + a_2x^4 + a_1x^2 + a_0x^0, \end{aligned}$$

which can be rearranged to be

$$\begin{aligned} (a_7 + a_6)x^7 + (a_5 + a_3)x^6 + (a_6 + a_5)x^5 + (a_7 + a_4 + a_2)x^4 + \\ (a_7 + a_6 + a_5 + a_4)x^3 + (a_5 + a_1)x^2 + (a_7 + a_6 + a_4)x + (a_6 + a_4 + a_0)x^0. \end{aligned} \tag{8.1}$$

Qubits position Step	$0\ldots7$	$8\ldots15$	$16\ldots23$	$24\ldots31$	$32\ldots39$
0	α				
1 ^ ^	α	α^2			
2 *	α	α^2	α^3		
3 ^ ^	α	α^2	α^3	α^{12}	
4 ^ ^	α		α^3	α^{12}	
5 *	α	α^{15}	α^3	α^{12}	
6 ^	α	α^{60}	α^3	α^{12}	
7 ^ ^	α	α^{60}	α^3		
8 *	α	α^{60}	α^3	α^{63}	
9 ^	α	α^{60}	α^3	α^{126}	
10 *	α	α^{60}	α^3	α^{126}	α^{127}
11 ^	α	α^{60}	α^3	α^{126}	α^{254}
12 ^	α	α^{60}	α^3	α^{63}	α^{254}
13 *	α	α^{60}	α^3		α^{254}
14 ^ ^	α	α^{60}	α^3	α^{12}	α^{254}
15 ^	α	α^{15}	α^3	α^{12}	α^{254}
16*	α		α^3	α^{12}	α^{254}
17 ^ ^	α	α^2	α^3	α^{12}	α^{254}
18 ^ ^	α	α^2	α^3		α^{254}
19 *	α	α^2			α^{254}
20 ^ ^	α				α^{254}

Figure 8.3: Reversible computation of $\alpha^{-1} = \alpha^{254}$ proposed in this chapter.

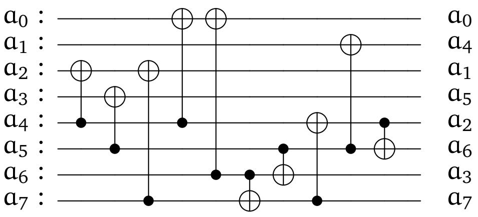


Figure 8.4: Squaring of an element of $\mathbb{F}[x]_2/(x^8 + x^4 + x^3 + x + 1)$ using quantum gates.

In Figure 8.4 we show our squaring circuit matching Equation 8.1. We bring to the reader's attention that in the figure the execution of the circuit is sequential from left to right and that the output has a different order from the input. Our circuit uses 10 CNOTs instead of 12 CNOTs in [GLRS16].

The multiplication uses the same circuit as the original provided in [GLRS16]; Figure 8.5 shows its circuit. We did an attempt to improve this using the tools in open-

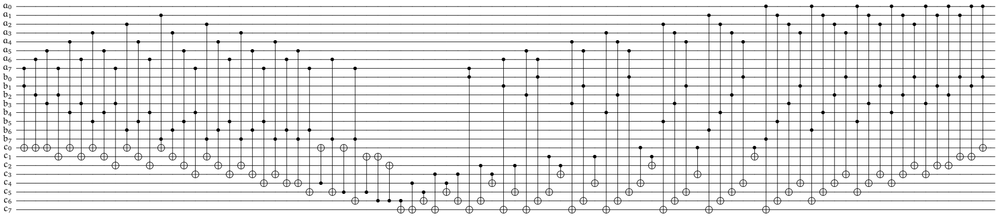


Figure 8.5: Multiplication circuit provided in [GLRS16]

QASM [CBSG17] and [Amy13]. However, this does not decrease the number of gates in the multiplication circuit, which are 64 Toffoli gates and 21 CNOT gates.

In total, the inverse function uses $7 \cdot 21 + 19 \cdot 10 + 8 \cdot 8 = 401$ CNOT and $7 \cdot 64 = 448$ Toffoli gates. The affine function uses a total of 24 CNOT and 4 NOT gates using the LUP decomposition of the matrix given in [Nat01] as follows

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

The total number of gates for SubBytes is 4 NOT, 425 CNOT and 448 Toffoli gates per state, in AES-128 we have a total of 16 states. The total number in one round is 64 NOT, 6 800 CNOT and 7 168 Toffoli gates. If the SubBytes function is applied sequentially on the 16 states the 24 ancillas suffice for the entire round. Higher levels of parallelism and thus reduced depth increase the ancilla usage.

Comparison. To compare with [GLRS16] we need to express our result in terms of T-gates and Clifford gates. We use the same decomposition as [GLRS16] and decompose 1 Toffoli gate as 7 T gates + 8 Clifford gates. We remark that this gives an upper bound on the number of T gates as we use the generic decomposition; the circuits above could be built using T-gates directly and possibly use fewer T gates. In [AG04], the authors show the Gottesman-Knill theorem, which states that NOT and CNOT gates belong to the Clifford group, thus we can use our NOT and CNOT gates for comparison.

Table 8.1 shows the comparison with the circuit designs in the literature. The values from [GLRS16] were acquired directly from their paper. For the values in [ASAM18] and [KHJ18] we use the same decomposition of Toffoli gates since they present their results in Toffoli and CNOT gates. In the squaring count we consider 4-th power and 64-th power as 2 and 6 squarings, respectively. For [KHJ18] we use the 12-CNOT squaring function as no better one was stated before our work. Note that [ASAM18] has a squaring circuit with 19 CNOTs.

8.3.2 – Other functions. After SubBytes, we execute the MixColumns function. Based on [GLRS16] the function uses a total of 279 CNOT gates. The function is called 4 times as we described in Section 8.1.3. The total number of gates for the MixColumns step is 1 116 CNOT gates. At last, we have the AddRoundKey function, which uses 128 CNOT

Table 8.1: Comparison of circuit designs for finding multiplicative inverse.

	Work in [GLRS16]	Work in [ASAM18]	Work in [KHJ18]	This work
Number of qubits	40	64	40	40
Number of multiplications	8	7	7	7
Number of squarings	29	14	33	19
Gates for one S-Box	3 584 T	3 136 T	3 136 T	3 136 T
	4 569 Clifford	4 072 Clifford	4 082 Clifford	3 866 Clifford

gates. We use a total of 7 168 Toffoli gates, 8 044 CNOT gates and 64 NOT gates for one round of AES.

Chapter 9

Grover's Algorithm and Preimage Search

9.1 — Introduction

Fix a function H. For any element x in the domain of H, the value $H(x)$ is called the image of x, and x is called a preimage of $H(x)$.[1]

Many attacks can be viewed as searching for preimages of specified functions. Consider, for example, the function H that maps an RSA private key (p, q) to the public key pq. Formally, define P as the set of pairs (p, q) of prime numbers with $p < q$, and define $H : P \to \mathbb{Z}$ as the function $(p, q) \mapsto pq$. Shor's quantum algorithm efficiently finds the private key (p, q) given the public key pq; in other words, it efficiently finds a preimage of pq.

As another example, consider a protocol that uses a secret 128-bit AES key k, and that reveals the encryption under k of a plaintext known to the attacker, say plaintext 0. Given this ciphertext $\text{AES}_k(0)$, a simple brute-force attack takes a random key x as a guess for k, computes $\text{AES}_x(0)$, and checks whether $\text{AES}_x(0) = \text{AES}_k(0)$. If $\text{AES}_x(0) \neq \text{AES}_k(0)$ then the attack tries again, for example replacing x with $x + 1 \bmod 2^{128}$.

Within, e.g., 2^{100} guesses the attack has probability almost 2^{-28} of successfully guessing k. We say "almost" because there could be preimages of $H(k)$ other than k: i.e., it is possible to have $H(x) = H(k)$ with $x \neq k$. This gives the attack more chances to find a preimage, but it means that any particular preimage selected as output is correspondingly less likely to be k. Typical protocols give the attacker a reasonably cheap way to see that these other preimages are not in fact k, and then the attacker can simply continue the attack until finding k.

This brute-force attack is not specific to AES, except for the details of how one computes $\text{AES}_k(0)$ given k. The general strategy for finding preimages of a function is to check many possible preimages. In this chapter we focus on faster attacks that work in the same level of generality. Some specific functions, such as the function $(p, q) \mapsto pq$ mentioned above, have extra structure allowing much faster preimage attacks, but we do not discuss those special-purpose attacks further.

[1] This chapter is based on a paper with Daniel J. Bernstein and it was published at Selected Areas in Cryptography 2017 [BB17] and it was presented at Quantum Cryptanalysis (Dagstuhl Seminar 17401) [MSSS18].

9.1.1. Multiple-target preimages. Often an attacker is given many images, say t images $H(x_1), \ldots, H(x_t)$, rather than merely a single image. For example, $x_1, \ldots, x_t$ could be secret AES keys for sessions between t pairs of users, where each key is used to encrypt plaintext 0; or they could be secret keys for one user running a protocol t times; or they could be secrets within a single protocol run.

The t-target preimage problem is the problem of finding a preimage of at least one of $y_1, \ldots, y_t$; i.e., finding x such that $H(x) \in \{y_1, \ldots, y_t\}$. A solution to this problem often constitutes a break of a protocol; and this problem can be easier than the single-target preimage problem, as discussed below.

Techniques used to attack the t-target preimage problem are also closely related to techniques used to attack the well-known collision problem: the problem of finding distinct x, x' with $H(x) = H(x')$.

The obvious way to attack the t-target preimage problem is to choose a random x and see whether $H(x) \in \{y_1, \ldots, y_t\}$. Typically $y_1, \ldots, y_t$ are distinct, and then the probability that $H(x) \in \{y_1, \ldots, y_t\}$ is the sum of the probability that $H(x) = y_1$, the probability that $H(x) = y_2$, and so on through the probability that $H(x) = y_t$. If x is a single-target preimage with probability about $1/N$ then x is a t-target preimage with probability about t/N.

Repeating this process for s steps takes a total of s evaluations of H on distinct choices of x, and has probability about st/N of finding a t-target preimage, i.e., high probability after N/t steps. This might sound t times faster than finding a single-target preimage, but there are important overheads in this algorithm, as we discuss next.

9.1.2. Communication costs and parallelism. Real-world implementations show that, as t grows, the algorithm stated above becomes bottlenecked not by the computation of $H(x)$ but rather by the check whether $H(x) \in \{y_1, \ldots, y_t\}$.

One might think that this check takes constant time, looking up $H(x)$ in a hash table of $y_1, \ldots, y_t$, but the physical reality is that random access to a table of size t becomes slower as t grows. Concretely, when a table of size t is laid out as a $\sqrt{t} \times \sqrt{t}$ mesh in a realistic circuit, looking up a random table entry takes time proportional to $\sqrt{t}$.

Furthermore, for essentially the same cost as a memory circuit capable of storing and retrieving t items, the attacker can build a circuit with t small parallel processors, where the ith processor searches for a preimage of y_i independently of the other processors. Running each processor for N/t fast steps has high success probability of finding a t-target preimage and takes total time N/t, since the processors run in parallel.

The "parallel rho method", introduced by van Oorschot and Wiener in 1994 [vOW94], does better. The van Oorschot–Wiener circuit has size p and reaches high probability after only N/pt fast steps. For example, with $p = t$, this circuit has size t and reaches high probability after only N/t^2 steps.

There are p small parallel processors in this circuit, arranged in a $\sqrt{p} \times \sqrt{p}$ square. There is also a parallel "mesh" network allowing each processor to communicate quickly with the processors adjacent to it in the square. Later, as part of the description of our quantum multi-target preimage-search algorithm, we will review how these resources are used in the parallel rho method. The analysis also shows how large p and t can be compared to N.

9.1.3. Quantum attacks. If a random input x has probability $1/N$ of being a preimage of y then brute force finds a preimage of y in about N steps. Quantum computers do better: specifically, Grover's algorithm [Gro96] finds a preimage of y in only about $\sqrt{N}$ steps as explained in Chapter 7.

However, increased awareness of communication costs and parallelism has produced increasingly frequent objections to this quantitative speedup claim. For example, NIST's "Submission Requirements and Evaluation Criteria for the Post-Quantum Cryptography Standardization Process" [NIS16] states security levels for AES-128, AES-192, and AES-256 that provide

> " substantially more quantum security than a naïve analysis might suggest. For example, categories 1, 3 and 5 are defined in terms of block ciphers, which can be broken using Grover's algorithm, with a quadratic quantum speedup. But Grover's algorithm requires a long-running serial computation, which is difficult to implement in practice. In a realistic attack, one has to run many smaller instances of the algorithm in parallel, which makes the quantum speedup less dramatic."

Concretely, Grover's algorithm has high probability at finding a preimage if it uses p small parallel quantum processors, each running for $\sqrt{N/p}$ steps, as in [GR03]. The speedup compared to p small parallel pre-quantum processors is only $\sqrt{N/p}$, which for reasonable values of p is much smaller than $\sqrt{N}$.

Furthermore, when the actual problem facing the attacker is a t-target preimage problem, the parallel rho machine with p small parallel pre-quantum processors reaches high success probability after only N/pt steps. This extra factor t can easily outweigh the $\sqrt{N/p}$ speedup from Grover's algorithm.

For example, a parallel rho machine of size p finds collisions in only $\sqrt{N}/p$ steps. This is certainly better than running Grover's algorithm for $\sqrt{N/p}$ steps.

Brassard, Høyer, and Tapp [BHT98] claimed a faster quantum algorithm to find collisions. Their algorithm chooses $t \approx N^{1/3}$, takes t random inputs $x_1, \ldots, x_t$, computes the corresponding images $y_1, \ldots, y_t$, and then builds a new function H' defined as follows: $H'(x) = 0$ if $H(x) \in \{y_1, \ldots, y_t\}$, otherwise $H'(x) = 1$. A random input is an H'-preimage of 0 with probability approximately $1/N^{2/3}$, so Grover's algorithm finds an H'-preimage of 0 after approximately $N^{1/3}$ steps.

However, Bernstein [Ber09] analyzed the communication costs in this algorithm and in several variants, and concluded that no known quantum collision-finding algorithms were faster than the pre-quantum parallel rho method.

NIST has stated that resistance to multi-key attacks is desirable. Our results show that simply using Grover's algorithm for single-target preimage search is not optimal in this context. NIST's post-quantum security claims for AES-128, AES-192, and AES-256 assume that it is optimal, and therefore need to be revised.

9.2 — Reversible Computation

We recall that a Toffoli gate maps bits (x, y, z) to $(x, y, z \oplus xy)$, where $\oplus$ means exclusive-or.

A reversible n-bit circuit is an n-bit-to-n-bit function expressed as a composition of a sequence of Toffoli gates on selected bits. We assume that adjacent Toffoli gates on separate bits are carried out in parallel: our model of time for a reversible circuit is the depth of the circuit rather than the total number of gates. To model realistic communication

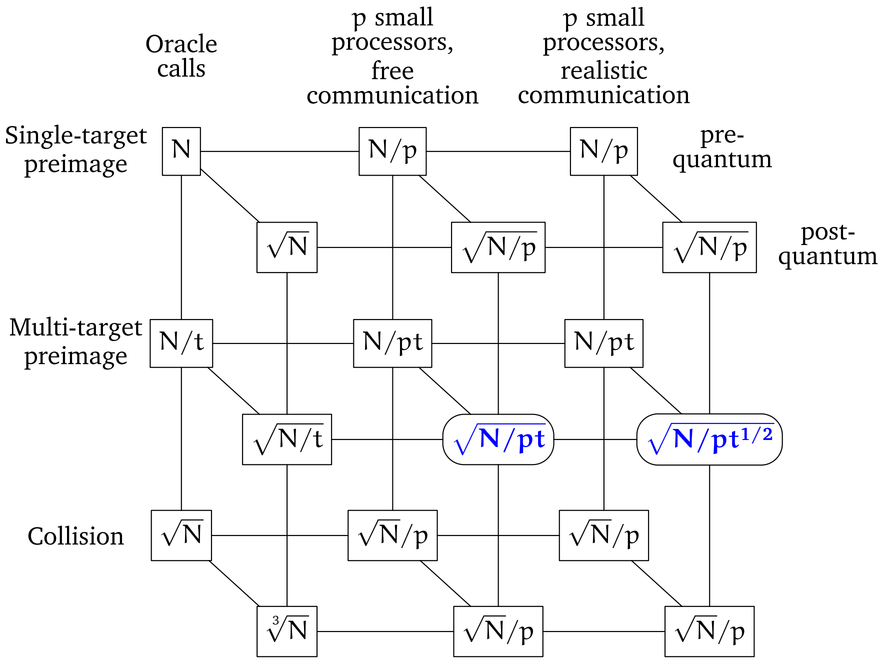


Figure 9.1: Overview of costs of pre-quantum and post-quantum attacks. Circled blue items are new results in this chapter. Lower-order factors are omitted. Pre-quantum single-target preimage attacks: brute force plus simple parallelization. Post-quantum single-target preimage attacks: Grover's algorithm [Gro96] plus simple parallelization [GR03]. Pre-quantum multi-target preimage attacks: brute force and the parallel rho method [vOW94]. Post-quantum multi-target preimage attacks: [HRS16] for oracle calls, this chapter for parallel methods. Pre-quantum collision attacks: the rho method and the parallel rho method. Post-quantum collision attacks: [BHT98] for oracle calls, plus the parallel rho method.

costs, we lay out the n bits in a square, and we require each Toffoli gate to be applied to bits that are laid out within a constant distance of each other.

Let H be a function from $\{0,1\}^b$ to $\{0,1\}^b$, where b is a nonnegative integer. An a-ancilla reversible circuit for H is a reversible $(2b+a)$-bit circuit that maps $(x, y, 0)$ to $(x, y \oplus H(x), 0)$ where x and y each have b bits and where the 0 has a bits. The behavior of this circuit on more general inputs (x, y, z) is not relevant.

Grover's method, given a reversible circuit for H, produces a quantum preimage-search algorithm. This algorithm uses s serial steps of H computation and negligible overhead, and has probability approximately s^2/N of finding a preimage, if a random input to H has probability $1/N$ of being a preimage.

In subsequent sections we convert the reversible circuit for H into a reversible circuit for a larger function H' using approximately $\sqrt{t}$ steps on t small parallel processors. H' is designed so that

- a random input to H' has probability approximately $t^{5/2}/N$ of being an H'-preimage and

- an H'-preimage produces a t-target H-preimage as desired.

Applying Grover's method to H', with $s \approx \sqrt{N/pt^{3/2}}$, uses overall $\sqrt{N/pt^{1/2}}$ steps on t small parallel processors, and has probability approximately t/p of finding a preimage. A machine with p/t parallel copies of Grover's method has high probability of finding a preimage and uses $\sqrt{N/pt^{1/2}}$ steps on p small parallel processors. Figure 9.1 shows our result highlighted and the known pre-quantum and post-quantum attacks considering several aspects such as: number of oracle calls, parallelism without communication cost and parallelism with communication cost.

9.3 — Reversible Iteration

As in the previous section, let H be a function from $\{0,1\}^b$ to $\{0,1\}^b$, where b is a nonnegative integer. Assume that we are given a reversible circuit for H using a ancillas and gate depth g (see, e.g., the circuit in [GLRS16] and Chapter 8). This section reviews the Bennett–Tompa technique [Ben89] to build a reversible circuit for H^n, where n is a positive integer, using $a + O(b \log_2 n)$ ancillas and gate depth $O(gn^{1+\epsilon})$. Here ϵ can be taken as close to 0 as desired, although the O constants depend on ϵ.

As a starting point, consider the following reversible circuit for H^2 using $a+b$ ancillas and depth $3g$:

time 0:	x	y	0	0
time 1:	x	y	$H(x)$	0
time 2:	x	$y \oplus H^2(x)$	$H(x)$	0
time 3:	x	$y \oplus H^2(x)$	0	0

Each step here is a reversible circuit for H, and in particular the last step adds $H(x)$ to $H(x)$, obtaining 0 (recall that $\oplus$ means xor).

More generally, if H uses a ancillas and depth g, and H' uses a' ancillas and depth g', then the following reversible circuit for $H' \circ H$ uses $\max(a, a'+b)$ ancillas and depth $2g+g'$:

time 0:	x	y	0	0
time 1:	x	y	$H(x)$	0
time 2:	x	$y \oplus H'(H(x))$	$H(x)$	0
time 3:	x	$y \oplus H'(H(x))$	0	0

Bennett now substitutes H^m and H^n for H and H' respectively, obtaining the following reversible circuit for H^{m+n} using $\max(a_m, a_n + b)$ ancillas and depth $2g_m + g_n$:

time 0:	x	y	0	0
time 1:	x	y	$H^m(x)$	0
time 2:	x	$y \oplus H^{m+n}(x)$	$H^m(x)$	0
time 3:	x	$y \oplus H^{m+n}(x)$	0	0

Bennett suggests taking $n = m$ or $n = m+1$, and then it is easy to prove by induction that $a_n = a + \lceil \log_2 n \rceil b$ and $g_n \leqslant 3^{\lceil \log_2 n \rceil} g \leqslant 3n^{\log_2 3} g$. For example, computing $H^{2^k}(x)$ uses $a + kb$ ancillas and depth $3^k g$.

More generally, with credit to Tompa, Bennett suggests a way to reduce the exponent $\log_2 3$ arbitrarily close to 1, at the expense of a constant factor in front of b. For example, one can start from the following reversible circuit for H^3 using $a + 2b$ ancillas and depth

5g:

time 0:	x	y	0	0	0
time 1:	x	y	$H(x)$	0	0
time 2:	x	y	$H(x)$	$H^2(x)$	0
time 3:	x	$y \oplus H^3(x)$	$H(x)$	$H^2(x)$	0
time 4:	x	$y \oplus H^3(x)$	$H(x)$	0	0
time 5:	x	$y \oplus H^3(x)$	0	0	0

Generalizing straightforwardly from H^3 to $H'' \circ H' \circ H$, and then replacing H, H', H'' with H^ℓ, H^m, H^n, produces a reversible circuit for $H^{\ell+m+n}$ using $\max(a_\ell + b\ , a_m + 2b, a_n + 2b)$ ancillas and depth $2g_\ell + 2g_m + g_n$. Splitting evenly between ℓ, m, n reduces $\log_2 3 \approx 1.58$ to $\log_3 5 \approx 1.46$. (An even split is not optimal: for a given ancilla budget one can afford to take a_ℓ larger than a_m and a_n. See [Kni95] for detailed optimizations along these lines.) By starting with H^4 instead of H^3 one reduces the exponent to $\log_4 7 \approx 1.40$, using, e.g., $a + 9b$ ancillas and depth $567g$ to compute H^{64}. By starting with H^8 one reduces the exponent to $\log_8 15 \approx 1.30$; etc.

9.4 — Reversible Distinguished Points

As above, let H be a function from $\{0,1\}^b$ to $\{0,1\}^b$, where b is a nonnegative integer; and assume that we are given an a-ancilla depth-g reversible circuit for H.

Fix $d \in \{0, 1, \ldots, b\}$. We say that $x \in \{0,1\}^b$ is distinguished if its first d bits are 0.

The rho method iterates H until finding a distinguished point or reaching a prespecified limit on the number of iterations, say n iterations. The resulting finite sequence $x, H(x), H^2(x), \ldots, H^m(x)$, either

- containing exactly one distinguished point $H^m(x)$ and having $m \leqslant n$ or
- containing zero distinguished points and having $m = n$,

is the chain for x, and its final entry $H^m(x)$ is the chain end for x.

This section explains a reversible circuit for the function that maps x to the chain end for x. This circuit has essentially the same cost as the Bennett–Tompa circuit from the previous section.

Define $H_d : \{0,1\}^b \to \{0,1\}^b$ as follows:

$$H_d(x) = \begin{cases} x & \text{if the first } d \text{ bits of } x \text{ are } 0 \\ H(x) & \text{otherwise.} \end{cases}$$

A reversible circuit for H_d is slightly more costly than a reversible circuit for H, since it needs an "OR" between the first d bits of x and a selection between x and $H(x)$.

If the chain for x is $x, H(x), H^2(x), \ldots, H^m(x)$ then the iterates

$$x, H_d(x), H_d^2(x), \ldots, H_d^m(x), H_d^{m+1}(x), \ldots, H_d^n(x)$$

are exactly $x, H(x), H^2(x), \ldots, H^m(x), H^m(x), \ldots, H^m(x)$. Hence the chain end for x, namely $H^m(x)$, is exactly $H_d^n(x)$. We compute H_d^n reversibly by substituting H_d for H in the previous section.

If x is chosen randomly and H behaves randomly then one expects each new H output to have chance $1/2^d$ of being distinguished. To have a reasonable chance that the chain end is distinguished, one should take n on the scale of 2^d: e.g., $n = 2^{d+1}$. If n is very large then chains will usually fall into loops before reaching distinguished points, but we will later take small n, roughly $\sqrt{t}$ for t-target preimage search.

9.4.1 – Reversible parallel distinguished points. Define b, H, a, g, d, n as before, and let t be a positive integer. This section explains a reversible circuit for the function that maps a vector $(x_1, \dots, x_t)$ of b-bit strings to the corresponding vector $(H_d^n(x_1), \dots, H_d^n(x_t))$ of chain ends.

This circuit is simply t parallel copies of the circuit from the previous section, where the ith copy handles x_i. The depth of the circuit is identical to the depth of the circuit in the previous section. The size of this circuit is t times larger than the size of the circuit in the previous section.

Communication in this circuit is only inside the parallel computations of H. There is no communication between the parallel circuits, and there is no dependence of communication costs upon t.

9.5 — Sorting on a Mesh Network

Define $S(c_1, c_2, \dots, c_t)$, where $c_1, c_2, \dots, c_t$ are b-bit strings, as $(d_1, d_2, \dots, d_t)$, where $d_1, d_2, \dots, d_t$ are the same as $c_1, c_2, \dots, c_t$ in lexicographic order.

This section presents a reversible computation of S using $O(t(b + (\log t)^2))$ ancillas and $O(t^{1/2}(\log t)^2)$ steps. Each step is a simple local operation on a two-dimensional mesh, repeated many times in parallel. We follow the general sorting strategy from [BBG+13] but choose different subroutines.

We start with odd-even mergesort [Bat68]. This algorithm is a sorting network: i.e., a sequence of comparators, where each comparator sorts two objects. Odd-even mergesort sorts t items using $O((\log t)^2)$ stages, where each stage involves $O(t)$ parallel comparators. For comparison, [BBG+13, Table 2] mentions bitonic sort, which is slower than odd-even mergesort, and AKS sort, which is asymptotically faster but slower for any reasonable size of t.

To make odd-even mergesort reversible, we record for each of the $O(t(\log t)^2)$ comparators whether the inputs were out of order, as in [BBG+13, Section 2.1]. This uses $O(t(\log t)^2)$ ancillas. The comparators themselves use $O(tb)$ ancillas.

The comparators in odd-even mergesort are not local when items are spread across a two-dimensional mesh. We fix this as in [BBG+13, Section 2.3]: before each stage, we permute the data so that the stage involves only local comparators. Each of these permutations is a constant determined by the structure of the sorting network; for odd-even mergesort each permutation is essentially a riffle shuffle.

The permutation strategy suggested in [BBG+13, Section 2.3] is to apply any sorting algorithm built from local operations. For a two-dimensional mesh, [BBG+13, Table 2] suggests “Bubble/Insertion sort”, but it is not at all clear which two-dimensional algorithm is meant here; the classic forms of bubble sort and insertion sort are not parallelizable. The same table also says that these are “sorting networks”, but most of the classic forms of bubble sort and insertion sort include conditional branches. We suggest using the Schnorr–Shamir algorithm [SS86], which has depth approximately $3\sqrt{t}$. It seems likely that an ad-hoc riffle algorithm would produce a better constant here.

9.6 — Multi-target Preimages

Fix images $y_1, \dots, y_t$. We build a reversible circuit that runs in a quantum computer and it performs the following function $f(\mathbf{x})$:

- Input a vector $(x_1, \dots, x_t)$.
- Compute, in parallel, the chain ends for $x_1, \dots, x_t$: i.e., $H_d^n(x_1), \dots, H_d^n(x_t)$.
- Precompute the chain ends for $y_1, \dots, y_t$.
- Sort the chain ends for $x_1, \dots, x_t$ and the chain ends for $y_1, \dots, y_t$.
- If there is a collision, say a collision between the chain end for x_i and the chain end for y_j: recompute the chain for x_i, checking each chain element to see whether it is a preimage for y_j.
- Output 0 if a preimage was found, otherwise 1.

As mentioned in Section 7.3, Grover's algorithm requires a search function; in our case this function is the circuit given by $f(\mathbf{x})$.

This circuit uses $O(a + b \log_2 n + tb + t(\log t)^2)$ ancillas. The chain computation has depth $O(gn^{1+\epsilon})$, and the sorting has depth $O(t^{1/2}(\log t)^2 \log b)$, where $O(\log b)$ accounts for the cost of a b-bit comparator.

If a chain for x_i ends with a distinguished point, and the chain includes a preimage (before this distinguished point) for y_j, then the chain for y_j will end with the same distinguished point. The recomputation will then find this preimage. The number of such chains is proportional to t (with a constant-factor loss for chains that end before a distinguished point), the number of elements in the chains is proportional to nt (with a constant factor reflecting the length of chains before distinguished points); the chance of a particular preimage being one of these elements is $1/N$; and there are t preimages, for an overall chance roughly nt^2/N.

We take $n \approx \sqrt{t}$, so the circuit uses $O(a + tb + t(\log t)^2)$ ancillas and has depth $O(gt^{1/2+\epsilon/2} + t^{1/2}(\log t)^2 \log b)$; one can also incorporate b, g, ϵ into the choice of n to better balance the two terms in this depth formula. The chance that the circuit finds a preimage is roughly $t^{5/2}/N$, as mentioned earlier. Finally, we apply p/t parallel copies of Grover's method to this circuit, each copy using approximately $\sqrt{N/pt^{3/2}}$ iterations, i.e., depth $O(\sqrt{N/pt^{1/2}}(gt^{\epsilon/2} + (\log t)^2 \log b))$, to reach a high probability of finding a t-target preimage.

Summary

Selected Constructive and Destructive Approaches to Post-Quantum Cryptography

The word *cryptography* comes from the Greek terms *krypto*, which means “hidden” or “secret” and *graphein*, which means “writing”. This means that cryptography can be defined as the art of hidden messages. In the past, cryptography had an important role in human history such as protecting Julius Caesar’s messages. Nowadays, cryptography is also used as a tool to protect data in payments, messaging in phones or securing access to a website. Modern cryptography relies on mathematical properties for being reliable and efficient. Those properties allow a user to reveal a message for the correct recipient and hide it from others. However, an attacker can try to decrypt the message without the key, the most naive way is to try to find the right key by testing all the keys possible. In a computer, the most common way to represent a key is by using a binary string and the search means finding the correct binary string of length k, i.e., an attacker needs to perform 2^k operations for finding the correct key. It is common to denote k as the security level of the cryptographic scheme. The situation changes if the attacker has a quantum computer. For example, as will be seen in this thesis, a search with a quantum computer can lower this value in half. In a more powerful attack, Shor’s algorithm is able to break cryptosystems based on the integer factoring problem or the discrete logarithm problem. In this thesis, I studied mathematical problems that are used in cryptosystems that are considered safe against quantum computers. A second area of work is the development of attacks using quantum algorithms, which contains the details of the design and the consequences of these attacks in the currently deployed cryptography.

Part I: Code-Based Cryptography. In the first part of the thesis, I present the background of linear codes and code-based cryptography. This includes a description of the McEliece cryptosystem, one of the oldest post-quantum cryptosystems. Second, I explain how it is possible to speed up the arithmetic of a special type of matrices, called dyadic matrices. More specifically, these matrices are used in a McEliece-based cryptosystem which uses a different linear code instead of Goppa codes, called Generalized Srivastava codes. In summary, the usage of this code allows smaller keys for McEliece, i.e., the keys have 19712 and 6400 bytes for public and private keys, respectively. Third, I show two attacks that are applied to code-based cryptography. The first attack is a side-channel attack. In this way, Chapter 5 shows that it is possible to apply a known side-channel attack to a relatively new implementation and gather enough information to recover the

encrypted message. After introducing the attack, the chapter shows that it is possible to find roots of a polynomial in a way that avoids the attack. Chapter 6 shows how a reaction attack is used against cryptosystems based on Low Rank Parity Check (LRPC) codes. A reaction attack consists of sending several random messages, waiting for the result and collecting the messages that the receiver was not able to decode. More specifically, we could exploit the structure of LRPC codes since a message cannot be decoded if certain conditions are not achieved.

Part II: Quantum Cryptanalysis. The second part of the thesis consists of quantum cryptanalysis. Before going into depth on quantum algorithms and the construction of quantum circuits, Chapter 7 discusses some aspects underlying the development of quantum computing such as the concepts of qubits, quantum registers and Grover's algorithm. Afterwards, Chapter 8 describes how to build the Advanced Encryption Standard (AES) algorithm in a quantum computer. While at the first glance it can seem easy to implement a well studied algorithm, quantum computation presents certain constraints such as reversibility. Therefore, it is not possible to reuse any AES construction, it is necessary to build an entire reversible circuit, Chapter 8 improves the number of operations and storage of the reversible circuit present in the literature. Finally, the thesis covers the construction of a quantum algorithm for finding preimages of a hash function and shows how to parallelize this algorithm. Furthermore, the algorithm developed shows that it is possible to run it with low communication costs between the quantum processors and without using any quantum memory. This is done by applying a new combination of distinguished points and Grover's algorithm.

Curriculum Vitae

Gustavo Souza Banegas was born on November 29, 1988, in Panambi, Brazil. In 2012, he completed his bachelor's degree in Computer Science at Universidade Federal de Santa Catarina. In the same year, he started his master at Universidade Federal de Santa Catarina which he completed in October of 2015 under the supervision of Ricardo Custódio and Daniel Panario.

At the end of 2015, he started his PhD project in the Coding theory and Cryptography group at the Eindhoven University of Technology under the supervision of Tanja Lange and Daniel J. Bernstein. The project was funded by the European Union's Horizon 2020 research and innovation program under the Marie Skłodowska-Curie grant agreement No. 643161. In 2016, he organized the first Quantum Research Retreat in Eindhoven and in 2018 he organized a second Quantum Research Retreat in Tenerife. During his PhD, he gained experience in industry with two internships: one at Riscure in Delft and another at CryptoExperts in Paris.

Bibliography

[ABB$^+$] Nicolas Aragon, Paulo S. L. M. Barreto, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Shay Gueron, Tim Güneysu, Carlos Aguilar Melchor, Rafael Misoczki, Edoardo Persichetti, Jean-Pierre Tillich, Valentin Vasseur, and Gilles Zémor. BIKE - Bit Flipping Key Encapsulation. NIST Post-Quantum Cryptography Project: First and Second Rounds Candidate Algorithms. URL: https://bikesuite.org.

[ABG$^+$17a] Nicolas Aragon, Jean-Christophe Blazy, Olivier Deneuville, Philippe Gaborit, Adrien Hauteville, Olivier Ruatta, Jean-Pierre Tillich, and Gilles Zémor. LAKE, December 2017. NIST Post-Quantum Cryptography Project: First Round Candidate Algorithms. URL: https://pqc-rollo.org/.

[ABG$^+$17b] Nicolas Aragon, Jean-Christophe Blazy, Olivier Deneuville, Philippe Gaborit, Adrien Hauteville, Olivier Ruatta, Jean-Pierre Tillich, and Gilles Zémor. LOCKER, December 2017. NIST Post-Quantum Cryptography Project: First Round Candidate Algorithms. URL: https://pqc-rollo.org/index.html.

[ABG$^+$19] Nicolas Aragon, Olivier Blazy, Philippe Gaborit, Adrien Hauteville, and Gilles Zémor. Durandal: A rank metric based signature scheme. In Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19-23, 2019, Proceedings, Part III*, volume 11478 of *Lecture Notes in Computer Science*, pages 728–758. Springer, 2019. doi:10.1007/978-3-030-17659-4_25.

[ADPS16] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange – A new hope. In Thorsten Holz and Stefan Savage, editors, *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, pages 327–343. USENIX Association, 2016. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/alkim.

[AG04] Scott Aaronson and Daniel Gottesman. Improved simulation of stabilizer circuits. *Phys. Rev. A*, 70:052328, Nov 2004. doi:10.1103/PhysRevA.70.052328.

[Al-01] Abdolrahman Kh. Al-Jabri. A statistical decoding algorithm for general linear block codes. In Bahram Honary, editor, *Cryptography and Coding, 8th IMA In-*

ternational Conference, Cirencester, UK, December 17-19, 2001, Proceedings, volume 2260 of *Lecture Notes in Computer Science*, pages 1–8. Springer, 2001. doi:10.1007/3-540-45325-3_1.

[Amy13] Matthew Amy. *Algorithms for the Optimization of Quantum Circuits*. PhD thesis, University of Waterloo, 2013. URL: http://hdl.handle.net/10012/7818.

[ASAM18] Mishal Almazrooie, Azman Samsudin, Rosni Abdullah, and Kussay N. Mutter. Quantum reversible circuit of AES-128. *Quantum Information Processing*, 17(5):112, Mar 2018. doi:10.1007/s11128-018-1864-3.

[Bar94] Alexander Barg. Some new NP-complete coding problems. *Probl. Peredachi Inf.*, 30:23–28, 1994. (in Russian).

[Bar97] Alexander Barg. Complexity issues in coding theory. *Electronic Colloquium on Computational Complexity (ECCC)*, 4(46), 1997.

[Bat68] Kenneth E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), pages 307–314, New York, NY, USA, 1968. ACM. doi:10.1145/1468075.1468121.

[BB17] **Gustavo Banegas** and Daniel J. Bernstein. Low-communication parallel quantum multi-target preimage search. In Carlisle Adams and Jan Camenisch, editors, *Selected Areas in Cryptography - SAC 2017 - 24th International Conference, Ottawa, ON, Canada, August 16-18, 2017, Revised Selected Papers*, volume 10719 of *Lecture Notes in Computer Science*, pages 325–335. Springer, 2017. doi:10.1007/978-3-319-72565-9_16.

[BBB+] Magali Bardet, Élise Barelli, Olivier Blazy, Rodolfo Canto Torres, Alain Couvreur, Philippe Gaborit, Ayoub Otmani, Nicolas Sendrier, and Jean-Pierre Tillich. BIGQUAKE - BInary Goppa QUAsi-cyclic Key Encapsulation. NIST Post-Quantum Cryptography Project: First Round Candidate Algorithms. URL: https://bigquake.inria.fr/.

[BBB+18] **Gustavo Banegas**, Paulo S. L. M. Barreto, Brice Odilon Boidje, Pierre-Louis Cayrel, Gilbert Ndollane Dione, Kris Gaj, Cheikh Thiécoumba Gueye, Richard Haeussler, Jean Belo Klamti, Ousmane Ndiaye, Duc Tri Nguyen, Edoardo Persichetti, and Jefferson E. Ricardini. DAGS: key encapsulation using dyadic GS codes. *J. Mathematical Cryptology*, 12(4):221–239, 2018. doi:10.1515/jmc-2018-0027.

[BBB+19] **Gustavo Banegas**, Paulo S. L. M. Barreto, Brice Odilon Boidje, Pierre-Louis Cayrel, Gilbert Ndollane Dione, Kris Gaj, Cheikh Thiécoumba Gueye, Richard Haeussler, Jean Belo Klamti, Ousmane Ndiaye, Duc Tri Nguyen, Edoardo Persichetti, and Jefferson E. Ricardini. DAGS reloaded: Revisiting dyadic key encapsulation. In Baldi et al. [BPS19], pages 69–85. doi:10.1007/978-3-030-25922-8_4.

[BBCO19] Magali Bardet, Manon Bertin, Alain Couvreur, and Ayoub Otmani. Practical algebraic attack on DAGS. In Baldi et al. [BPS19], pages 86–101. doi:10.1007/978-3-030-25922-8_5.

[BBG$^+$13] Robert Beals, Stephen Brierley, Oliver Gray, Aram W. Harrow, Samuel Kutin, Noah Linden, Dan Shepherd, and Mark Stather. Efficient distributed quantum computing. *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 469(2153), 2013. doi:10.1098/rspa.2012.0686.

[BBPS18] **Gustavo Banegas**, Paulo S. L. M. Barreto, Edoardo Persichetti, and Paolo Santini. Designing efficient dyadic operations for cryptographic applications. *Mathcrypt, to appear*, 2018. https://eprint.iacr.org/2018/650.

[BC18] Élise Barelli and Alain Couvreur. An efficient structural attack on NIST submission DAGS. In Thomas Peyrin and Steven D. Galbraith, editors, *Advances in Cryptology – ASIACRYPT 2018 – 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2-6, 2018, Proceedings, Part I*, volume 11272 of *Lecture Notes in Computer Science*, pages 93–118. Springer, 2018. doi:10.1007/978-3-030-03326-2_4.

[BCD$^+$16] Joppe W. Bos, Craig Costello, Léo Ducas, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Ananth Raghunathan, and Douglas Stebila. Frodo: Take off the ring! practical, quantum-secure key exchange from LWE. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 1006–1018. ACM, 2016. doi:10.1145/2976749.2978425.

[BCDR17] Dominic Bucerzan, Pierre-Louis Cayrel, Vlad Dragoi, and Tania Richmond. Improved timing attacks against the secret permutation in the McEliece PKC. *International Journal of Computers Communications & Control*, 12(1):7–25, 2017.

[BCG06] Marco Baldi, Franco Chiaraluce, and Roberto Garello. On the usage of quasi-cyclic low-density parity-check codes in the McEliece cryptosystem. In *Proceedings of the First International Conference on Communication and Electronics (ICEE'06)*, pages 305–310, October 2006.

[BCGO09] Thierry P. Berger, Pierre-Louis Cayrel, Philippe Gaborit, and Ayoub Otmani. Reducing key length of the McEliece cryptosystem. In Bart Preneel, editor, *Progress in Cryptology - AFRICACRYPT 2009, Second International Conference on Cryptology in Africa, Gammarth, Tunisia, June 21-25, 2009. Proceedings*, volume 5580 of *Lecture Notes in Computer Science*, pages 77–97. Springer, 2009. doi:10.1007/978-3-642-02384-2_6.

[BCL$^+$] Daniel J. Bernstein, Tung Chou, Tanja Lange, Ingo von Maurich, Rafael Misoczki, Ruben Niederhagen, Edoardo Persichetti, Christiane Peters, Peter Schwabe, Nicolas Sendrier, Jakub Szefer, and Wen Wang. Classic McEliece. NIST Post-Quantum Cryptography Project: First and Second Rounds Candidate Algorithms. URL: https://classic.mceliece.org/.

[BCNS15] Joppe W. Bos, Craig Costello, Michael Naehrig, and Douglas Stebila. Post-quantum key exchange for the TLS protocol from the ring learning with errors problem. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 553–570. IEEE Computer Society, 2015. doi:10.1109/SP.2015.40.

[BCP18] **Gustavo Banegas**, Ricardo Custódio, and Daniel Panario. A new class of irreducible pentanomials for polynomial-based multipliers in binary fields. *Journal of Cryptographic Engineering*, Nov 2018. doi:10.1007/s13389-018-0197-6.

[BCS13] Daniel J. Bernstein, Tung Chou, and Peter Schwabe. McBits: Fast constant-time code-based cryptography. In Guido Bertoni and Jean-Sébastien Coron, editors, *Cryptographic Hardware and Embedded Systems - CHES 2013 - 15th International Workshop, Santa Barbara, CA, USA, August 20-23, 2013. Proceedings*, volume 8086 of *Lecture Notes in Computer Science*, pages 250–272. Springer, 2013. doi:10.1007/978-3-642-40349-1_15.

[BD08] Johannes A. Buchmann and Jintai Ding, editors. *Post-Quantum Cryptography, Second International Workshop, PQCrypto 2008, Cincinnati, OH, USA, October 17-19, 2008, Proceedings*, volume 5299 of *Lecture Notes in Computer Science*. Springer, 2008. doi:10.1007/978-3-540-88403-3.

[Ben89] Charles H. Bennett. Time/space trade-offs for reversible computation. *SIAM J. Comput.*, 18(4):766–776, 1989. doi:10.1137/0218053.

[Ber70] Elwyn R. Berlekamp. Factoring polynomials over large finite fields. *Mathematics of computation*, 24(111):713–735, 1970.

[Ber05] Daniel J. Bernstein. Cache-timing attacks on AES, 2005. URL: https://cr.yp.to/antiforgery/cachetiming-20050414.pdf.

[Ber09] Daniel J. Bernstein. Cost analysis of hash collisions: Will quantum computers make sharcs obsolete? *SHARCS'09 Special-purpose Hardware for Attacking Cryptographic Systems*, page 105, 2009.

[Ber10] Daniel J. Bernstein. Grover vs. McEliece. In Sendrier [Sen10], pages 73–80. doi:10.1007/978-3-642-12929-2_6.

[Ber15] Elwyn R. Berlekamp. *Algebraic Coding Theory*. WORLD SCIENTIFIC, 2015. doi:10.1142/9407.

[BFP13] Luk Bettale, Jean-Charles Faugère, and Ludovic Perret. Cryptanalysis of HFE, multi-HFE and variants for odd and even characteristic. *Des. Codes Cryptography*, 69(1):1–52, 2013. doi:10.1007/s10623-012-9617-2.

[BFS99] Jonathan F. Buss, Gudmund S. Frandsen, and Jeffrey O. Shallit. The computational complexity of some problems of linear algebra. *Journal of Computer and System Sciences*, 58(3):572–596, 1999.

[BGG$^+$17] Paulo S. L. M. Barreto, Shay Gueron, Tim Güneysu, Rafael Misoczki, Edoardo Persichetti, Nicolas Sendrier, and Jean-Pierre Tillich. CAKE: code-based algorithm for key encapsulation. In Máire O'Neill, editor, *Cryptography and Coding - 16th IMA International Conference, IMACC 2017, Oxford, UK, December 12-14, 2017, Proceedings*, volume 10655 of *Lecture Notes in Computer Science*, pages 207–226. Springer, 2017. doi:10.1007/978-3-319-71045-7_11.

[BHLY16] Leon Groot Bruinderink, Andreas Hülsing, Tanja Lange, and Yuval Yarom. Flush, Gauss, and Reload - A cache attack on the BLISS Lattice-based signature scheme. In Benedikt Gierlichs and Axel Y. Poschmann, editors, *Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*, volume 9813 of *Lecture Notes in Computer Science*, pages 323–345. Springer, 2016. doi:10.1007/978-3-662-53140-2_16.

[BHT98] Gilles Brassard, Peter Høyer, and Alain Tapp. Quantum cryptanalysis of hash and claw-free functions. In Claudio L. Lucchesi and Arnaldo V. Moura, editors, *LATIN '98: Theoretical Informatics, Third Latin American Symposium, Campinas, Brazil, April, 20-24, 1998, Proceedings*, volume 1380 of *Lecture Notes in Computer Science*, pages 163–169. Springer, 1998. doi:10.1007/BFb0054319.

[Bla05] Paul E. Black. Fisher-Yates shuffle. *Dictionary of algorithms and data structures*, 19, 2005.

[BLM11] Paulo S. L. M. Barreto, Richard Lindner, and Rafael Misoczki. Monoidic codes in cryptography. In Bo-Yin Yang, editor, *Post-Quantum Cryptography - 4th International Workshop, PQCrypto 2011, Taipei, Taiwan, November 29 - December 2, 2011. Proceedings*, volume 7071 of *Lecture Notes in Computer Science*, pages 179–199. Springer, 2011. doi:10.1007/978-3-642-25405-5_12.

[BLP10] Daniel J. Bernstein, Tanja Lange, and Christiane Peters. Wild McEliece. In Alex Biryukov, Guang Gong, and Douglas R. Stinson, editors, *Selected Areas in Cryptography - 17th International Workshop, SAC 2010, Waterloo, Ontario, Canada, August 12-13, 2010, Revised Selected Papers*, volume 6544 of *Lecture Notes in Computer Science*, pages 143–158. Springer, 2010. doi:10.1007/978-3-642-19574-7.

[BMvT78] Elwyn R. Berlekamp, Robert J. McEliece, and Henk C. A. van Tilborg. On the inherent intractability of certain coding problems (corresp.). *Information Theory, IEEE Transactions on*, 24(3):384 – 386, may 1978. doi:10.1109/TIT.1978.1055873.

[Boy01] Colin Boyd, editor. *Advances in Cryptology – ASIACRYPT 2001, 7th International Conference on the Theory and Application of Cryptology and Information Security, Gold Coast, Australia, December 9-13, 2001, Proceedings*, volume 2248 of *Lecture Notes in Computer Science*. Springer, 2001. doi:10.1007/3-540-45682-1.

[BP18] Daniel J. Bernstein and Edoardo Persichetti. Towards KEM unification. Cryptology ePrint Archive, Report 2018/526, 2018. https://eprint.iacr.org/2018/526.

[BPS19] Marco Baldi, Edoardo Persichetti, and Paolo Santini, editors. *Code-Based Cryptography – 7th International Workshop, CBC 2019, Darmstadt, Germany, May 18-19, 2019, Revised Selected Papers*, volume 11666 of *Lecture Notes in Computer Science*. Springer, 2019. doi:10.1007/978-3-030-25922-8.

[BS08] Bhaskar Biswas and Nicolas Sendrier. McEliece cryptosystem implementation: Theory and practice. In Buchmann and Ding [BD08], pages 47–62. doi:10.1007/978-3-540-88403-3_4.

[CBSG17] Andrew W. Cross, Lev S. Bishop, John A. Smolin, and Jay M. Gambetta. Open quantum assembly language. *arXiv preprint arXiv:1707.03429*, 2017.

[CFS01] Nicolas Courtois, Matthieu Finiasz, and Nicolas Sendrier. How to achieve a McEliece-based digital signature scheme. In Boyd [Boy01], pages 157–174. doi:10.1007/3-540-45682-1.

[Cho17] Tung Chou. McBits revisited. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, volume 10529 of *Lecture Notes in Computer Science*, pages 213–231. Springer, 2017. doi:10.1007/978-3-319-66787-4_11.

[CHP12] Pierre-Louis Cayrel, Gerhard Hoffmann, and Edoardo Persichetti. Efficient implementation of a CCA2-secure variant of McEliece using generalized Srivastava codes. In Marc Fischlin, Johannes A. Buchmann, and Mark Manulis, editors, *Public Key Cryptography - PKC 2012 - 15th International Conference on Practice and Theory in Public Key Cryptography, Darmstadt, Germany, May 21-23, 2012. Proceedings*, volume 7293 of *Lecture Notes in Computer Science*, pages 138–155. Springer, 2012. doi:10.1007/978-3-642-30057-8_9.

[CL04] Daniel J. Costello and Shu Lin. *Error Control Coding: Fundamentals and Applications*. Pearson, 2nd edition, 2004.

[Cou01] Nicolas Courtois. Efficient zero-knowledge authentication based on a linear algebra problem MinRank. In Boyd [Boy01], pages 402–421. doi:10.1007/3-540-45682-1_24.

[CR88] Benny Chor and Ronald L. Rivest. A knapsack-type public key cryptosystem based on arithmetic in finite fields. *IEEE Trans. Information Theory*, 34(5):901–909, 1988. doi:10.1109/18.21214.

[DGZ17] Jean-Christophe Deneuville, Philippe Gaborit, and Gilles Zémor. Ouroboros: A simple, secure and efficient key exchange protocol based on coding theory. In Lange and Takagi [LT17], pages 18–34. doi:10.1007/978-3-319-59879-6_2.

[DHF$^+$18] Jean-Luc Danger, Youssef El Housni, Adrien Facon, Cheikh Thiecoumba Gueye, Sylvain Guilley, Sylvie Herbel, Ousmane Ndiaye, Edoardo Persichetti, and Alexander Schaub. On the performance and security of multiplication in $GF(2^n)$. *Cryptography*, 2(3):25, 2018. doi:10.3390/cryptography2030025.

[Div15] NIST Computer Security Division. SHA-3 standard: Permutation-based hash and extendable-output functions. FIPS Publication 202, National Institute of Standards and Technology, U.S. Department of Commerce, Aug 2015. doi:10.6028/NIST.FIPS.202.

[DPP16] James H. Davenport, Christophe Petit, and Benjamin Pring. A generalised successive resultants algorithm. In Sylvain Duquesne and Svetla Petkova-Nikova, editors, *Arithmetic of Finite Fields - 6th International Workshop, WAIFI 2016, Ghent, Belgium, July 13-15, 2016, Revised Selected Papers*, volume 10064 of *Lecture Notes in Computer Science*, pages 105–124, 2016. doi:10.1007/978-3-319-55227-9_9.

[DR13] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES-the advanced encryption standard*. Springer Science & Business Media, 2013.

[FA76] Bernard J. Fino and V. Ralph Algazi. Unified matrix treatment of the fast Walsh-Hadamard transform. *IEEE Trans. Computers*, 25(11):1142–1146, 1976. doi:10.1109/TC.1976.1674569.

[FGO$^+$13] Jean-Charles Faugère, Valérie Gauthier-Umaña, Ayoub Otmani, Ludovic Perret, and Jean-Pierre Tillich. A distinguisher for high-rate McEliece cryptosystems. *IEEE Trans. Information Theory*, 59(10):6830–6844, 2013. doi:10.1109/TIT.2013.2272036.

[FGP$^+$15] Jean-Charles Faugère, Danilo Gligoroski, Ludovic Perret, Simona Samardjiska, and Enrico Thomae. A polynomial-time key-recovery attack on MQQ cryptosystems. In Jonathan Katz, editor, *Public-Key Cryptography - PKC 2015 - 18th IACR International Conference on Practice and Theory in Public-Key Cryptography, Gaithersburg, MD, USA, March 30 - April 1, 2015, Proceedings*, volume 9020 of *Lecture Notes in Computer Science*, pages 150–174. Springer, 2015. doi:10.1007/978-3-662-46447-2_7.

[FOP$^+$16] Jean-Charles Faugère, Ayoub Otmani, Ludovic Perret, Frédéric de Portzamparc, and Jean-Pierre Tillich. Structural cryptanalysis of McEliece schemes with compact keys. *Des. Codes Cryptography*, 79(1):87–112, 2016. doi:10.1007/s10623-015-0036-z.

[FOPT10] 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, 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Monaco / French Riviera, May 30 - June 3, 2010. Proceedings*, volume 6110 of *Lecture Notes in Computer Science*, pages 279–298. Springer, 2010. doi:10.1007/978-3-642-13190-5_14.

[FT02] Sergei V. Fedorenko and Peter V. Trifonov. Finding roots of polynomials over finite fields. *IEEE Transactions on communications*, 50(11):1709–1711, 2002.

[Gab85] Ernest Mukhamedovich Gabidulin. Theory of codes with maximum rank distance. *Problemy Peredachi Informatsii*, 21(1):3–16, 1985.

[Gal63] Robert G. Gallager. *Low-Density Parity-Check Codes*. PhD thesis, M.I.T., 1963.

[GC00] Louis Goubin and Nicolas Courtois. Cryptanalysis of the TTM cryptosystem. In Tatsuaki Okamoto, editor, *Advances in Cryptology – ASIACRYPT 2000, 6th International Conference on the Theory and Application of Cryptology and Information Security, Kyoto, Japan, December 3-7, 2000, Proceedings*, volume 1976 of *Lecture Notes in Computer Science*, pages 44–57. Springer, 2000. doi:10.1007/3-540-44448-3_4.

[GJS16] 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 – 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I*, volume 10031 of *Lecture Notes in Computer Science*, pages 789–815, 2016. doi:10.1007/978-3-662-53887-6_29.

[GKK$^+$17] Lucky Galvez, Jon-Lark Kim, Myeong Jae Kim, Young-Sik Kim, and Nari Lee. McNie: Compact Mceliece-Niederreiter Cryptosystem, December 2017. NIST Post-Quantum Cryptography Project: First Round Candidate Algorithms. URL: https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/round-1/submissions/McNie.zip.

[GLRS16] Markus Grassl, Brandon Langenberg, Martin Roetteler, and Rainer Steinwandt. Applying Grover's algorithm to AES: quantum resource estimates. In Tsuyoshi Takagi, editor, *Post-Quantum Cryptography – 7th International Workshop, PQCrypto 2016, Fukuoka, Japan, February 24-26, 2016, Proceedings*, volume 9606 of *Lecture Notes in Computer Science*, pages 29–43. Springer, 2016. doi:10.1007/978-3-319-29360-8_3.

[GMRZ13] Philippe Gaborit, Gaétan Murat, Olivier Ruatta, and Gilles Zemor. Low Rank Parity Check codes and their application to cryptography. In Lilya Budaghyan, Tor Helleseth, and Matthew G. Parker, editors, *The International Workshop on Coding and Cryptography (WCC 13)*, Bergen, Norway, April 2013. ISBN 978-82-308-2269-2. URL: https://hal.archives-ouvertes.fr/hal-00913719.

[GR03] Lov Grover and Terry Rudolph. How significant are the known collision and element distinctness quantum algorithms? *arXiv preprint quant-ph/0309123*, 2003.

[Gra06] Robert M. Gray. Toeplitz and circulant matrices: A review. *Foundations and Trends® in Communications and Information Theory*, 2(3):155–239, 2006. doi:10.1561/0100000006.

[Gro96] Lov K. Grover. A fast quantum mechanical algorithm for database search. In Gary L. Miller, editor, *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*, pages 212–219. ACM, 1996. doi:10.1145/237814.237866.

[GRSZ14] Philippe Gaborit, Olivier Ruatta, Julien Schrek, and Gilles Zémor. New results for rank-based cryptography. In David Pointcheval and Damien Vergnaud, editors, *Progress in Cryptology, AFRICACRYPT 2014, 7th International Conference on Cryptology in Africa, Marrakesh, Morocco, May 28-30, 2014. Proceedings*, volume 8469 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2014. doi:10.1007/978-3-319-06734-6_1.

[Gul73] Mohammed N. Gulamhusein. Simple matrix-theory proof of the discrete dyadic convolution theorem. *Electronics Letters*, 9(10):238–239, 1973.

[Hel72] Hermann Helgert. Srivastava codes. *IEEE Transactions on Information Theory*, 18(2):292–297, March 1972. doi:10.1109/TIT.1972.1054760.

[HHK17] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. A modular analysis of the Fujisaki-Okamoto transformation. In Yael Kalai and Leonid Reyzin, editors, *Theory of Cryptography - 15th International Conference, TCC 2017, Baltimore, MD, USA, November 12-15, 2017, Proceedings, Part I*, volume 10677 of *Lecture Notes in Computer Science*, pages 341–371. Springer, 2017. doi:10.1007/978-3-319-70500-2_12.

[HHL09] Aram W. Harrow, Avinatan Hassidim, and Seth Lloyd. Quantum algorithm for linear systems of equations. *Physical review letters*, 103(15):150502, 2009.

[HRS16] Andreas Hülsing, Joost Rijneveld, and Fang Song. Mitigating multi-target attacks in hash-based signatures. In Chen-Mou Cheng, Kai-Min Chung, Giuseppe Persiano, and Bo-Yin Yang, editors, *Public-Key Cryptography – PKC 2016 - 19th IACR International Conference on Practice and Theory in Public-Key Cryptography, Taipei, Taiwan, March 6-9, 2016, Proceedings, Part I*, volume 9614 of *Lecture Notes in Computer Science*, pages 387–416. Springer, 2016. doi:10.1007/978-3-662-49384-7_15.

[HS13] Yann Hamdaoui and Nicolas Sendrier. A non asymptotic analysis of information set decoding. Cryptology ePrint Archive, Report 2013/162, 2013. http://eprint.iacr.org/2013/162.

[KHJ18] Panjin Kim, Daewan Han, and Kyung Chul Jeong. Time–space complexity of quantum search algorithms in symmetric cryptanalysis: applying to aes and sha-2. *Quantum Information Processing*, 17(12):339, Oct 2018. doi:10.1007/s11128-018-2107-3.

[KKG$^+$18] Jon-Lark Kim, Young-Sik Kim, Lucky Galvez, Myeong Jae Kim, and Nari Lee. McNie: a code-based public-key cryptosystem. *arXiv preprint arXiv:1812.05008*, 2018. URL: https://arxiv.org/abs/1812.05008.

[Kni95] Emanuel Knill. An analysis of Bennett's pebble game. *CoRR*, abs/math/9508218, 1995. URL: http://arxiv.org/abs/math/9508218.

[KO62] Anatoly Karatsuba and Yuri Ofman. Multiplication of multidigit numbers by automata. *Soviet Physics Doklady*, 7:595, 12 1962.

[KS99] Aviad Kipnis and Adi Shamir. Cryptanalysis of the HFE public key cryptosystem by relinearization. In Wiener [Wie99], pages 19–30. doi:10.1007/3-540-48405-1_2.

[KT17] Ghazal Kachigar and Jean-Pierre Tillich. Quantum information set decoding algorithms. In Lange and Takagi [LT17], pages 69–89. doi:10.1007/978-3-319-59879-6_5.

[LSP98] Hoi-Kwong Lo, Tim Spiller, and Sandu Popescu. *Introduction to quantum computation and information*. World Scientific, 1998.

[LT17] Tanja Lange and Tsuyoshi Takagi, editors. *Post-Quantum Cryptography - 8th International Workshop, PQCrypto 2017, Utrecht, The Netherlands, June 26-28, 2017, Proceedings*, volume 10346 of *Lecture Notes in Computer Science*. Springer, 2017. doi:10.1007/978-3-319-59879-6.

[LT18] Terry Shue Chien Lau and Chik How Tan. Key recovery attack on McNie based on low rank parity check codes and its reparation. In Atsuo Inomata and Kan Yasuda, editors, *Advances in Information and Computer Security - 13th International Workshop on Security, IWSEC 2018, Sendai, Japan, September 3-5, 2018, Proceedings*, volume 11049 of *Lecture Notes in Computer Science*, pages 19–34. Springer, 2018. doi:10.1007/978-3-319-97916-8_2.

[MAAB$^+$17] Carlos Melchor Aguilar, Nicolas Aragon, Slim Bettaieb, Loïc Bidoux, Jean-Christophe Blazy, Olivier Deneuville, Philippe Gaborit, Adrien Hauteville, and Gilles Zémor. Ouroboros-R, December 2017. NIST Post-Quantum Cryptography Project: First Round Candidate Algorithms. URL: https://pqc-ouroborosr.org/.

[MAAB$^+$19] Carlos Melchor Aguilar, Nicolas Aragon, Magali Bardet, Slim Bettaieb, Loïc Bidoux, Jean-Christophe Blazy, Olivier Deneuville, Philippe Gaborit, Adrien Hauteville, Ayoub Otmani, Olivier Ruatta, Jean-Pierre Tillich, and Gilles Zémor. Rollo - Rank-Ouroboros, LAKE and LOCKER, January 2019. NIST Post-Quantum Cryptography Project: Second Round Candidate Algorithms. URL: https://pqc-rollo.org/index.html.

[MB09] Rafael Misoczki and Paulo S. L. M. Barreto. Compact McEliece keys from Goppa codes. In Michael J. Jacobson Jr., Vincent Rijmen, and Reihaneh Safavi-Naini, editors, *Selected Areas in Cryptography, 16th Annual International Workshop, SAC 2009, Calgary, Alberta, Canada, August 13-14, 2009, Revised Selected Papers*, volume 5867 of *Lecture Notes in Computer Science*, pages 376–392. Springer, 2009. doi:10.1007/978-3-642-05445-7_24.

[MBC19] Douglas Marcelino Beppler Martins, **Gustavo Banegas**, and Ricardo Felipe Custódio. Don't forget your roots: Constant-time root finding over $\mathbb{F}_{2^m}$. In Schwabe and Thériault [ST19], pages 109–129. doi:10.1007/978-3-030-30530-7_6.

[McE78] Robert J. McEliece. A public-key cryptosystem based on algebraic coding theory. *Deep Space Network Progress Report*, 44:114–116, January 1978.

[MS77] Florence Jessie MacWilliams and Neil James Alexander Sloane. *The theory of error-correcting codes*, volume 16. Elsevier, 1977.

[MSSS18] Michele Mosca, Nicolas Sendrier, Rainer Steinwandt, and Krysta Svore. Quantum Cryptanalysis (Dagstuhl Seminar 17401). *Dagstuhl Reports*, 7(10):1–13, 2018. doi:10.4230/DagRep.7.10.1.

[MTSB13] Rafael Misoczki, Jean-Pierre Tillich, Nicolas Sendrier, and Paulo L.S.M. Barreto. MDPC-McEliece: New McEliece variants from moderate density parity-check codes. In *IEEE International Symposium on Information Theory – ISIT'2013*, pages 2069–2073, Istambul, Turkey, 2013. IEEE.

[Nat01] National Institute of Standards and Technology. *Advanced Encryption Standard (AES)*. pub-NIST, November 2001. Supersedes FIPS PUB 180 1993 May 11. URL: https://www.nist.gov/publications/advanced-encryption-standard-aes.

[Nie86] Harald Niederreiter. Knapsack-type cryptosystems and algebraic coding theory. *Prob. Control and Inf. Theory*, 15(2):159–166, 1986.

[Nie11] Ruben Niebuhr. Statistical decoding of codes over $\mathbb{F}_q$. In Bo-Yin Yang, editor, *Post-Quantum Cryptography: 4th International Workshop, PQCrypto 2011, Taipei, Taiwan, November 29 – December 2, 2011. Proceedings*, pages 217–227, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. doi:10.1007/978-3-642-25405-5_14.

[NIKM08] Ryo Nojima, Hideki Imai, Kazukuni Kobara, and Kirill Morozov. Semantic security for the McEliece cryptosystem without random oracles. *Des. Codes Cryptography*, 49(1-3):289–305, 2008. doi:10.1007/s10623-008-9175-9.

[NIS16] NIST. Submission requirements and evaluation criteria for the post-quantum cryptography standardization process, 2016. http://csrc.nist.gov/groups/ST/post-quantum-crypto/documents/call-for-proposals-final-dec-2016.pdf.

[NIS17] NIST. Post-quantum cryptography – round 1 submissions, 2017. URL: https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-1-Submissions.

[NJS19] Alexander Nilsson, Thomas Johansson, and Paul Stankovski. Error amplification in code-based cryptography. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(1):238–258, 2019. doi:10.13154/tches.v2019.i1.238-258.

[NPC$^+$17] Robert Niebuhr, Edoardo Persichetti, Pierre-Louis Cayrel, Stanislav Bulygin, and Johannes A. Buchmann. On lower bounds for information set decoding over $\mathbb{F}_q$ and on the effect of partial knowledge. *IJICoT*, 4(1):47–78, 2017. doi:10.1504/IJICOT.2017.10002266.

[Orw83] G. Orwell. *1984*. HMH Books, 1983.

[Pat75] Nicholas Patterson. The algebraic decoding of Goppa codes. *IEEE Transactions on Information Theory*, 21(2):203–207, 1975.

[Per12a] Edoardo Persichetti. Compact McEliece keys based on quasi-dyadic Srivastava codes. *Journal of Mathematical Cryptology*, 6(2):149–169, 2012.

[Per12b] Edoardo Persichetti. *Improving the efficiency of code-based cryptography*. PhD thesis, The University of Auckland, 2012. URL: http://hdl.handle.net/2292/19803.

[Per13] Edoardo Persichetti. Secure and anonymous hybrid encryption from coding theory. In Philippe Gaborit, editor, *Post-Quantum Cryptography: 5th International Workshop, PQCrypto 2013, Limoges, France, June 4-7, 2013. Proceedings*, pages 174–187, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. doi:10.1007/978-3-642-38616-9_12.

[Pet] Christiane Peters. Scripts. URL: https://cbcrypto.org/publications/scripts/.

[Pet10] Christiane Peters. Information-set decoding for linear codes over $\mathbb{F}_q$. In Sendrier [Sen10], pages 81–94. doi:10.1007/978-3-642-12929-2.

[Pet11] Christiane Peters. *Curves, codes, and cryptography*. PhD thesis, Technische Universiteit Eindhoven, 2011. doi:10.6100/IR711052.

[Pet14] Christophe Petit. Finding roots in GF(p^n) with the successive resultant algorithm. *IACR Cryptology ePrint Archive*, 2014:506, 2014.

[Pra62] E. Prange. The use of information sets in decoding cyclic codes. *IRE Transactions*, IT-8:S5–S9, 1962.

[PS17] Lukas Polok and Pavel Smrz. Pivoting strategy for fast LU decomposition of sparse block matrices. In *Proceedings of the 25th High Performance Computing Symposium*, HPC '17, pages 14:1–14:12, San Diego, CA, USA, 2017. Society for Computer Simulation International. URL: http://dl.acm.org/citation.cfm?id=3108096.3108110.

[RSA78] R. L. Rivest, A. Shamir, and L. M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[Th19] The Sage Developers. Sagemath, the Sage Mathematics Software System (Version 8.7), 2019. URL: https://www.sagemath.org.

[Sav97] Carla Savage. A survey of combinatorial Gray codes. *SIAM review*, 39(4):605–629, 1997.

[SBCB19] Paolo Santini, Massimo Battaglioni, Franco Chiaraluce, and Marco Baldi. Analysis of reaction and timing attacks against cryptosystems based on sparse parity-check codes, 2019. URL: http://https://arxiv.org/pdf/1904.12215.pdf.

[Sen10] Nicolas Sendrier, editor. *Post-Quantum Cryptography, Third International Workshop, PQCrypto 2010, Darmstadt, Germany, May 25-28, 2010. Proceedings*, volume 6061 of *Lecture Notes in Computer Science*. Springer, 2010. doi:10.1007/978-3-642-12929-2.

[Sho97] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, 1997.

[Sin00] Simon Singh. *The code book: the secret history of codes and code-breaking*. Fourth Estate, 2000.

[SS86] Claus-Peter Schnorr and Adi Shamir. An optimal sorting algorithm for mesh connected computers. In Juris Hartmanis, editor, *Proceedings of the 18th Annual ACM Symposium on Theory of Computing, May 28-30, 1986, Berkeley, California, USA*, pages 255–263. ACM, 1986. doi:10.1145/12130.12156.

[SSMS09] Abdulhadi Shoufan, Falko Strenzke, H. Gregor Molter, and Marc Stöttinger. A timing attack against Patterson algorithm in the McEliece PKC. In Dong Hoon Lee and Seokhie Hong, editors, *Information, Security and Cryptology - ICISC 2009, 12th International Conference, Seoul, Korea, December 2-4, 2009, Revised Selected Papers*, volume 5984 of *Lecture Notes in Computer Science*, pages 161–175. Springer, 2009. doi:10.1007/978-3-642-14423-3_12.

[SSPB19] Simona Samardjiska, Paolo Santini, Edoardo Persichetti, and **Gustavo Banegas**. A reaction attack against cryptosystems based on LRPC codes. In Schwabe and Thériault [ST19], pages 197–216. doi:10.1007/978-3-030-30530-7_10.

[ST19] Peter Schwabe and Nicolas Thériault, editors. *Progress in Cryptology - LATINCRYPT 2019 - 6th International Conference on Cryptology and Information Security in Latin America, Santiago de Chile, Chile, October 2-4, 2019, Proceedings*, volume 11774 of *Lecture Notes in Computer Science*. Springer, 2019. doi:10.1007/978-3-030-30530-7.

[STM$^+$08] Falko Strenzke, Erik Tews, H. Gregor Molter, Raphael Overbeck, and Abdulhadi Shoufan. Side channels in the McEliece PKC. In Buchmann and Ding [BD08], pages 216–229. doi:10.1007/978-3-540-88403-3_15.

[Str10] Falko Strenzke. A timing attack against the secret permutation in the McEliece PKC. In Sendrier [Sen10], pages 95–107. doi:10.1007/978-3-642-12929-2_8.

[Str12] Falko Strenzke. Fast and secure root finding for code-based cryptosystems. In Josef Pieprzyk, Ahmad-Reza Sadeghi, and Mark Manulis, editors, *Cryptology and Network Security, 11th International Conference, CANS 2012, Darmstadt, Germany, December 12-14, 2012. Proceedings*, volume 7712, pages 232–246. Springer, 2012. doi:10.1007/978-3-642-35404-5_18.

[Str13] Falko Strenzke. *Efficiency and implementation security of code-based cryptosystems*. PhD thesis, Technische Universität, 2013.

[Str15] Barry Strauss. *The Death of Caesar: The Story of History's Most Famous Assassination*. Simon and Schuster, 2015.

[SW16] Peter Schwabe and Bas Westerbaan. Solving binary *MQ* with Grover's algorithm. In Claude Carlet, M. Anwar Hasan, and Vishal Saraswat, editors, *Security, Privacy, and Applied Cryptography Engineering - 6th International Conference, SPACE 2016, Hyderabad, India, December 14-18, 2016, Proceedings*, volume 10076 of *Lecture Notes in Computer Science*, pages 303–322. Springer, 2016. doi:10.1007/978-3-319-49445-6_17.

[TJR01] Trieu-Kien Truong, Jyh-Horng Jeng, and Irving S. Reed. Fast algorithm for computing the roots of error locator polynomials up to degree 11 in Reed-Solomon decoders. *IEEE Trans. Communications*, 49(5):779–783, 2001. doi:10.1109/26.923801.

[VDvT02] Eric R. Verheul, Jeroen M. Doumen, and Henk C. A. van Tilborg. Sloppy Alice attacks! Adaptive chosen ciphertext attacks on the McEliece public-key cryptosystem. In Mario Blaum, Patrick G. Farrell, and Henk C. A. van Tilborg, editors, *Information, Coding and Mathematics: Proceedings of Workshop honoring Prof. Bob McEliece on his 60th birthday*, pages 99–119, Boston, MA, 2002. Springer US. doi:10.1007/978-1-4757-3585-7_7.

[vOW94] Paul C. van Oorschot and Michael J. Wiener. Parallel collision search with application to hash functions and discrete logarithms. In Dorothy E. Denning, Raymond Pyle, Ravi Ganesan, and Ravi S. Sandhu, editors, *CCS '94, Proceedings of the 2nd ACM Conference on Computer and Communications Security, Fairfax, Virginia, USA, November 2-4, 1994.*, pages 210–218. ACM, 1994. doi:10.1145/191177.191231.

[Wie99] Michael J. Wiener, editor. *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*. Springer, 1999. doi:10.1007/3-540-48405-1.

[WSN18] Wen Wang, Jakub Szefer, and Ruben Niederhagen. FPGA-based Niederreiter cryptosystem using binary Goppa codes. In Tanja Lange and Rainer Steinwandt, editors, *Post-Quantum Cryptography - 9th International Conference, PQCrypto 2018, Fort Lauderdale, FL, USA, April 9-11, 2018, Proceedings*, volume 10786 of *Lecture Notes in Computer Science*, pages 77–98. Springer, 2018. doi:10.1007/978-3-319-79063-3_4.