Technische Universität München

Attacking and Optimizing Code-Based Post-Quantum Cryptosystems

Dissertation

Sabine H. Pircher

TUM School of Computation, Information and Technology

Professorship for Coding and Cryptography

TECHNISCHE UNIVERSITÄT MÜNCHEN
TUM SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY

Attacking and Optimizing Code-Based Post-Quantum Cryptosystems

SABINE H. PIRCHER

Vollständiger Abdruck der von der TUM School of Computation, Information and Technology der Technischen Universität München zur Erlangung einer

Doktorin der Ingenieurwissenschaften (Dr.-Ing.)

genehmigten Dissertation.

Vorsitz:	Prof. Dr.-Ing. Georg Sigl
Prüferinnen der Dissertation:	1. Prof. Dr.-Ing. Antonia Wachter-Zeh
	2. Jun.-Prof. Dr.-Ing. Elif Bilge Kavun

Die Dissertation wurde am 25.09.2023 bei der Technischen Universität München eingereicht und durch die TUM School of Computation, Information and Technology am 31.01.2024 angenommen.

Zusammenfassung

Das Forschungsfeld der Post-Quanten-Kryptografie (PQC) beschäftigt sich mit kryptografischen Verfahren, die auf klassischen binären Computern laufen und als resistent gegen Angriffe von Quantencomputern gelten. In der vorliegenden Dissertation werden Optimierungen und Angriffe auf das asymmetrische kryptografische Verfahren *Classic McEliece*, welches auf das *McEliece* kryptografische Verfahren und ihrer Variante dem *Niederreiter* Verfahren aufbaut und auf Fehlerkorrekturverfahren basiert, untersucht. Die grundlegenden mathematischen Aspekte, sowie auch die implementierungsspezifischen Details sind Schwerpunkte dieser Arbeit.

Generalisierte Goppa Codes (GGC), die auch (L,g)-Codes genannt werden, werden auf ihre Eigenschaften erforscht und ihr Einsatz in einem Niederreiter-basierten Verfahren evaluiert. Eine Anleitung zum Aufstellen der Paritätsprüfmatrix für Stützstellen jeglichen Grades wird konstruiert, sowie auch ein effizienter Decoder vorgestellt. Bei binären generalisierten Goppa Codes kann die Körpergröße viel kleiner gewählt werden, als bei binären klassischen Goppa Codes und trotzdem dieselbe Codelänge erreicht werden. Es wird gezeigt, dass die untere Schranke des Minimalabstandes verbessert werden kann und dass ein Decoder, welcher den erweiterten euklidischen Algorithmus benutzt, bis zur Hälfte des Minimalabstandes decodieren kann. Ein Einsatz der binären generalisierten Goppa Codes in einem Niederreiter Verfahren hat keine Reduzierung des öffentlichen Schlüssels, bei gleichbleibenden Sicherheitslevel bezüglich Information Set Decoding (ISD), im Vergleich zu den klassischen Goppa Codes hervorgebracht.

Implementierungen kryptografischer Verfahren können durch die Nutzung dedizierter Hardware für Vektoroperationen, die in Prozessoren integriert ist, beschleunigt werden. In dieser Arbeit wird gezeigt, wie das Gausseliminationsverfahren anhand der freien RISC-V Befehlssatzarchitektur (ISA) und ihrer zugehörigen Erweiterung für Vektoroperationen, beschleunigt wird. Simulationen auf einem angepassten Befehlssatzsimulator demonstrieren eine Reduktion der Speicherzugriffe und eine Laufzeitbeschleunigung um das 6 bis 18-fache für eine Speicherschnittstellenbreite von 64 bit bis 256 bits, im Vergleich zu einer nicht-vektorisierten Implementierung.

Es wird ein Fehlerangriff auf das *Classic McEliece* Verfahren entwickelt. Durch die freie Wahl von Ciphertexten und spezifischen Fehlerinjektionen auf das Fehlerlokalisierungspolynom und die Validitätsprüfungen beim Entschlüsseln kann der private Schlüssel rekonstruiert werden, vorausgesetzt man hat Zugriff auf die Ein- und Ausgänge der Entschlüsselungsfunktion. Bei jedem Entschlüsselungsvorgang wird eine Gleichung in den Unbekannten der geheimen Stützstellen aufgestellt und in einem Gleichungssystem zusammengefasst. Dessen Lösung wird dazu benutzt ein dazugehöriges Goppapolynom aufzustellen und damit einen alternativen privaten Schlüssel zu generieren. Die Fehlerinjektionen werden für zwei RISC-V Prozessoren auf Registertransferebene (RTL) simuliert und die Fehlerpunkte, die zu einem erfolgreichen Angriff führen, identifiziert.

Abstract

Post-quantum cryptography (PQC) refers to cryptographic systems that run on classical binary computers but are considered to be resistant against attacks from quantum computers. This thesis studies optimizations and attacks for public-key post-quantum cryptosystems based on error correcting codes. The focus is on the McEliece cryptosystem and its Niederreiter variation, in particular the Classic McEliece key-encapsulation mechanism that is built on it. General mathematical aspects as well as implementation details on hardware are examined.

Generalized Goppa Codes (GGC), also called generalized (L,g)-codes, are researched and evaluated for use in a Niederreiter cryptosystem. The properties of binary GGC as well as an efficient decoder are presented. The field size can be much smaller than for binary classical Goppa codes to achieve the same code length. A construction of parity-check matrices for any degree of code locators is given. It is shown that the lower bound on the minimum distance is improved and a decoding algorithm using the Extended Euclidean Algorithm (EEA) can decode errors up to half of the minimum distance. Code parameters for binary GGC that reduce the public key size for given security level based on Information Set Decoding (ISD) could not be found and classical Goppa codes still show the best public key size.

Implementations of Classic McEliece can be accelerated on processors having dedicated hardware for vector operations. The RISC-V open source Instruction Set Architecture (ISA) and its vector extension are used to optimize the Gaussian elimination algorithm needed in the cryptosystem. Simulations on a dedicated instruction set simulator show the reduction of memory accesses and an acceleration by a factor of 6 to 18 depending on the size of the memory port width from 64 to 256 bits, respectively.

A chosen-ciphertext fault attack on Classic McEliece is developed. Having full control of the input and output of the decryption function, fault injections on the error locator polynomial of the Goppa code and on the validity checks of the cryptosystem can lead to recovery of the secret key. Faulty decryption outputs are utilized to generate a system of polynomial equations in the secret support elements of the Goppa code. Solving the system of equations and determining a suitable corresponding Goppa polynomial results in an alternative secret key that can be used to encrypt and decrypt in place of the original secret key. The fault injections are simulated at the Register Transfer Level (RTL) of two RISC-V processors, finding viable injection points.

Acknowledgement

I want to thank Prof. Dr. Antonia Wachter-Zeh for supervising this thesis. She gave me the opportunity to start my PhD studies in collaboration with industry and supported me through unexpected turns, both scientifically and practically.

I acknowledge HENSOLDT Cyber for financing the bulk of my research. In particular, I thank Alexander Zeh, my initial supervisor at the company, for mentoring and initiating my research.

My time as doctoral researcher has been shaped by the COVID-19 pandemic that erased many of the joys that may otherwise characterize PhD life. I especially want to thank my colleague Johannes Geier, who accompanied me throughout my PhD with video calls and long discussions during times of social distancing. I thank my co-authors Daniel Müller-Gritschneder, Julian Danner and Hedongliang Liu for good collaboration and everyone at COD for the stimulating environment.

I am grateful to my family and friends for supporting me all along the way, especially to my parents and my husband for their unlimited patience and support.

Contents

Abbreviations X

Nomenclature XII

1 Motivation & Overview 1

2 Background of Coding Theory 5
2.1 Error Correcting Codes . . . 5
2.1.1 Finite Fields . . . 6
2.1.2 Linear Codes . . . 8
2.1.3 Goppa Codes . . . 10
2.1.4 Syndrome-Based Unique Decoding . . . 11
2.2 Extended Euclidean Algorithm (EEA) . . . 13
2.3 Syndrome Decoding Problem (SDP) . . . 16

3 Introduction to the McEliece Cryptosystem and its Variations 19
3.1 McEliece Cryptosystem . . . 20
3.2 Niederreiter Cryptosystem . . . 22
3.3 Classic McEliece Cryptosystem . . . 23
3.3.1 Key generation . . . 23
3.3.2 Encapsulation . . . 24
3.3.3 Decapsulation . . . 24
3.4 Security Levels and Categories . . . 27
3.5 Best Known Attacks . . . 30
3.5.1 Find secret key from public key - Key Attack . . . 30
3.5.2 Information Set Decoding - Message Attack . . . 32

4 Investigating Generalized Goppa Codes 37
4.1 Generalized Goppa Codes . . . 37
4.1.1 Binary Generalized Goppa Codes . . . 38
4.1.2 Decoding of Binary Generalized Goppa Codes . . . 44
4.2 Niederreiter Cryptosystem with Binary Generalized Goppa Codes . . . 46

5 Accelerations using the RISC-V Vector Extension 53
5.1 Overview of Classic McEliece implementations . . . 54
5.1.1 Platform independent implementations . . . 55
5.1.2 AVX/SSE Implementations . . . 56
5.2 Performance Analysis of Implementations . . . 57

5.3 Gaussian Elimination Algorithm (GEA) . . . 58
5.4 Introduction to RISC-V . . . 60
5.4.1 RISC-V Vector Extension (RVV) . . . 61
5.5 Acceleration of GEA with RVV . . . 63
5.6 Experimental Evaluation . . . 65
5.6.1 Simulation Environment . . . 66
5.6.2 Performance Analysis . . . 66

6 Attacking Classic McEliece using Fault Injections 69
6.1 Mathematical Description of Key-Recovery Fault Injection Attack . . . 70
6.1.1 Fault Model . . . 71
6.1.2 Implementation Specific Behaviour of Decoding . . . 71
6.1.3 Overview of Attack . . . 72
6.1.4 Fault Injection on the Validity Checks (VCB) . . . 73
6.1.5 Locating the Zero Element in the Support . . . 74
6.1.6 Fault Injection on the ELP Coefficients . . . 75
6.2 Computation of Alternative Secret Key . . . 79
6.2.1 System of polynomial equations contains all unknowns of the support set . . . 79
6.2.2 System of polynomial equations contains only a subset of unknowns . . . 81
6.3 Simulation . . . 82
6.3.1 Key-Recovery Simulation . . . 82
6.3.2 De-hashing: Obtaining the faulty error vector from hash output 84
6.3.3 Simulation at Register Transfer Level . . . 85

7 Conclusion 91

A Source Code of Vectorized GEA 95

B Gröbner Basis and Buchberger's Algorithm 99
B.1 Background of Gröbner Basis . . . 99
B.2 Solving Systems of Polynomial Equations . . . 100

Bibliography 103

Abbreviations

AES	Advanced Encryption Standard
ASIC	Application-Specific Integrated Circuit
BCH	Bose–Ray-Chaudhuri–Hocquenghem
BIKE	Bit Flipping Key Encapsulation
BSI	Bundesamt für Sicherheit in der Informationstechnik
CISC	Complex Instruction Set Computer
CSR	Control Status Register
DSL	Domain-Specific Language
EEA	Extended Euclidean Algorithm
EEP	Error Evaluator Polynomial
ELP	Error Locator Polynomial
FPGA	Field Programmable Gate Array
GEA	Gaussian Elimination Algorithm
GCC	GNU Compiler Collection
GGC	Generalized Goppa Codes
GPR	General Purpose Register
GRS	Generalized Reed–Solomon
HDL	Hardware Description Language
HQC	Hamming Quasi-Cyclic
IEC	International Electrotechnical Commission
IND-CCA2	Indistinguishability under Chosen Ciphertext Attacks
IND-CPA	Indistinguishability under Chosen Plaintext Attacks
ISA	Instruction Set Architecture
ISD	Information Set Decoding
ISO	International Organization for Standardization
ISS	Instruction Set Simulation
KEM	Key Encapsulation Mechanism
MDPC	Moderate Density Parity-Check Codes
NIST	U.S. National Institute of Standards and Technology
PKE	Public-Key Encryption
PQC	Post-Quantum Cryptography
RSA	Rivest, Shamir, Adleman
RTL	Register Transfer Level
RVV	RISC-V Vector Extension
SDP	Syndrome Decoding Problem
SIKE	Supersingular Isogeny Key Encapsulation
SSE	Streaming SIMD Extension

TEE	Trusted Execution Environment
VHDL	Very High Speed Integrated Circuits Hardware Description Language
VLSU	Vector Load/Store Unit
VP	Virtual Prototype

Nomenclature & Symbols

Basics

$\mathbf{x}$	Row vector
$\mathbf{A}$	Matrix
$a_{i,j}$	Element in i-th row and j-th column of matrix $\mathbf{A}$
$\mathbf{I}_k$	Identity matrix of size $k \times k$
$\mathbf{A} \cdot \mathbf{B}$	(Matrix) product
$\langle \mathbf{u}, \mathbf{v} \rangle$	Scalar (inner) product
$\mathbf{A}^\top$	Transpose of matrix $\mathbf{A}$
$\mathcal{E} = \{\dots\}$	Set of elements
$\lvert\mathcal{E}\rvert = \lvert\{\dots\}\rvert$	Cardinality of a set
$\gcd(x, y)$	Greatest common divider of x and y
$\mathrm{supp}(\mathbf{x})$	Support of $\mathbf{x}$ (indices of non-zero elements of $\mathbf{x}$)
$a \mid b$	a divides b, a is a divisor of b

Finite Fields

$\mathbb{F}_p$	Prime field of size p
$\mathbb{F}_q = \mathbb{F}_{p^m}$	Extension field of size p^m with extension degree m
$\mathbb{F}_2 = \{0, 1\}$	Binary finite field
$\mathbb{F}_{2^m}$	Binary extension field of size 2^m with extension degree m
$\mathbb{F}_q[x]$	Set of univariate polynomials with coefficients in $\mathbb{F}_q$
$\mathbb{F}_q[x_1, \dots, x_n]$	Set of multivariate polynomials with coefficients in $\mathbb{F}_q$
$\mathbb{F}_q^{a \times b}$	Set of all $a \times b$ matrices over $\mathbb{F}_q$
$\mathbb{F}_q^n = \mathbb{F}_q^{1 \times n}$	Set of all row vectors of length n over $\mathbb{F}_q$

Codes

$\mathcal{C}$	Code (set of vectors)
n	Length of a code
k	Dimension of a code
d	Minimum distance of a code
$g(x)$	Goppa polynomial

t	Degree of a Goppa polynomial
$\mathcal{L}$	Set of code locators with n elements (also called support)
$\mathbf{G}$	Generator Matrix
$\mathbf{H}$	Parity-Check Matrix
$\mathrm{wt_H}(\mathbf{c})$	Hamming weight of vector $\mathbf{c}$

Cryptography

$\mathcal{H}(.)$	SHA-3 Keccak SHAKE-256 hash function
$\xleftarrow{\$}$	uniformly randomly chosen
$\mathcal{O}$	Big O notation
$\mathbf{K}$	Scrambled public key of Niederreiter cryptosystem
$\mathbf{G}'$	Scrambled public key of McEliece cryptosystem
$\mathbf{T}$	Public key of Classic McEliece cryptosystem
K	Session key of KEM
$\mathcal{A}$	Adversary
$\mathsf{Gen}(.)$	Key generation function
$\mathsf{Enc}(.)$	Encryption function
$\mathsf{Dec}(.)$	Decryption function

Chapter 1

Motivation & Overview

The history of cryptography goes back to ancient times, where people used various techniques to hide sensitive information. In earlier times the understanding of cryptography was in the sense of encryption, meaning that readable information was concealed to an unintelligible form. Only with the knowledge of some special secret a legitimate receiver could restore the original information. This corresponds to today's security requirement of *confidentiality*, the protection against unauthorized information retrieval. Modern cryptography does not only refer to securely exchanging sensitive information. It makes also sure that sensitive information is protected against unauthorized and unnoticed modification (*data integrity*). Furthermore, cryptography is used to ensure the authenticity of an identity (*authenticity*) and to protect against the denial of performed actions (*non-repudiation*).

Today, there exist two different fundamental types of cryptographic schemes: symmetric and asymmetric. In symmetric cryptography both sender and legitimate receivers of a message have the same secret key. A copy of this secret key needs to have been exchanged a-priori over a secure channel, e.g. physically. In asymmetric cryptography sender and receivers possess different keys. In encryption systems each receiver generates a public-secret keypair and sends the public key to the sender that uses it to encrypt the message. In digital signatures, a message is signed using the secret key of the signer and everyone can verify the integrity and authenticity of the message using the corresponding public key of the signer. Thus, asymmetric cryptography is also called public-key cryptography. Additionally, hash functions are an important part of today's cryptography. Hash functions map a variable size input to a fixed size output. This mapping is unique and cannot be inverted. Thus, these are one-way functions.
Current cryptographic schemes are internationally standardized by the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC) in the joint technical committee ISO/IEC JTC 1/SC 27 [ISO]. Individual countries also set their own recommendations, guidelines and/or national standard specifications.

For symmetric encryption schemes one of the famous and widely implemented cryptosystem is Advanced Encryption Standard (AES) [DBN+01; ISO10], which is based on the Rijndael algorithm. For asymmetric cryptosystems the Rivest, Shamir, Adle-

man (RSA) [MKJR16; ISO06] algorithm and Elliptic Curve Cryptography (ECC) [ISO16; ISO22] mechanisms are widely adopted and implemented. The main security of cryptosystems relies on mathematical problems that are mathematically proven to be hard so solve. Current standardized cryptosystems are designed to be hard for classical (binary) computers that calculate using transistors that represent two states (0 and 1). The security of RSA is based on the mathematical hardness of finding the prime factors of an integer while the security of ECC depends on the difficulty of determining discrete logarithms. The parameters of these algorithms (e.g. key length) are chosen such that the cryptosystem meets a desired security level against attacks running on classical computers.

The advent of quantum computers challenges widely used cryptography. Quantum computers work on qbits that can represent many states in parallel. They operate differently from classical computers, allowing them to solve some specific mathematical problems more quickly. Quantum computers are well suited for solving optimization problems, data analysis and simulations. Therefore, the most relevant potential attacks on the current deployed cryptosystems will change once capable quantum computers are available. Current standardized cryptosystems need to be updated and/or replaced accordingly to meet also the security levels for attacks from quantum computers. In 1996, Grover published a special search algorithm for quantum computers [Gro96; Gro97]. He showed that in an unsorted database that contains N records, one specific record can be found in only $\mathcal{O}(\sqrt{N})$ evaluation steps. For classical computers this problem cannot be solved in fewer than $\mathcal{O}(N)$ evaluation steps. This means the quantum mechanical algorithm for searching is polynomially faster than any classical algorithm. For symmetric cryptography Grover's algorithm allows to speed-up brute-force attacks such as an exhaustive key search on AES [GLRS16]. Therefore, the security level of AES needs to be reassessed, finding that the current key lengths need to roughly be doubled to ensure security in the future. In 1994, Shor developed a quantum mechanical algorithm for factorization that brings exponential speed-up compared to classical algorithms [Sho94]. Using this algorithm, cryptosystems based on integer factorizations and discrete logarithms will be broken in polynomial time. Hence, all current cryptosystems based on these mathematical problems (e.g. RSA, DSA, ECC) are no longer secure enough once capable quantum computers are available and must therefore be exchanged with new ones.

New asymmetric cryptosystems designed to be secure enough against attacks performed with quantum computers are urgently needed. Cryptography, including symmetric encryption schemes, that withstands attacks from quantum computers and is executed on classical computers is called Post-Quantum Cryptography (PQC). PQC relies on the hardness of solving mathematical problems with both classical and quantum computers, e.g. lattice problems, decoding error correcting codes, solving multivariate equations or zero-knowledge proofs. In 2016, the U.S. National Institute of Standards and Technology (NIST) started a competition for post-quantum Key Encapsulation Mechanism (KEM) and digital signature systems to standardize one or more algorithms and to replace their current national standard and recommendations (e.g. FIPS 186 and SP 800-56A/B/C) [Nat17]. Some of the evaluation criteria are

a well established security analysis, performance analysis on x86 machines and key sizes.

The submitted schemes changed and improved over the competition rounds. By the end of Round 3 the KEM CRYSTALS-Kyber, a cryptosystem based on lattice problems, was chosen as the encryption scheme to be standardized. For digital signatures, CRYSTALS-Dilithium and Falcon, which are also based on lattice problems, and SPHINCS+, which is based on hash functions, were chosen to be standardized by NIST. As it is desirable to standardize schemes based on different mathematical problems, NIST opened up a new competition for digital signatures, and for the already running competition they introduced an additional Round 4 consisting of 3 code-based cryptosystems and one isogeny-based cryptosystem. The isogeny-based cryptosystem Supersingular Isogeny Key Encapsulation (SIKE) was broken in 2022 as it has been surprisingly shown that it is attackable by a classical computer in [CD22]. The other code-based cryptosystems are Classic McEliece, Hamming Quasi-Cyclic (HQC)[MAB+21] and Bit Flipping Key Encapsulation (BIKE) [ABB+22].

The work presented in this thesis focuses on Classic McEliece, the cryptosystem submitted to NIST [BCL+19; BCL+20]. Classic McEliece is a McEliece cryptosystem [McE78] implementing the Niederreiter variant [Nie86]. McEliece is an old (from 1978) and well researched cryptosystem. It is based on error-correcting codes and the hardness of decoding a random code (`NP-completeness` of the Syndrome Decoding Problem (SDP)). The code class that is used in Classic McEliece is Goppa codes, which still today shows resistance against structural attacks. The advantage of Classic McEliece is the efficient encoding and decoding and its reliable decryption. A drawback is the large public key size compared to other cryptosystems in the NIST competition. The German Bundesamt für Sicherheit in der Informationstechnik (BSI) supports the international standardization of Classic McEliece at current standardization efforts of ISO for PQC [Ste22].

In this thesis, research towards an efficient and secure implementation of the code-based cryptosystem Classic McEliece was done, considering the research questions below.

Research questions

- Which optimizations in terms of computational complexity (time and resource consumption) can be done for the cryptosystem?
- How can the size of the public key be reduced from the mathematical point of view? How does this impact the security level?
- Which algorithmic aspects support hardware fault attacks?

Chapter 2 gives an introduction to coding theory and reviews the mathematical basics. In Chapter 3 the Classic McEliece cryptosystem is presented and its security discussed.

In Chapter 4, the code class of Generalized Goppa Codes (GGC) is researched and investigated for use in the Classic McEliece cryptosystem. For binary GGC,

a construction of parity-check matrices is developed and bounds for the minimum distance, dimension and unique decoding radius are derived. A discussion on Classic McEliece with binary GGC with focus on public key size, security level and field size is given.

Chapter 5 presents an optimization of Classic McEliece at the implementation level, by using the RISC-V vector extension. Profiling the implementation revealed the most time and resource intensive computations of the cryptosystem. A rapid prototyping approach based on an Instruction Set Simulation (ISS) is presented to accelerate a Gaussian Elimination Algorithm (GEA) with the instructions of the RISC-V vector extension.

In Chapter 6, a fault-attack on the Classic McEliece cryptosystem is presented. The developed method recovers the secret key using fault injections on the Error Locator Polynomial (ELP). A description of the attack that finds the support of the Goppa code by setting up a system of polynomial equations is given. The system of polynomial equations is solved in the finite extension field using Gröbner basis and Buchberger's algorithm. If part of the secret key is revealed, the complete secret key can be reconstructed. The attack is simulated on software level in C-code and solved in SageMath. A simulation and feasibility study for RISC-V on RTL level via Virtual Prototype (VP) is established.

A conclusion and summary is given in the last Chapter 7.

Chapter 2

Background of Coding Theory

This chapter introduces the coding theory background used throughout this thesis. It is intended to provide readers with the tools to understand the body of this thesis and to introduce the notation used in the following chapters. The content presented in this chapter can be found in textbooks (e.g. [LN97; Rot06; MS83; HP03]).

First, the algebraic structure of finite fields is introduced. Especially extension fields and polynomials over these fields are described and some important properties used in this thesis are given. Based on this, linear error correcting codes are introduced. These concepts are then particularised to the case of Goppa codes. Syndrome-based unique decoding is explained specifically for Goppa codes.

Furthermore, Section 2.2 discusses the Extended Euclidean Algorithm (EEA) in detail, as it is used several times in the remainder of this work. The final part of this chapter concerns the Syndrome Decoding Problem (SDP), the mathematically hard problem on which code-based cryptography relies.

2.1 Error Correcting Codes

Traditionally, error correcting codes are used to efficiently encode and decode messages containing information, as well as redundancy so that the information can be reconstructed in case the message gets corrupted on its way to the receiver. Common applications are in digital radio communication or data storage media.

Conceptually, one can think of an error-correcting code as a mapping between some information and a message, that will be sent over a channel. Usually, the information $\mathbf{u}$ is represented as a vector of k elements (e.g. information bits) and is mapped to a codeword $\mathbf{c}$, represented as a vector of n elements, see Eq. (2.1). The codeword is sent as a message over the channel. All possible codewords together form the code.

$$\mathbf{u} = (u_1, u_2, \ldots, u_k) \longmapsto (c_1, c_2, \ldots, c_n) = \mathbf{c} \qquad n \geqslant k \tag{2.1}$$

In the following, we introduce Goppa codes, a family of error-correcting codes introduced by Goppa in 1970 [Gop70]. They are a class of linear codes acting on finite fields. The final part of this section introduces syndrome-based unique decoding at the example of Goppa codes.

+	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	0
2	2	3	4	0	1
3	3	4	0	1	2
4	4	0	1	2	3

(a) Addition

·	0	1	2	3	4
0	0	0	0	0	0
1	0	1	2	3	4
2	0	2	4	1	3
3	0	3	1	4	2
4	0	4	3	2	1

(b) Multiplication

Table 2.1: Operation tables for $\mathbb{F}_5$

2.1.1 Finite Fields

Finite fields are an algebraic structure investigated in mathematical group theory. They were first described by Galois in 1830. A finite field, as the name suggests, is a field that consists of a finite number of elements. That is in contrast to real numbers or integer numbers which are fields that consist of an infinite number of elements.

In general, a field is represented as a set of elements, on which the operations of addition and multiplication are defined. These operations follow the law of commutativity, distributivity, associativity and closure. There exists a neutral element for both an additive and multiplicative operation. For every element of a field there exists an additive inverse element and except for the additive neutral element also a multiplicative inverse element. For $a, b, c \in \mathbb{F}$ we have:

$$\begin{aligned} \text{Associativity: } & a + (b + c) = (a + b) + c, \quad a(bc) = (ab)c \\ \text{Distributivity: } & a(b + c) = ab + ac \\ \text{Commutativity: } & a + b = b + a, \quad ab = ba \end{aligned}$$

For closure, the result of an operation on two elements is again an element of the field $\mathbb{F}$. Common finite fields are prime fields whose elements are represented by a finite set of integers. Addition and multiplication on prime fields work the same way as on integers, only that their result is taken modulo a prime number p. For example, the addition and multiplication tables for $< \mathbb{F}_5 = \{0, 1, 2, 3, 4\}, + \mod 5, \cdot \mod 5 >$ are given in Table 2.1. As, for a binary field $\mathbb{F}_2$ are given in Table 2.2. An addition in $\mathbb{F}_2$ can be executed by classical computers via a logical XOR and a multiplication in $\mathbb{F}_2$ via a logical AND.

A finite field $\mathbb{F}_{p^m}$ is an extension field of $\mathbb{F}_p$, if there exist a subset $\mathbb{F}_p \subset \mathbb{F}_{p^m}$ that is itself a field under the operations of $\mathbb{F}_{p^m}$. Finite extension fields are constructed using operations modulo a prime power and are denoted by $\mathbb{F}_{p^m} = \mathbb{F}_q$ of prime p and extension degree m. Every finite field is of prime-power order [LN97, Chap. 2]. It has p^m elements, where p is called the characteristic of $\mathbb{F}_{p^m}$ and m its degree of its prime subfield. Elements of an extension field can be represented as a polynomial of degree $m - 1$ with coefficients in the base field $\mathbb{F}_p$, e.g.

$$\mathbb{F}_{2^2} = \mathrm{GF}(4) = \{0z^1 + 0z^0, 0z^1 + 1z^0, 1z^1 + 0z^0, 1z^1 + 1z^0\} = \{0, 1, z, z + 1\}.$$

$+$	0	1
0	0	1
1	1	0

(a) Addition (XOR)

$\cdot$	0	1
0	0	0
1	0	1

(b) Multiplication (AND)

Table 2.2: Operation tables for $\mathbb{F}_2$

Operations on the elements are conducted on the polynomials. The coefficients follow the operations of their base field, e.g. $\mathbb{F}_5$ or $\mathbb{F}_2$ as given in Table 2.1 and Table 2.2. Multiplicative operations are conducted via polynomial multiplication and division modulo a suitable irreducible polynomial $a(z)$ (defined below) for the finite field. The irreducible polynomial $a(z)$ defines the field and the extension field can also be denoted by $\mathbb{F}_{p^m}[z]/a(z)$. For storing elements of an extension field in bits on a classical computer, a representation as array of size m with suitable datatype to store the coefficients as elements on the base field $\mathbb{F}_p$ can be chosen.

One can also consider polynomials whose coefficients are elements of an extension field $\mathbb{F}_q$. Univariate polynomials with indeterminate x are denoted by $\mathbb{F}_q[x]$ and multivariate polynomials with indeterminates $x_1, \dots, x_n$ are denoted by $\mathbb{F}_q[x_1, \dots, x_n]$. In the following, we introduce some properties of polynomials in extension fields.

Theorem 2.1 (Roots of Polynomials [LN97, Thm. 1.64]).
An element $b \in \mathbb{F}_q$ is a root of the polynomial $f(x) \in \mathbb{F}_q[x]$ if and only if $x - b$ divides $f(x)$.

Definition 2.2 (Irreducible Polynomial [LN97, Def. 1.57]).
A polynomial $f(x) \in \mathbb{F}_q[x]$ is irreducible over $\mathbb{F}_q$ if $f(x)$ has positive degree and $f(x) = a(x) \cdot b(x)$ with $a(x), b(x) \in \mathbb{F}_q[x]$ and either $a(x)$ or $b(x)$ is a constant polynomial.

In other words, if a polynomial $f(x) \in \mathbb{F}_q[x]$ has no divisors in $\mathbb{F}_q[x]$ except scalars in $\mathbb{F}_q$ and scalar multiples of itself, then $f(x)$ is an irreducible polynomial in $\mathbb{F}_q[x]$.

Definition 2.3 (Splitting Field [Rot06, Prop. 3.14]).
Let $f(x)$ be a polynomial of degree $n \geqslant 0$ over a field $\mathbb{F}_{p^m}$. Then, there exists an extension field $\mathbb{F}_{p^s}$ of $\mathbb{F}_{p^m}$ in which $f(x)$ has n roots, with s a positive integer and $s \geqslant m$.

The extension field $\mathbb{F}_{p^s}$ to which all roots of the polynomial $f(x) \in \mathbb{F}_{q^m}[x]$ belong is the splitting field of $f(x)$.

The following relation holds for any q a prime power and $f(x)$ a polynomial of $\mathbb{F}_q[x]$ using Exercise 2.17 and Lemma 2.3 in [LN97] .

Lemma 2.4 ([LN97, Lemma 2.3, Ex. 2.17]).
Let $\mathbb{F}_q$ be a finite field with q elements, then for every $f(x) \in \mathbb{F}_q[x]$ applies

$$f(x^q) = f(x)^q \tag{2.2}$$

The splitting roots of a polynomial are defined in the next theorem.

Theorem 2.5 (Splitting Roots [LN97, Thm. 2.14]).
Any irreducible polynomial $f(x) \in \mathbb{F}_q[x]$ of degree τ can be represented as

$$f(x) = (x+\beta)(x+\beta^q)\cdots(x+\beta^{q^{\tau-1}}), \tag{2.3}$$

where $\beta \in \mathbb{F}_{q^\tau}$ and $\mathbb{F}_{q^\tau}$ is the splitting field of $f(x)$.

2.1.2 Linear Codes

A linear (block) code $\mathcal{C}$ is a k-dimensional linear subspace of $\mathbb{F}_q^n$ and thus consists of a set of codewords $\mathbf{c}$ of length n. The elements of a codeword are from a finite field $\mathbb{F}_q$ such that $\mathbf{c} \in \mathbb{F}_q^n$. The information is represented with k information elements also from the finite field $\mathbb{F}_q$ with $\mathbf{u} \in \mathbb{F}_q^k$. The maximum number of different information vectors is q^k and all are one-to-one mapped to a corresponding codeword that consists of the information and redundancy. Therefore, the number of codewords is also q^k and specifies the size of the code. The overhead number of redundancy elements in a codeword is $n-k$. Thus, a rate

$$R = \frac{k}{n} \tag{2.4}$$

of the code can be achieved. To encode an information vector to a codeword vector in linear codes a generator matrix $\mathbf{G}$ of size $k \times n$ uniquely defines the map

$$\mathbf{c} = \mathbf{uG} \tag{2.5}$$

for all possible information vectors $\mathbf{u}$. The generator matrix contains k basis vectors that span the code $\mathcal{C}$. Hence, a linear block code is a k-dimensional subspace of the vector space $\mathbb{F}_q^n$, where $1 \leqslant k < n$,

$$\mathcal{C} = \{\mathbf{uG} \mid \mathbf{u} \in \mathbb{F}_q^{1\times k}\}. \tag{2.6}$$

The value k is also called the dimension of the code. As there exists many bases to span a vector space, there exist many different generator matrices that define the same set of codewords. Only, the encoding mapping changes among the different generator matrices. A generator matrix is called systematic, if it can be written in the form $\mathbf{G}_{\text{sys}} = [\mathbf{I}_{\text{k}} \mid \mathbf{A}]$ with $\mathbf{A} \in {\mathbb{F}_q}^{(n-k)\times k}$ and $\mathbf{I}_{\text{k}} \in {\mathbb{F}_q}^{k\times k}$ an identity matrix of size $k \times k$. Then, the first k elements of the systematic codewords $\mathbf{c}_{\text{sys}}$ correspond to the information vector $\mathbf{u}$ and the last $n-k$ elements are redundancy.

$$\mathbf{u} = (u_1, u_2, \ldots, u_k) \longmapsto (u_1, u_2, \ldots, u_k, q_1, \ldots, q_{n-k}) = \mathbf{c}_{\text{sys}} \qquad n \geqslant k \tag{2.7}$$

The all-zero codeword is always part of the code.

In order to study and construct codes with specific properties, e.g. the number of errors that can uniquely be corrected, it is useful to consider a specific dual space of the code that maps each codeword to a scalar product equal to zero. This is represented

by the dual code

$$\mathcal{C}^{\perp} = \{\mathbf{c}^{\perp} \in \mathbb{F}_q{}^n \mid \left\langle \mathbf{c}, \mathbf{c}^{\perp} \right\rangle = 0, \forall \mathbf{c} \in \mathcal{C}\}, \tag{2.8}$$

where $\langle \cdot, \cdot \rangle$ denotes the scalar product of two vectors. Each codeword of the code and each codeword of the dual code are orthogonal to each other. The basis vectors that span the dual code are represented by $\mathbf{H}$ and can be constructed using the property $\mathbf{G} \cdot \mathbf{H}^{\top} = 0$ that follows from the definition of the dual code. Therefore, it follows that $\mathbf{c}\mathbf{H}^{\top} = 0$ and it can be easily checked if a received message is a valid codeword $\mathbf{c}$ of the code $\mathcal{C}$. The matrix $\mathbf{H}$ is called parity-check matrix. A parity-check matrix can be constructed from the generator matrix in systematic form $\mathbf{G}_{\mathrm{sys}} = [\mathbf{I}_{\mathrm{k}} \mid \mathbf{A}]$ such that $\mathbf{H} = [-\mathbf{A}^{\top} \mid \mathbf{I}_{\text{n-k}}]$ and vice versa from the parity-check matrix to the generator matrix.

To quantify the utility of linear codes as error-correcting codes, we introduce the Hamming metric. The Hamming distance between two words is calculated by counting the number of elements that differ,

$$\mathrm{d}(\mathbf{x}, \mathbf{y}) := |\{i : x_i \neq y_i, i = 0, \ldots, n-1\}| \tag{2.9}$$

with $\mathbf{x}, \mathbf{y}$ vectors of length n. The Hamming weight of a word is then the number of elements that differ from zero,

$$\mathrm{wt}_{\mathrm{H}}(\mathbf{x}) = |\{i : x_i \neq 0, i = 0, \ldots, n-1\}|. \tag{2.10}$$

The minimum distance of a linear code is defined as the minimal weight of any codeword of the code,

$$d = \min_{\substack{\mathbf{x} \in \mathcal{C} \\ \mathbf{x} \neq 0}} \{\mathrm{wt}(\mathbf{x})\} \tag{2.11}$$

Consider a codeword that is transmitted over a noisy channel (discrete memoryless channels). A received word $\mathbf{r}$ is a noisy version of $\mathbf{c}$ and can be written as $\mathbf{r} = \mathbf{c} + \mathbf{e}$ with $\mathbf{e}$ denoting the error produced by the noisy channel. The maximum number of corrupted elements of a codeword that can be detected are $d-1$, with d the minimum distance of the code.

To find d from $\mathbf{H}$, every possible number of columns needs to be checked for linear independence. If any $d-1$ columns of $\mathbf{H}$ are linearly independent, the minimum distance of the code is found. Then, there exist d columns of $\mathbf{H}$ that are linearly dependent. Thus, any two codewords differ in at least $d-1$ elements. So, the code $\mathcal{C}$ can detect $d-1$ errors and uniquely correct $\left\lfloor \frac{d-1}{2} \right\rfloor$ errors. In communication systems it is desired to transmit as little redundancy as possible. Therefore, linear codes can be designed to have a high rate $R = \frac{k}{n}$. The most famous code classes based on the Hamming metric are (Generalized) Reed-Solomon codes, Bose-Ray-Chaudhuri-Hocquenghem (BCH)-codes, Hamming codes, Alternant codes, Goppa codes.

Properties of Goppa codes:

Minimum distance:	$d \geqslant \deg(g(x)) + 1$
Length:	$n \leqslant p^m$
Dimension:	$k = n - \operatorname{rank}(\mathbf{H}_{\mathrm{GC}}) \geqslant n - m \cdot \deg(g(x))$

2.1.3 Goppa Codes

Goppa codes [Gop70; Ber73] are codes that belong to the family of Alternant codes. Alternant codes are subfield subcodes of Generalized Reed-Solomon (GRS) codes. So, are Goppa codes. Goppa codes are specified by a Goppa polynomial $g(x)$ with coefficients in $\mathbb{F}_{p^m}$ for a positive integer m, and a set $\mathcal{L} = (\alpha_1, \ldots, \alpha_n)$ of n distinct elements chosen such that $\alpha_i \in \mathbb{F}_{p^m}$ and $g(\alpha_i) \neq 0$ for all $i = 1, \ldots, n$. Then, the Goppa code $\Gamma(\mathcal{L}, g)$ can be defined as follows.

Definition 2.6 (Goppa code).
Let $\mathcal{L} = \{\alpha_1, \ldots, \alpha_n\}$ with $\alpha_i \in \mathbb{F}_{p^m}$ and $g(x) \in \mathbb{F}_{p^m}[x]$ a polynomial with $g(\alpha_i) \neq 0$ for all $i = 1, \ldots, n$. The Goppa code is then

$$\Gamma(\mathcal{L}, g) := \left\{ \mathbf{c} \in \mathbb{F}_p^n \;\middle|\; \sum_{i=1}^{n} \frac{c_i}{x - \alpha_i} \equiv 0 \mod g(x) \right\}. \tag{2.12}$$

Goppa codes are also called $(\mathcal{L}, g)$-codes. The set $\mathcal{L}$ is referred to as the support and its elements α_i are called code locators, because their values and ordering define the code.

The degree of the Goppa polynomial is denoted by $t = \deg(g(x))$. The minimum distance d is at least $\deg(g(x)) + 1$ and thus at least $\left\lfloor \frac{\deg(g(x))}{2} \right\rfloor$ errors can uniquely be corrected. Based on the definition of $\mathcal{L}$, the length n of the code is restricted to $n \leqslant p^m$. Goppa codes can be constructed via their parity-check matrix. The matrix $\mathbf{H} \in \mathbb{F}_{p^m}^{t \times n}$ with

$$\mathbf{H} = \begin{pmatrix} \frac{1}{g(\alpha_1)} & \frac{1}{g(\alpha_2)} & \cdots & \frac{1}{g(\alpha_n)} \\ \frac{\alpha_1}{g(\alpha_1)} & \frac{\alpha_2}{g(\alpha_2)} & \cdots & \frac{\alpha_n}{g(\alpha_n)} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\alpha_1^{t-1}}{g(\alpha_1)} & \frac{\alpha_2^{t-1}}{g(\alpha_2)} & \cdots & \frac{\alpha_n^{t-1}}{g(\alpha_n)} \end{pmatrix} \tag{2.13}$$

is used to generate a parity-check matrix $\mathbf{H}_{\mathrm{GC}}$ of a Goppa code. The parity-check matrix $\mathbf{H}_{\mathrm{GC}} \in \mathbb{F}_p^{mt \times n}$ is obtained by replacing each entry $h_{ij} \in \mathbb{F}_{p^m}, i = 1, \ldots, t, j = 1, \ldots, n$ of $\mathbf{H}$ by the corresponding column vector $\mathbf{h}_{ij} \in \mathbb{F}_p^m$ of length m in $\mathbb{F}_p$. For this, a unique mapping of each element of the extension field $\mathbb{F}_{p^m}$ to a vector of length m in the base field $\mathbb{F}_p$ needs to be chosen. Remember that $\mathbb{F}_p \subset \mathbb{F}_{p^m}$ for $m > 1$ and it applies that $\mathbf{c}\mathbf{H}_{\mathrm{GC}}^\top = \mathbf{0}$ and $\mathbf{c}\mathbf{H}^\top = \mathbf{0}$ [MS83, Chap.7 §7]. The true dimension k of the code is

$$k = n - \operatorname{rank}(\mathbf{H}_{\mathrm{GC}}). \tag{2.14}$$

If $\mathbf{H}_{\text{GC}}$ has full rank, then $k = n - mt$ and $mt - 1 = n - k - 1$.

Several subtypes of Goppa codes are of interest. A Goppa code is called irreducible, if its Goppa polynomial is irreducible over $\mathbb{F}_{p^m}$. A Goppa code is cyclic if and only if $g(x) = x^t$. In that case the minimum distance is $d \leqslant t + 1$ and corresponds to the minimum distance d_{BCH} of a primitive BCH code [Gop70; Gop71]. Primitive BCH codes are the only subclass of Goppa codes that are cyclic with $g(x) = x^t$ and $\mathcal{L} \subseteq \mathbb{F}_{q^m} \backslash \{0\}$.

For binary Goppa codes the extension field is based on the prime $p = 2$ such that $\mathbb{F}_{p^m} = \mathbb{F}_{2^m}$. The elements α_i of $\mathcal{L}$ and the coefficients of $g(x)$ are in $\mathbb{F}_{2^m}$. Binary Goppa codes can also be written [MS83, p. 341] as

$$\Gamma_{\text{bin}}(\mathcal{L}, g) := \left\{ \mathbf{c} \in \mathbb{F}_2^n \;\middle|\; \sum_{i=1}^{n} c_i \frac{f'(x)}{f(x)} \equiv 0 \mod g(x) \right\} \tag{2.15}$$

where

$$f(x) = \prod_{i \in \text{supp}(\mathbf{c})} (x - \alpha_i) \tag{2.16}$$

and $f'(x)$ the formal derivative of $f(x)$ with

$$f'(x) = \sum_{i \in \text{supp}(\mathbf{c})} \prod_{\substack{j \in \text{supp}(\mathbf{c}), \\ j \neq i}} (x - \alpha_j) \tag{2.17}$$

such that

$$\sum_{i=1}^{n} c_i \frac{f'(x)}{f(x)} = \sum_{i=1}^{n} c_i \frac{1}{x - \alpha_i}. \tag{2.18}$$

We call a binary Goppa code separable, if the binary Goppa polynomial $g(x)$ is separable, meaning $g(x)$ has no multiple zeros and has only distinct roots. Then, the minimum distance of the code is at least $2 \cdot \deg(g(x)) + 1$. A proof can be found in [Gop70] and [MS83, p.341-342]. The parity-check matrix of separable Goppa codes is a Cauchy matrix, where the elements are $h_{ij} = \frac{1}{x_i - \alpha_j}$ for $i = 1, \dots, t$ and $j = 1, \dots, n$ with x_i the distinct roots of $g(x) = (x - x_1)(x - x_2) \cdots (x - x_t)$ [MS83, p.345-346]. Binary irreducible Goppa codes also share the property of binary separable Goppa codes, with $d \geqslant 2 \cdot \deg(g(x)) + 1$ [MS83, p.345, eq.21].

2.1.4 Syndrome-Based Unique Decoding

Every error pattern that has Hamming weight less than half of the minimum distance of the code can uniquely be decoded. All unique decoders follow the same main idea in uniquely correcting an erroneous received word $\mathbf{r} \in \mathbb{F}_{q^n}$ with up to $\left\lfloor \frac{d-1}{2} \right\rfloor$ errors to a valid codeword $\mathbf{c}$.

Goppa codes are Alternant codes that are subfield subcodes of GRS codes. Therefore, decoders that decode GRS codes and codes of the family can also be used to decode Goppa codes. However, that might not fully use the capability of the Goppa code, as the minimum distance of the Goppa code might be larger than the minimum

Syndrome-based unique decoding $(\mathrm{wt_H}(\mathbf{e}) \leqslant \left\lfloor \frac{d-1}{2} \right\rfloor)$**:**

1. Syndrome computation: $\mathbf{s} = \mathbf{r}\mathbf{H}^\top$
2. Find error locator polynomial (ELP) $\sigma_e(x) = \prod_{i\in\mathcal{E}}(x-\alpha_i)$ and error evaluator polynomial (EEP) $\eta(x) := \sum_{i\in\mathcal{E}} e_i \prod_{\kappa\in\mathcal{E}\setminus\{i\}}(x-\alpha_\kappa)$
3. Find error locations $\mathcal{E}$ and error values e_i

distance of the GRS code.

The main property of syndrome-based decoders are, that they are solving a key equation that will be introduced in Eq. (2.28). A simple algorithm to solve the key equation is based on Peterson [Pet60] and Gorenstein, Zierler [GZ61]. More efficient decoders are for example the Sugiyama algorithm based on Euclid's algorithm [SKST75] or the Patterson algorithm [Pat75]. The probably fastest syndrome-based decoder is the Berlekamp-Massey algorithm that works on shift registers [Mas69a].

An alternative approach to syndrome-based unique decoders, are interpolation-based unique decoders, e.g. the Welch-Berlekamp decoder, that directly calculate the information vector $\mathbf{u}$ by solving a linear system of equations and subsequent factorising to receive $\mathbf{u}$. Usually, interpolation-based decoder are slower in terms of computation time as syndrome-based decoders, because the linear system of equations is usually bigger than for syndrome-based decoders.

For syndrome-based decoders, the goal is to find the error vector $\mathbf{e} \in \mathbb{F}_{q^n}$ with error positions and error values using only the received word and a parity-check matrix. It is handy to define an Error Locator Polynomial (ELP) and an Error Evaluator Polynomial (EEP) that can be constructed from the received word $\mathbf{r}$ and a parity-check matrix. First, the syndrome $\mathbf{s}$ of the code is calculated by multiplying the received word $\mathbf{r}$ with a parity-check matrix $\mathbf{H}$ of the code, such that

$$\mathbf{s} = \mathbf{r}\mathbf{H}^\top = (\mathbf{c}+\mathbf{e})\mathbf{H}^\top = \mathbf{e}\mathbf{H}^\top. \tag{2.19}$$

Second, the ELP and EEP are obtained using the syndrome. Third, the locations and values of the errors are calculated and the errors are corrected.

Suppose, that $\mathrm{wt}(\mathbf{e}) = w_e$. Let us denote the error vector $\mathbf{e} = (e_1, \dots, e_n)$ with e_i the error value at position i for $i = 1, \dots, n$ and the error locations with $\mathcal{E} := \mathrm{supp}(\mathbf{e}) = \{i : e_i \neq 0, i = 0, \dots, n\}$ and $|\mathcal{E}| = w_e$. The syndrome $\mathbf{s} = (s_1, s_2, \dots, s_{d-1})$ can be represented as a polynomial $s(x)$ of degree $d-2$ with

$$s(x) := \sum_{j=1}^{d-1} s_j x^{j-1} \tag{2.20}$$

and

$$s_j = \sum_{i=1}^{n} r_i \frac{\alpha_i^{j-1}}{g(\alpha_i)} = \sum_{i=1}^{n} e_i \frac{\alpha_i^{j-1}}{g(\alpha_i)} = \sum_{i\in\mathcal{E}} e_i \frac{\alpha_i^{j-1}}{g(\alpha_i)}. \tag{2.21}$$

Using the definition of a Goppa code in Eq. (2.12) a syndrome polynomial can also be written as

$$s(x) \equiv \sum_{i=1}^{n} \frac{e_i}{x - \alpha_i} \mod g(x) \tag{2.22}$$

in the ring of residues of the polynomials $\mathbb{F}_{p^m}[x]$ modulo $g(x)$. The goal is to use the syndrome to obtain an ELP and an EEP. By defining the Error Locator Polynomial (ELP) as

$$\sigma(x) := \prod_{i \in \mathcal{E}} (x - \alpha_i) \quad \in \mathbb{F}_{p^m}[x] \ , \tag{2.23}$$

the roots of $\sigma(x)$ gives us the error locations. Thus, if

$$\sigma(\alpha_\mu) = 0 \ , \tag{2.24}$$

then $\mu \in \mathcal{E}$. All elements of $\mathcal{L}$ can be checked if they are a root of the ELP. If there is a root among $\mathcal{L}$, the index μ is an error position. An efficient method to find the roots of the ELP is called "Chien search" [Rot06, Sec. 6.3].
By defining the Error Evaluator Polynomial (EEP) as

$$\eta(x) := \sum_{i \in \mathcal{E}} e_i \prod_{\kappa \in \mathcal{E} \setminus \{i\}} (x - \alpha_\kappa) \quad \in \mathbb{F}_{p^m}[x] \ , \tag{2.25}$$

then $\eta(\alpha_\mu) \neq 0$ if $\mu \in \mathcal{E}$. From the definitions it follows that $\sigma(x)$ and $\eta(x)$ have no common factors and

$$\gcd(\sigma(x), \eta(x)) = 1. \tag{2.26}$$

If $\eta(x) = 0$ and $\sigma(x) = 1$, no error has occurred in $\mathbf{r}$ and $s(x) = 0$. The degree of $\sigma(x)$ is exactly w_e and the degree of $\eta(x)$ is less than w_e. Since $w_e \leqslant \left\lfloor \frac{d-1}{2} \right\rfloor$ we have

$$\deg \eta(x) < \deg \sigma(x) \leqslant \left\lfloor \frac{d-1}{2} \right\rfloor . \tag{2.27}$$

The ELP and EEP are defined such that they are related to the syndrome. The following relation holds

$$\sigma(x) s(x) \equiv \eta(x) \mod g(x) \tag{2.28}$$

and is used to find the polynomials knowing only the syndrome. Together with equations (2.26) and (2.27) this forms a key equation to be solved for decoding. Solving the key equation in (2.28) is the core calculation of decoding. There exist many different algorithms that solve the key equation, essentially transforming it into a system of linear equations [Rot06, Chap.6.3.1].

2.2 Extended Euclidean Algorithm (EEA)

This section introduces the polynomial Extended Euclidean Algorithm (EEA) which is used in Chapter 4 to derive a parity-check matrix and to solve the key equation for

generalized Goppa codes. The EEA is an extension of the Euclidean algorithm which can be found in many textbooks e.g. [Rot06, p.192] or [HP03, p.102]. It is an efficient method for computing the greatest common divisor of two numbers or polynomials and the coefficients of Bézout's identity (see below).

The polynomial EEA calculates the polynomial greatest common divisor (gcd) $d(x)$ of two univariate polynomials $a(x), b(x) \in \mathbb{F}_q[x]$ where $\deg a(x) > \deg b(x)$. Additionally, it calculates besides $d(x)$ also the polynomials $w(x)$ and $v(x)$ satisfying Bèzout's identity

$$d(x) = \gcd(a(x), b(x)) = a(x)w(x) + b(x)v(x). \tag{2.29}$$

The polynomial greatest common divisor is generally defined up to the multiplication by a non-zero constant. Therefore, multiple polynomial greatest common dividers exist for the same polynomials $a(x), b(x)$. In Algorithm 1 we define the polynomial greatest common divider output as a monic polynomial. This makes the output well-defined and unique. In Algorithm 2, uniqueness is ensured by a requirement on the last computed remainder.

A call to Algorithm 1 is denoted as $(d(x), w(x), v(x)) = \text{EEA}\,(a(x), b(x))$ with inputs and outputs order. We assume $a(x) \neq 0$ (the dividend). The initial values of the algorithm are set to $r_0(x) = a(x)$ and $r_1(x) = b(x)$. The algorithm starts at $i = 2$ and stops at step $\nu + 1$ with ν the largest index for which $r_i(x) \neq 0$. The polynomial $r_i(x)$ denotes the remainder and $q_i(x)$ denotes the quotient of the polynomial division in step i. For every i we have

$$r_i(x) = a(x)w_i(x) + b(x)v_i(x).$$

It applies that

$$\deg(r_i(x)) < \deg(r_{i-1}(x)) \tag{2.30}$$

and

$$\deg v_i(x) + \deg r_{i-1}(x) = \deg a(x) \tag{2.31}$$

for $i = 1, \ldots, \nu + 1$. From these two relations it follows that $\deg(v_i(x)) + \deg(r_i(x)) < \deg(a(x))$ and from (2.30) it follows that $\deg(r_i) < \deg(a(x))$ for $i = 1, \ldots, \nu + 1$. Additionally,

$$\gcd(r_0, r_1) = \gcd(r_1, r_2) = \cdots = \gcd(r_\nu, 0) = r_\nu.$$

In the special case of $\gcd(a(x), b(x)) = 1$ the multiplicative inverse $b(x)^{-1} \mod a(x)$ is $v(x)$ since

$$\begin{aligned} 1 &= a(x)w(x) + b(x)v(x) \\ 1 &\equiv b(x)v(x) \mod a(x) \\ 1 &\equiv b(x)b(x)^{-1} \mod a(x) \end{aligned} \tag{2.32}$$

and thus the multiplicative inverse can be found by the EEA. The EEA can also be

Algorithm 1 Extended Euclidean Algorithm (EEA)

Input: polynomial $a(x), b(x)$
1: $r_0(x) \leftarrow a(x),\ r_1(x) \leftarrow b(x)$
2: $w_0(x) \leftarrow 1,\ w_1(x) \leftarrow 0$
3: $v_0(x) \leftarrow 0,\ v_1(x) \leftarrow 1$
4: $i \leftarrow 2$
5: **while** $r_{i-1}(x) \neq 0$ **do**
6: $\quad q_i(x) \leftarrow$ quotient of $r_{i-2}(x)/r_{i-1}(x)$
7: $\quad r_i(x) \leftarrow$ remainder of $r_{i-2}(x)/r_{i-1}(x) = r_{i-2}(x) - q_i(x)r_{i-1}(x)$
8: $\quad w_i(x) \leftarrow w_{i-2}(x) - q_i(x)w_{i-1}(x)$
9: $\quad v_i(x) \leftarrow v_{i-2}(x) - q_i(x)v_{i-1}(x)$
10: $\quad i \leftarrow i + 1$
11: $c \leftarrow$ the leading coefficient of $r_{i-2}(x)$
12: $d(x) \leftarrow$ monic polynomial $r_{i-2}(x)/c$
13: $w(x) \leftarrow w_{i-2}(x)/c$
14: $v(x) = v_{i-2}(x)/c$

Output: $d(x), w(x), v(x)$

used to calculate values $v(x)$ and $d(x)$ for known polynomials $a(x), b(x)$ satisfying

$$d(x) \equiv b(x)v(x) \mod a(x) \tag{2.33}$$

which is needed for decoding Goppa codes (see Eq. (2.28)) and Generalized Goppa Codes (GGC) in Section 4.1.2.

The EEA can also be used for solving the key equation as described in [Rot06, Sec. 6.4]. Algorithm 2 reflects the steps necessary to determine the ELP $\sigma(x)$ and the EEP $\eta(x)$ knowing the syndrome polynomial $s(x)$ and the Goppa polynomial $g(x)$ of an $\Gamma(\mathcal{L}, g)$)-code. In contrast to Algorithm 1, Algorithm 2 stops at Line 5 when the degree of $r_{i-1}(x)$ falls below $\frac{\deg(g(x))}{2}$ such that

$$\deg(r_{i-1}(x)) < \frac{\deg g(x)}{2} \leqslant \deg(r_{i-2}(x)). \tag{2.34}$$

We denote the value of $i - 1$ when the algorithm stopped, with μ. Then the outputs $r_{\mu(x)}$ and $v_\mu(x)$ are unique [Rot06, Prop. 6.4] and $\sigma(x) = r_{\mu(x)}/c$ and $\nu(x) = v_\mu(x)/c$. The outputs satisfy the properties for the key equation (2.26)-(2.28) with [Rot06, Prop. 6.3]

$$\gcd(c^{-1}v_\mu(x), c^{-1}r_\mu(x)) = 1 \tag{2.35}$$

$$\deg(c^{-1}v_\mu(x)) + \deg(c^{-1}r_\mu(x)) < \deg(g(x)) \tag{2.36}$$

$$c^{-1}v_\mu(x)s(x) \equiv c^{-1}r_\mu(x) \mod g(x). \tag{2.37}$$

Algorithm 2 Extended Euclidean Algorithm (EEA) for Solving the Key Equation

Input: polynomial $g(x), s(x)$
1: $r_0(x) \leftarrow g(x),\ r_1(x) \leftarrow s(x)$
2: $w_0(x) \leftarrow 1,\ w_1(x) \leftarrow 0$
3: $v_0(x) \leftarrow 0,\ v_1(x) \leftarrow 1$
4: $i \leftarrow 2$
5: **while** $r_{i-1}(x) \neq 0$ and $\deg(r_{i-1}(x)) \geqslant \deg(g(x))/2$ **do**
6: $\quad q_i(x) \leftarrow$ quotient of $r_{i-2}(x)/r_{i-1}(x)$
7: $\quad r_i(x) \leftarrow$ remainder of $r_{i-2}(x)/r_{i-1}(x) = r_{i-2}(x) - q_i(x)r_{i-1}(x)$
8: $\quad w_i(x) \leftarrow w_{i-2}(x) - q_i(x)w_{i-1}(x)$
9: $\quad v_i(x) \leftarrow v_{i-2}(x) - q_i(x)v_{i-1}(x)$
10: $\quad i \leftarrow i + 1$
11: $c \leftarrow$ the constant coefficient of $r_{i-1}(x)$, if it exists
12: $r_\mu(x) \leftarrow r_{i-1}(x)/c$
13: $w_\mu(x) \leftarrow w_{i-1}(x)/c$
14: $v_\mu(x) \leftarrow v_{i-1}(x)/c$
Output: $r_\mu(x), w_\mu(x), v_\mu(x)$

2.3 Syndrome Decoding Problem (SDP)

In decoding, the goal is to find the closest codeword $\mathbf{c}$ to a given noisy received word $\mathbf{r}$. In other words for $\mathbf{r} = \mathbf{c} + \mathbf{e}$ and $\mathbf{c}$ a codeword, the goal is to find the error $\mathbf{e}$ with $\mathrm{wt}(\mathbf{e}) = w_e$. This problem is equivalent to the problem of syndrome decoding in Definition 2.7.

Definition 2.7 (Syndrome Decoding Problem).
For a given parity-check matrix $\mathbf{H} \in \mathbb{F}_q^{(n-k)\times n}$ with rank $n-k$ and syndrome $\mathbf{s} \in \mathbb{F}_q^{n-k}$, find a vector $\mathbf{e}$ with Hamming weight $\mathrm{wt}(\mathbf{e}) = w_e$ such that $\mathbf{s} = \mathbf{e}\mathbf{H}^\top$.

Basically, the SDP is like solving a linear system of equations with a non-linear constraint for the solution. Were it not for the constraint of the Hamming weight $\mathrm{wt}(\mathbf{e}) = w_e$, then the task would be easily solved by matrix inversion and thus Gaussian elimination. There exist some codes and decoding distances w_e for which the problem is easy to solve (e.g. the constraint of $w_e = 1$ is easy to solve). For well-chosen codes and w_e the problem is hard to solve.

For estimating the complexity of the SDP Definition 2.7 is rephrased in a decisional problem in Definition 2.8 (yes-no question).

Definition 2.8 (Decisional Syndrome Decoding Problem).
For inputs $\mathbf{H} \in \mathbb{F}_q^{(n-k)\times n}$ and $\mathbf{s} \in \mathbb{F}_q^{1\times n-k}$ where n, k, w_e are positive integers with $k \leqslant n$ and integer $w_e \leqslant n$, decide whether there exists an $\mathbf{e} \in \mathbb{F}_q^n$ of Hamming weight w_e such that $\mathbf{s} = \mathbf{e}\mathbf{H}^\top$.

In 1978, Berlekamp [BMT78] showed that the Decisional Syndrome Decoding Prob-

lem for $q = 2$ is in the complexity class of `NP-complete`. And in 1994, Barg [Bar94; Bar97] showed the `NP-completeness` over all finite fields. `NP-complete` is a term from computational complexity theory meaning that for a Turing machine (mathematical model of a general classical computer) the complexity for solving the task lies in the class of `NP` (Non-Deterministic Polynomial) and is the hardest problem in this class. The `NP-complete` class belongs to the `EXP` (Exponential) class without the `P` (Polynomial) class (assuming `P`$\neq$`NP`), meaning that the syndrome decoding problem cannot be solved in polynomial time by a Turing machine for all inputs. Generally speaking, it is hard to solve a problem that lies in `NP-complete`. The syndrome decoding problem is also hard to solve for quantum computers, although there exist quantum algorithms that bring speed-ups [CDE21; BR22].

The security of code-based cryptography is build on the hardness of solving the SDP when, for example, parity-check matrices $\mathbf{H}$ and error vectors $\mathbf{e}$ are uniformly chosen at random. The security of such a cryptosystem relies then on the hardness of the decoding problem when the input is a random linear code.

Definition 2.9 (Cryptographic Decoding Problem).
For inputs $\mathbf{H}$ *and* $\mathbf{s}$ *with* $\mathbf{H}$ *a uniformly at random chosen parity-check matrix* $\mathbf{H} \xleftarrow{\$} \mathbb{F}_q^{(n-k)\times n}$ *and vector* $\mathbf{x} \xleftarrow{\$} \mathbb{F}_q^{1\times n}$ *such that* $\mathbf{s} = \mathbf{x}\mathbf{H}^\top$ *with* $\mathrm{wt}(\mathbf{x}) = w_e$*, find an error* $\mathbf{e} \in \mathbb{F}_q^{1\times n}$ *of Hamming weight* $\mathrm{wt}(\mathbf{e}) = w_e$ *such that* $\mathbf{s} = \mathbf{e}\mathbf{H}^\top$*.*

The task of finding a vector $\mathbf{e}$ does not necessarily mean to recover exactly $\mathbf{x}$. For cryptography it is enough to find an $\mathbf{e}$ satisfying the conditions in Definition 2.9. However, if w_e is smaller than half of the minimum distance of the chosen code, the solution is unique.

The hardness of the SDP can be used to construct public-key cryptosystems. The best known algorithms to solve the SDP are based on Information Set Decoding (ISD) (see also Section 3.5.2).

Chapter 3

Introduction to the McEliece Cryptosystem and its Variations

The Classic McEliece cryptosystem [BCL+20] is an asymmetric (public-key) Key Encapsulation Mechanism (KEM) based on the McEliece and Niederreiter cryptosystem [McE78; Nie86] using Goppa codes as the underlying linear code. It is believed to be resistant against attacks from quantum computers.

The McEliece and Niederreiter cryptosystem have been adopted by Bernstein et al. [BCL+20] to design the Classic McEliece cryptosystem for submission to the NIST post-quantum cryptography competition as a KEM meeting the requirements set by NIST (see [Nat16]). A KEM is a cryptographic protocol that uses asymmetric algorithms to transmit symmetric keys for further communication via symmetric cryptography. The symmetric keys can be randomly generated by the asymmetric algorithm. Thus, the secret message in Classic McEliece is randomly chosen. As a requirement for KEMs, these cryptosystems need to be designed to meet the resistance requirements against adaptive chosen ciphertext attacks (IND-CCA2) and chosen plaintext attacks (IND-CPA) (more in Section 3.4). Compared to other NIST candidates Classic McEliece is very efficient in encoding and decoding. Its drawback is that it needs very large public keys to achieve the desired security levels.

The underlying McEliece Public-Key Encryption (PKE) is the oldest known and well-researched cryptosystems among the PQC systems submitted to NIST. It was first described by McEliece in 1978 [McE78] using classical Goppa codes. The security of the system is based on the hardness of the syndrome decoding problem (SDP) for error-correcting codes, which is discussed in Section 2.3. It corresponds to the hardness of decoding a random linear code. Without the secret key, the linear code looks like a random code and thus one would need to solve the hard SDP. Only if the algebraic structure of an error-correcting code is known, then there exist fast decoders for decoding. A legitimate user knows the algebraic structure using its secret key and has therefore access to an efficient decoder. The original cryptosystem of McEliece consists of a scrambled public key that is left multiplied by an invertible matrix and right multiplied by a permutation matrix, but in Classic McEliece scrambling is achieved by calculating the systematic form using Gaussian elimination.

In 1986, Niederreiter [Nie86] introduced a new version of the McEliece public-key

cryptosystem. The main difference between McEliece and Niederreiter is, that in McEliece's cryptosystem the public key consists of a generator matrix of a code and in Niederreiter's cryptosystem the public key consists of a parity-check matrix. The McEliece cryptosystem and the Niederreiter cryptosystem based on Goppa codes have the same security when set up with the same parameters [LDW94]. In Classic McEliece, the Niederreiter cryptosystem is implemented because it gives better efficiency in computations, e.g. the parity-check matrix of the Goppa code can be directly set up at construction and there is no need to compute a generator matrix and reverse to a parity-check matrix for decoding. Therefore, in Classic McEliece the public key is represented with the parity-check matrix.

The secret key of Classic McEliece consists of the Goppa polynomial and the code locators of a Goppa code (see also Section 2.1.3). The chosen Goppa code needs to be kept secret. The public key is a scrambled version of the parity-check matrix constructed using the secret key and Gaussian elimination that destroys the structure in the matrix. For encryption, a random vector $\mathbf{x}$ with Hamming weight $\mathrm{wt}(\mathbf{x}) = w_e$ is chosen and multiplied with the public key. The ciphertext is then a syndrome. For decryption, the syndrome can be efficiently decoded using the secret key (see also Section 2.1.4).

Several authors proposed to replace the binary Goppa codes in the McEliece cryptosystem with other linear codes from different families, e.g. GRS codes [Nie86], Reed-Muller codes [Sid94], Quasi-Cylic Low Density Parity Check (QC-LDPC) [BCGM07] often with the goal of reducing the public key size. However, many of them turned out to be broken and insecure due to structural attacks that recover the secret key from the public key by distinguishing the code from a random code. For example, McEliece with GRS codes was proposed in 1986 by Niederreiter [Nie86] and broken by Sidelnikov & Shestakov in 1992 [SS92]. More details can be found in Section 3.5.1. So far, the McEliece cryptosystem using Goppa codes remains and keeps still its security and is assumed to be resistant against structural attacks if parameters are chosen accordingly.

In summary, the key strength of Classic McEliece is its long history of research on its security. It is very efficient in encoding and decoding, but it has a large public key size compared to other KEMs, which is a drawback in computation time and storage space.

In the following sections, the Classic McEliece cryptosystem is described. In Chapter 5 the work depends on the Classic McEliece implementation submitted for NIST competition Round 2 [BCL+19], and in Chapter 6 on the implementation submitted for Round 3 [BCL+20].

3.1 McEliece Cryptosystem

The McEliece cryptosystem first described by McEliece in 1978 [McE78] uses classical Goppa codes. It is a Public-Key Encryption (PKE) system. The secret key consists of a generator matrix $\mathbf{G}$, a scrambling marix $\mathbf{S}$ and a permutation matrix $\mathbf{P}$. The public key is a scrambled and permuted version of the generator matrix from the secret key.

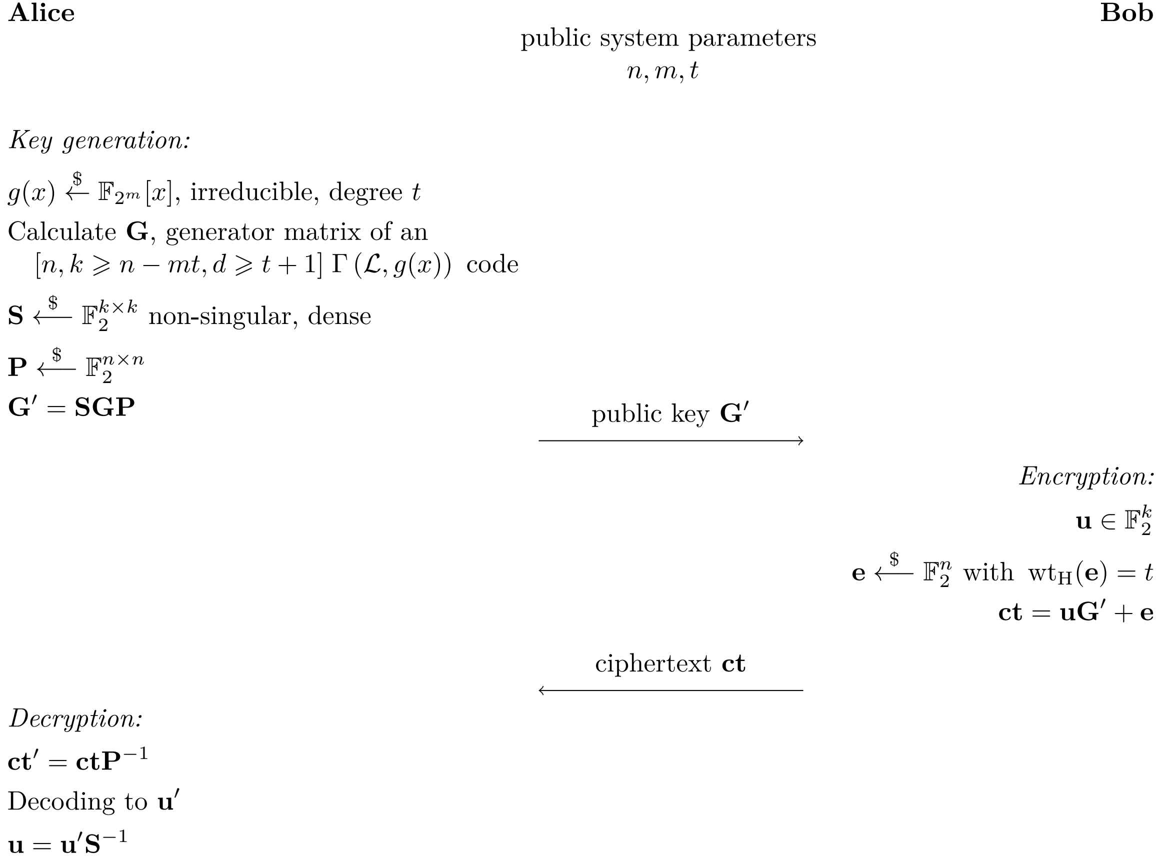


Figure 3.1: Summary of the McEliece cryptosystem in a protocol view. Alice is generating the keys. Bob encrypts the secret message $\mathbf{u}$ and sends it in encrypted form to Alice. Only Alice with her secret key can decrypt the ciphertext $\mathbf{ct}$ to obtain the plaintext of the secret message of Bob.

For the key generation, a binary irreducible Goppa code of code length n and dimension k with a random irreducible Goppa polynomial of degree t over $\mathbb{F}_{2^m}$ is chosen. The parameters n, t, m are publicly known. A generator matrix $\mathbf{G}$ of size $k \times n$ for the previous chosen code, a random dense $k \times k$ non-singular matrix $\mathbf{S}$ and a random $n \times n$ permutation matrix is produced. These matrices are kept private. The public key is calculated with $\mathbf{G}' = \mathbf{SGP}$. This public generator matrix $\mathbf{G}'$ generates a linear code with the same rate and minimum distance as the code generated by $\mathbf{G}$.

For encryption, a plaintext message of length k, denoted by $\mathbf{u}$, is first multiplied by the public generator matrix $\mathbf{G}' \in \mathbb{F}_2^{k \times n}$. Then, a random vector $\mathbf{e}$ of length n and Hamming weight t is added to the product, such that the ciphertext is $\mathbf{ct} = \mathbf{uG}' + \mathbf{e}$ and has size n.

For decryption, the ciphertext is right multiplied by the inverse of the permutation matrix $\mathbf{P}^{-1}$, such that $\mathbf{ct}' = \mathbf{ctP}^{-1}$. The resulting vector $\mathbf{ct}'$ is then an erroneous codeword in the previously chosen Goppa code and can be decoded to $\mathbf{u}'$ using an efficient decoding algorithm. The original plaintext $\mathbf{u}$ is obtained by computing $\mathbf{u} = \mathbf{u}'\mathbf{S}^{-1}$.

A summary is given in Fig. 3.1, where Alice and Bob are two interlocutors that communicate via encrypted messages.

3.2 Niederreiter Cryptosystem

The Niederreiter cryptosystem [Nie86] is also a PKE system and by using Goppa codes as the underlying linear error-correcting code, it is the dual version of the McEliece cryptosystem. The secret key consists of a parity-check matrix, a scrambling matrix $\mathbf{M}$ and a permutation matrix $\mathbf{P}$. The public key is a scrambled and permuted version of the parity-check matrix of the secret key.

For key generation, a linear error-correcting code of length n, dimension k and error-correction capability t is chosen. The code length n and the extension degree m is publicly known. A parity-check matrix of the chosen code is computed. Let $\mathbf{M}$ be a randomly chosen non-singular matrix of size $(n-k) \times (n-k)$ and $\mathbf{P}$ a randomly chosen permutation matrix of size $n \times n$ that can be generated by permuting the rows of a non-singular diagonal matrix. To compute the public key $\mathbf{K}$, the secret parity-check matrix is left multiplied by the scrambling matrix $\mathbf{M}$ and right multiplied by the permutation matrix $\mathbf{P}$, such that $\mathbf{K} = \mathbf{MHP}$.

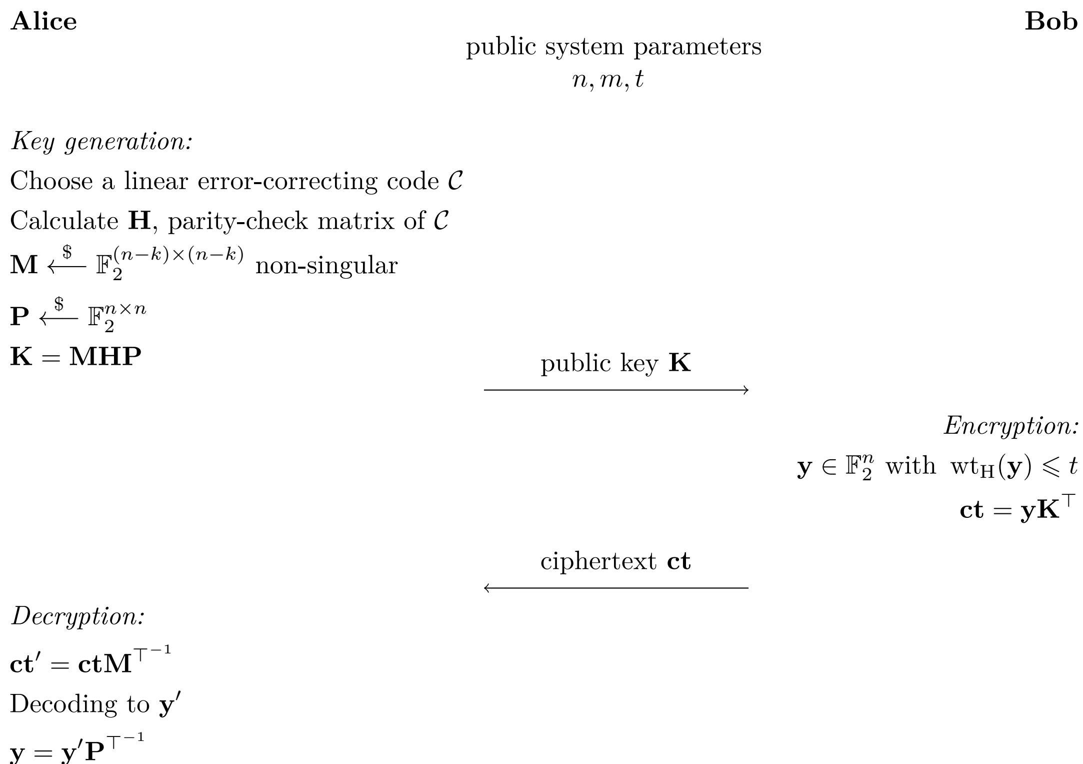


Figure 3.2: Summary of the Niederreiter cryptosystem in a protocol view. Alice is generating the keys. Bob encrypts the secret message $\mathbf{y}$ and sends it in encrypted form to Alice. Only Alice with her secret key can decrypt the ciphertext $\mathbf{ct}$ to obtain the plaintext of the secret message of Bob.

For encryption, a plaintext message $\mathbf{y}$ of length n with Hamming weight $\mathrm{wt}_{\mathrm{H}}(\mathbf{y}) \leqslant t$ is multiplied by the public parity-check matrix $\mathbf{K} \in \mathbb{F}_2^{(n-k)\times n}$, such that the ciphertext is $\mathbf{ct} = \mathbf{y}\mathbf{K}^\top$ and has size $(n-k)$.

For decryption, the ciphertext is right multiplied by the inverse of the transposed scrambling matrix $\mathbf{M}^{\top^{-1}}$, such that $\mathbf{ct}' = \mathbf{ct}\mathbf{M}^{\top^{-1}}$. The resulting vector $\mathbf{ct}'$ corresponds to $\mathbf{y}\mathbf{P}^\top\mathbf{H}^\top = \mathbf{y}'\mathbf{H}^\top$ and $\mathbf{y}'$ is found by using an efficient decoding algorithm. The original plaintext $\mathbf{y}$ is obtained by computing $\mathbf{y} = \mathbf{y}'\mathbf{P}^{\top^{-1}}$.

It has been shown that the original proposed Niederreiter cryptosystem using Reed-Solomon codes is vulnerable to structural attacks (see also Section 3.5.1). However, using Goppa codes as the underlying linear error-correcting code, it is still today unbroken.

A summary is given in Fig. 3.2, where Alice and Bob are two interlocutors that communicate via encrypted messages.

3.3 Classic McEliece Cryptosystem

Classic McEliece is a Key Encapsulation Mechanism (KEM) that is build on the Niederreiter cryptosystem using Goppa codes as the underlying error-correcting code. The encryption and decryption algorithms are incorporated into an encapsulation and decapsulation algorithm, which generate and recover a secret symmetric key to be used in further communication over a symmetric cryptosystem. This symmetric key is also called shared secret or session key, as for every new connection a new symmetric key is exchanged using the KEM.

The cryptosystem implements three main algorithms: key generation, encapsulation with encryption and decapsulation with decryption. The key generation algorithm produces a key pair consisting of public and secret (private) key. The encapsulation algorithm chooses a random vector, calculates its ciphertext and generates a secret session key from the combination of both. The decapsulation algorithm decrypts the ciphertext and recovers the secret session key.

We represent here the "Model Classic McEliece" as described in [BCL+20] for better readability. The description of the actual implementation with representation in bits is given where it is necessary.

The hash function that is used in the cryptosystem is the SHA-3 Keccak SHAKE-256 hash function defined in [Nat15] and is denoted by $\mathcal{H}$, whose output is the first 256 bits of SHAKE256(x), independent of its input length. In particular, we write $\mathcal{H}(2, \mathbf{v})$ and $\mathcal{H}(i, \mathbf{v}, C)$ for the hash of the concatenation of an initial byte valued $i \in \{0, 1\}$ or 2, vector $\mathbf{v} \in \mathbb{F}_2^n$ and ciphertext C, see also [BCL+20, Sec. 2.5.2].

3.3.1 Key generation

The goal of the key generation is to generate a secret (private) key and the corresponding public key. In our model the secret key is a tuple $(\mathbf{s}, \gamma)$ where $\mathbf{s}$ is a bit-vector in $\mathbb{F}_2^n$ and $\gamma = (\, g(x), \mathcal{L}\,)$ a generator tuple consisting of the monic irreducible Goppa polynomial $g(x) \in \mathbb{F}_{2^m}[x]$ of degree t and the support $\mathcal{L} = (\alpha_1, \ldots, \alpha_n)$ with code

locators α_i for $i = 1, \ldots, n$. Then, γ defines the binary irreducible Goppa code (see also Eq. (2.12))

$$\Gamma(\mathcal{L}, g) = \left\{ (c_1, \ldots, c_n) \in \mathbb{F}_2^n \mid \sum_{i \in \mathrm{supp}(c)} \frac{1}{(x - \alpha_i)} = 0 \text{ in } \mathbb{F}_{2^m}[x]/g(x) \right\} \subseteq \mathbb{F}_2^n \tag{3.1}$$

with $\alpha_i \neq \alpha_j$ for $i \neq j$ and $g(\alpha_i) \neq 0$. In the actual implementation the secret key does not contain the support $\mathcal{L}$ explicitly, but instead the seed of the random function that is used to generate it.

Algorithm 3 presents the construction of the secret and public keys in Classic McEliece. The public key is given by the matrix $\mathbf{T} \in \mathbb{F}_2^{(n-k) \times k}$, that is computed via a parity-check matrix in systematic form

$$\mathbf{H}_{\mathrm{sys}} = (\mathbf{I}_{n-k} | \mathbf{T}) \tag{3.2}$$

where $\mathbf{I}_{n-k}$ is the identity matrix of size $n - k$. If $\mathbf{H}$ has not full rank, then the systematic form cannot be computed in Line 5 of Algorithm 3 and the key generation algorithm needs to start over with a new tuple γ. On average this takes about three trials. This ensures that the Goppa code $\Gamma(\mathcal{L}, g)$ has dimension $k = n - mt$ with code length n and allows efficient correction of up to t errors.

Note, that in particular the code $\Gamma(\mathcal{L}, g)$ itself is public knowledge since $\mathbf{T}$ is public. However, the algebraic structure, i.e. the Goppa polynomial and the support, are part of the secret key. The Gaussian elimination used for computing the systematic form destroys the structure of the code in the parity-check matrix such that $\mathbf{T}$ looks like a random matrix.

3.3.2 Encapsulation

For encapsulation, a random vector $\mathbf{e} \in \mathbb{F}_2^n$ of Hamming weight t is generated. The ciphertext $C = (\mathbf{c_0}, C_1)$ is generated by encoding the vector $\mathbf{e}$ using the public key such that $\mathbf{c_0} = \mathbf{e}{\mathbf{H}_{\mathrm{sys}}}^\top$ and by calculating the hash $C_1 = \mathcal{H}(2, \mathbf{e})$. The secret session key K is then the hash $\mathcal{H}(1, \mathbf{e}, C)$.

The encapsulating party uses the public key and the random vector as plaintext to generate a ciphertext from which only the holder of the secret key can extract the random plaintext again. This can be used to establish a common secret session key. Details can be found in Algorithm 4 and Algorithm 5.

3.3.3 Decapsulation

In Algorithm 6, the session key K is reconstructed by the decapsulating party holding the secret key. Decapsulation is done by splitting the received ciphertext $C = (\mathbf{c_0}, C_1)$ in parts $\mathbf{c_0} \in \mathbb{F}_2^{n-k}$ and the hash C_1. $\mathbf{c_0}$ is decoded to a vector $\mathbf{e}' \in \mathbb{F}_2^n$ of weight t using the secret γ of the Goppa code. Then the result is verified by calculating $C_1' = \mathcal{H}(2, \mathbf{e}')$ and checking if C_1' and C_1 are equal. If no error occurred and C_1' and C_1 match, then the output is the reconstructed session key $K' = \mathcal{H}(1, \mathbf{e}', C)$ and K is equal to K'. The

Algorithm 3 Key Generation

Input: parameters $m, t, n \leq 2^m$, $f(z) \in \mathbb{F}_2[z]$ irreducible of degree m.
Output: secret key $(\mathbf{s}_\mathsf{r}, \gamma)$, public key $\mathbf{T}$.

1: Generate a uniformly random monic irreducible polynomial $g(x) \in \mathbb{F}_{2^m}[x]$ of degree t.
2: Select a uniform random sequence $\mathcal{L} = (\alpha_1, \alpha_2, \ldots, \alpha_n)$ of n distinct elements in $\mathbb{F}_{2^m}$.
3: Compute the $t \times n$ matrix $\mathbf{H} = \{h_{ij}\}$ over $\mathbb{F}_{2^m}$ where $h_{ij} = \alpha_j^{i-1}/g(\alpha_j)$ for $i \in [t], j \in [n]$, i.e.

$$\mathbf{H} = \begin{pmatrix} \frac{1}{g(\alpha_1)} & \frac{1}{g(\alpha_2)} & \cdots & \frac{1}{g(\alpha_n)} \\ \frac{\alpha_1}{g(\alpha_1)} & \frac{\alpha_2}{g(\alpha_2)} & \cdots & \frac{\alpha_n}{g(\alpha_n)} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\alpha_1^{t-1}}{g(\alpha_1)} & \frac{\alpha_2^{t-1}}{g(\alpha_2)} & \cdots & \frac{\alpha_n^{t-1}}{g(\alpha_n)} \end{pmatrix}.$$

4: Form matrix $\hat{\mathbf{H}} \in \mathbb{F}_2^{mt \times n}$ by replacing each entry $c_0 + c_1 z + \ldots + c_{m-1} z^{m-1} \in \mathbb{F}_2[z]/\langle f(z) \rangle \cong \mathbb{F}_{2^m}$ of $\mathbf{H} \in \mathbb{F}_{2^m}^{t \times n}$ with a column of m bits $c_0, c_1, \ldots, c_{m-1}$.
5: Reduce $\hat{\mathbf{H}}$ to systematic form $(\mathbf{I}_{n-k}|\mathbf{T})$ where $\mathbf{I}_{n-k}$ is an identity matrix of $(n-k) \times (n-k)$ and $k = n - mt$.
6: If Step 5 fails, go back to Step 1 .
7: Generate a uniform random n-bit string $\mathbf{s}_\mathsf{r}$ (needed if decapsulation fails).
8: Output secret key: $(\mathbf{s}_\mathsf{r}, \gamma)$ with $\gamma = (g(x), \alpha_1, \alpha_2, \ldots, \alpha_n)$
9: Output public key: $\mathbf{T} \in \mathbb{F}_2^{(n-k) \times k}$

Algorithm 4 Encoding

Input: weight-t row vector $\mathbf{e} \in \mathbb{F}_2^n$, public key $\mathbf{T}$
Output: syndrome $\mathbf{c_0}$

1: Construct $\mathbf{H}_{\text{sys}} = (\mathbf{I}_{n-k}|\mathbf{T})$.
2: Compute and output $\mathbf{c_0} = \mathbf{e}{\mathbf{H}_{\text{sys}}}^\top \in \mathbb{F}_2^{n-k}$.

Algorithm 5 Encapsulation

Input: public key $\mathbf{T}$
Output: session key K, ciphertext C

1: Generate a uniform random vector $\mathbf{e} \in \mathbb{F}_2^n$ of Hamming weight t.
2: Use the encoding subroutine defined in Algorithm 4 on $\mathbf{e}$ and public key $\mathbf{T}$ to compute $\mathbf{c_0}$.
3: Compute $C_1 = \mathcal{H}(2, \mathbf{e})$ (input to hash function is a concatenation of 2 and $\mathbf{e}$ as a 1-byte and $\lceil n/8 \rceil$-byte string representation).
4: Put $C = (\mathbf{c_0}, C_1)$.
5: Compute $K = \mathcal{H}(1, \mathbf{e}, C)$ (input to hash function is a concatenation of 1, $\mathbf{e}$ and C as a 1-byte, $\lceil n/8 \rceil$-byte and $\lceil mt/8 \rceil + \lceil \ell/8 \rceil$ string representation).

Algorithm 6 Decapsulation

Input: ciphertext C, private key $(\mathbf{s}, \gamma)$
Output: session key K'
1: Split the ciphertext C as $(\mathbf{c_0}, C_1)$ with $\mathbf{c_0} \in \mathbb{F}_2^{n-k}$ and hash C_1.
2: Set $b \leftarrow 1$.
3: Use the decoding subroutine defined in Algorithm 7 on $\mathbf{c_0}$ and private key γ to compute $\mathbf{e}'$. If the subroutine returns Failure, set $\mathbf{e}' \leftarrow \mathbf{s}$ and $b \leftarrow 0$.
4: Compute $C_1' = \mathcal{H}(2, \mathbf{e}')$ (input to hash function is concatenation of 2 and $\mathbf{e}'$ as a 1-byte and $\lceil n/8 \rceil$-byte string representation).
5: If $C_1' \neq C_1$, set $\mathbf{e}' \leftarrow \mathbf{s}$ and $b \leftarrow 0$.
6: Compute $K' = \mathcal{H}(b, \mathbf{e}', C)$ (input to hash function is concatenation of b, $\mathbf{e}'$ and C as a 1-byte, $\lceil n/8 \rceil$-byte and $\lceil mt/8 \rceil + \lceil \ell/8 \rceil$ string representation).
7: Output session key K'.

Algorithm 7 Decoding

Input: vector $\mathbf{c_0} \in \mathbb{F}_2^{n-k}$, private key (s, γ)
Output: weight-t vector $\mathbf{e}' \in \mathbb{F}_2^n$ or Failure
1: Extend $\mathbf{c_0}$ to $\mathbf{v} = (\mathbf{c_0}, 0, ..., 0) \in \mathbb{F}_2^n$ by appending k zeros.
2: Find the unique codeword $\mathbf{c}$ in the Goppa code defined by γ that is at distance $\leq t$ from $\mathbf{v}$. If there is no such codeword, return failure.
3: Set $\mathbf{e}' = \mathbf{v} + \mathbf{c}$.
4: If $\mathrm{wt}(\mathbf{e}') = t$ and $\mathbf{c_0} = \mathbf{e}'\mathbf{H}_{\mathrm{sys}}{}^{\top}$, return $\mathbf{e}'$, otherwise return Failure.

encapsulating party and decapsulating party will then both know the same session key K. In case the input $\mathbf{c_0}$ or C_1 is no valid ciphertext, the decoding step will fail, or the plaintext confirmation of $C_1' = C_1$ will fail. In this case a predefined output $K' = \mathcal{H}(0, \mathbf{s}, C)$ is returned, where $\mathbf{s} \in \mathbb{F}_2^n$ is part of the secret key (see line 7 in Algorithm 3). These checks are implemented to protect against chosen-ciphertext attacks.

For decoding, the syndrome $\mathbf{c_0} = \mathbf{e}\mathbf{H}_{\mathrm{sys}}{}^{\top} \in \mathbb{F}_2^{n-k}$ is first appended with k zeros to form a received vector $\mathbf{v} = (\mathbf{c_0}, 0, \ldots, 0) \in \mathbb{F}_2^n$ in Algorithm 7 - Line 1. By construction, the syndrome of $\mathbf{v}$ with $\mathbf{H}_{\mathrm{sys}}$ is exactly $\mathbf{c_0}$ such that $\mathbf{c_0} = \mathbf{v}\mathbf{H}_{\mathrm{sys}}{}^{\top}$. Then, different syndrome-based unique decoders can be used to decode $\mathbf{v}$ and to find a codeword $\mathbf{c} \in \Gamma(\mathcal{L}, g)$ such that $\mathbf{v} = \mathbf{c} + \mathbf{e}$. Classic McEliece uses the Berlekamp-Massey Algorithm [Ber68; Mas69b] in Algorithm 7 - Line 2 that calculates the ELP $\sigma_e(x)$ (see Section 2.1.4 with $p = 2$), whose degree is the number of positions in $\mathbf{v}$ that need to be corrected in order to receive a valid codeword $\mathbf{c}$. The error $\mathbf{e}$ can then be reconstructed directly from the zeros of $\sigma_e(x)$, since $\mathbf{e}_i = 1$ if and only if $\sigma_e(\alpha_i) = 0$ for all $i \in \{1, \ldots, n\}$. The roots of the ELP correspond to the specific positions that need to be corrected. The reconstructed error vector $\mathbf{e}'$ is returned if its Hamming weight is t and the re-encoding $\mathbf{e}'\mathbf{H}_{\mathrm{sys}}{}^{\top}$ matches with the input vector $\mathbf{c_0}$, otherwise Failure is given back to the decapsulation algorithm.

CAT 1	"Any attack that breaks the relevant security definition must require computational resources comparable to or greater than those required for key search on a block cipher with a 128-bit key (e.g. AES 128)"
CAT 3	"Any attack that breaks the relevant security definition must require computational resources comparable to or greater than those required for key search on a block cipher with a 192-bit key (e.g. AES 192)"
CAT 5	"Any attack that breaks the relevant security definition must require computational resources comparable to or greater than those required for key search on a block cipher with a 256-bit key (e.g. AES 256)"

Table 3.1: NIST defined categories, extracted from [Nat16].

AES 128	2^{170}/MAXDEPTH quantum gates or 2^{143} classical gates
AES 192	2^{233}/MAXDEPTH quantum gates or 2^{207} classical gates
AES 256	2^{298}/MAXDEPTH quantum gates or 2^{272} classical gates

Table 3.2: NIST defined security levels, extracted from [Nat16].

3.4 Security Levels and Categories

A cryptosystem submitted to NIST must be designed such that the security of the system can be categorized in one or more of the NIST-defined categories. The categories CAT 1, CAT 3 and CAT 5 are listed in Table 3.1. NIST writes that the computational resources for an attack must be comparable or greater than the stated threshold with respect to all metrics that NIST thinks that are potentially relevant for practical security [Nat16, p. 17]. NIST also writes that they consider a variety of possible metrics that reflect different predictions about the future development of quantum and classical computing technology and that those metrics can change over time during the standardization competition. As a preliminary, NIST gave the security levels in Table 3.2 as guidance to the submitters. In Table 3.2 the variable MAXDEPTH takes values from 2^{46} to 2^{96} (for details see [Nat16, p. 17]).

In Classic McEliece the parameters n, m, t need to be chosen such that they meet the security categories defined by NIST. The submitters chose CAT 1, CAT 3 and CAT 5 for interest. To choose good parameters for the specified security categories, the attacks in the following list need to be considered. As a prerequisite, the cryptosystem needs to be designed to be based on a mathematically hard to solve problem. In Classic McEliece this is the SDP for decoding random linear codes (see also Section 2.3). It needs to be investigated that for chosen parameters it is hard for an attacker to obtain

1. the secret key (Key Attack - in Section 3.5.1) and
2. the secret message (Message Attack - in Section 3.5.2).

The submitters of Classic McEliece chose the parameters listed in Table 3.3. In 2022, Zweydinger et al. [EMZ22] state that the code parameters specified for CAT 3 with $n = 4608$ and the parameters specified for CAT 5 with $n = 6688$ and $n = 6960$

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

Table 3.3: Parameter Sets of Classic McEliece KEM [BCL+20]

do not meet the necessary security levels. Thus, parameters are under discussion in the community [PQC20; PQC21].

Not only security levels are important, but also the construction of the KEM as cryptographic protocol. A KEM needs to meet the requirements for Indistinguishability under Chosen Ciphertext Attacks (IND-CCA2) and Indistinguishability under Chosen Plaintext Attacks (IND-CPA) [Nat16, p. 14-15]. IND-CPA means that an attacker who can generate ciphertexts from arbitrary plaintexts, which is always the case in a public-key cryptosystem, cannot distinguish between the input plaintexts by looking at the ciphertexts. Otherwise, the attacker could learn something about the secret key. IND-CCA2 means that an attacker with access to a decryption function can decrypt arbitrary ciphertexts but cannot distinguish from the different outputs which ciphertext has been decrypted and thus cannot learn anything about the secret key. It is ensured that no secret data is leaked if faulty ciphertexts are processed.
These can be formalized considering the following games [KL21, p.405, p.414]. Let's assume an adversary $\mathcal{A}$, which has in the case of IND-CPA access to an encryption algorithm Enc (e.g. Algorithm 5) and in the case of IND-CCA2 also access to a decryption algorithm Dec (e.g. Algorithm 6). At the beginning of the games, a probabilistic polynomial-time key generation algorithm Gen (e.g. Algorithm 3) with security parameter 1^λ chooses according to some distribution a public and private key pair (pk, sk). It is assumed that these keys have a length of at least λ. The public key pk is given to the adversary $\mathcal{A}$. The adversary $\mathcal{A}$ generates two plaintext messages m_0 and m_1 from the message space $\mathcal{M}_{pk}$ defined by the public key pk. The challenger chooses uniformly at random from $\{0, 1\}$ and encrypts the message m_b using the encryption algorithm The challenge ciphertext is given to $\mathcal{A}$. The adversary $\mathcal{A}$ needs to decide, whether the message m_0 or the message m_1 corresponds to the challenge ciphertext. For IND-CPA, the adversary $\mathcal{A}$ has access to the encryption algorithm as long as she needs (adaptively). For IND-CCA2, the adversary $\mathcal{A}$ has access to the encryption and the decryption algorithm as long as she needs (adaptively). She stores her decision to b'. If $b' = b$, then $\mathcal{A}$ succeeds and the experiment outputs 1 otherwise 0. The public-key cryptosystem has indistinguishable encryptions under a chosen-plaintext attack if for all probabilistic polynomial-time adversaries $\mathcal{A}$ there is a negligible function $\mathsf{negl}(.)$ such that

$$\Pr[\text{ IND-CPA}^{\mathcal{A}}_{\mathsf{Enc}}(\lambda) = 1] \leqslant \frac{1}{2} + \mathsf{negl}(\lambda). \tag{3.3}$$

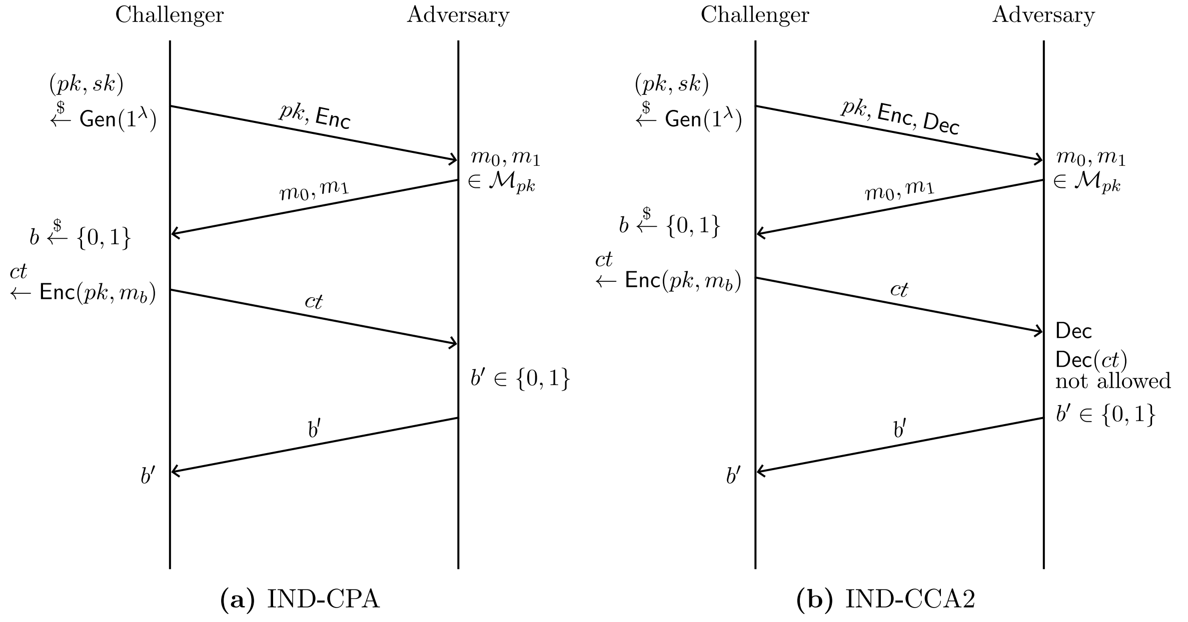


Figure 3.3: IND-CPA and IND-CCA2 games

The public-key cryptosystem has indistinguishable encryptions under a chosen-ciphertext attack if for all probabilistic polynomial-time adversaries $\mathcal{A}$ there is a negligible function $\mathsf{negl}(.)$ such that

$$\Pr[\ \text{IND-CCA}^{\mathcal{A}}_{\mathsf{Dec}}(\lambda) = 1] \leqslant \frac{1}{2} + \mathsf{negl}(\lambda). \tag{3.4}$$

A summary is shown in Algorithm 8 and Algorithm 9 and depicted in Section 3.4.

These games can also be transformed to KEMs that operate on encapsulation and decapsulation algorithms. The IND-CCA2 security can be achieved by implementing the Fujisaki-Okamoto transform [FO13] that most of the cryptosystems submitted to NIST integrate. Classic McEliece takes another approach and achieves IND-CCA2

Algorithm 8 IND-CPA$^{\mathcal{A}}_{\mathsf{Enc}}(\lambda)$

1: $(pk, sk) \xleftarrow{\$} \mathsf{Gen}(1^\lambda)$
2: Public key pk is given to the adversary.
3: Access to Enc is given to the adversary.
4: $\mathcal{A}$ chooses two messages $m_0, m_1 \in \mathcal{M}_{pk}$ of the same length and sends it to the challenger.
5: Challenger chooses $b \xleftarrow{\$} \{0,1\}$ and sends a ciphertext $ct \leftarrow \mathsf{Enc}(pk, m_b)$ to $\mathcal{A}$.
6: $\mathcal{A}$ has still access to Enc
7: $\mathcal{A}$ outputs a bit b'.
8: **Return** 1 if $b = b'$, 0 else.

Algorithm 9 IND-CCA2$^{\mathcal{A}}_{\mathsf{Dec}}(\lambda)$

1: $(pk, sk) \xleftarrow{\$} \mathsf{Gen}(1^\lambda)$
2: Public key pk is given to the adversary and thus access also to Enc.
3: Access to Dec is given to the adversary.
4: Adversary $\mathcal{A}$ chooses two messages $m_0, m_1 \in \mathcal{M}_{pk}$ of the same length and sends it to the challenger.
5: Challenger chooses $b \xleftarrow{\$} \{0, 1\}$ and sends a ciphertext $ct \leftarrow \mathsf{Enc}(pk, m_b)$ to $\mathcal{A}$.
6: Adversary $\mathcal{A}$ has still access to Dec, but it is not allowed to request a decryption of ct itself.
7: Adversary $\mathcal{A}$ outputs a bit b'.
8: **Return** 1 if $b = b'$, 0 else.

security in its own way with plaintext confirmation in Line 3 and Line 4 of Algorithm 5 and implicit rejection in Algorithm 6). In the submission for Round 4 [BCL+22] the plaintext confirmation in Line 3 of Algorithm 5 has been removed due to a patent inquiry.

NIST also states that submissions whose implementation is resistant to side-channel attacks, multi-key attacks and misuse are more desirable than those which are not. The designers have the problem to find the best trade-off between security, speed and implementation aspects of a cryptosystem.

3.5 Best Known Attacks

The two different attack scenarios mentioned above are studied. The first attack scenario is to recover the private key from the public key (key attack). If successful, it allows the attacker to reveal all secret information and impersonate the attacked party. The second attack scenario is to recover the secret message from the ciphertext using the public key, without learning about the private key (message attack). In a message attack, a single secret message is revealed. To reveal multiple messages, the message attack needs to be repeated. The most efficient attacks known for message attacks are based on ISD.

3.5.1 Find secret key from public key - Key Attack

Attacks that find the secret key by knowing only the public key and the public parameters n, m, t are also known as structural attacks. Multiple aspects need to be taken into account when designing the cryptosystem to prevent key attacks.

First, it needs to be ensured that the parameters n and t are large enough such that it is hard to find the secret key from the public key $\mathbf{T}$ by trying all possible constructions of codes with n and t. This is the case for the proposed parameters. There exists $\mathcal{I}_{2^m}(t)$ different possibilities for choosing an irreducible Goppa polynomial of degree t over $\mathbb{F}_{2^m}$ (see Eq. (4.7)) and there exist $\binom{2^m}{n} n!$ different possibilities of

Code class	Proposal	Attack
GRS codes	1986: Niederreiter [Nie86]	1992: Sidelnikov, Shestakov [SS92]
Gabidulin codes	1993: Gabidulin [Gab95]	1996: Gibson [Gib96]
		2008: Overbeck [Ove08]
Reed-Muller codes	1994: Sidelnikov [Sid94]	2007: Minder, Shokollahi [MS07]
		2013: Chizhov, Borodin [CB13]
Concatenated codes	1995: Sendrier [Sen95]	1998: Sendrier [Sen98]
		2015: Puchinger et al. [PMIB17]
Algebraic geometry codes	1996: Janwa, Moreno [JM96]	2008: Faure, Minder [FM08]
		2014: Couvreur et al. [CMP17]
Subcodes for GRS codes	2005: Berger, Loidreau [BL05]	2009: Wieschebrink [Wie10]
QC-LDPC	2007: Baldi et al. [BCGM07]	2010: Otmani et al. [OTD10]
MDPC	2013: Misoczki [MB09]	

Table 3.4: Some proposals and corresponding attacks that break them for McEliece/Niedrreiter cryptosystem using other code classes.

choosing a set of code locators of size n. For the CAT 1 parameters with $n = 3488, m = 12, t = 64$ this is $\mathcal{I}_{2^{12}}(64) = 2^{762} \approx 10^{229}$ and $3488! \approx 2^{36022} \approx 10^{10843}$. Together, this makes 10^{11073} different possibilities, which is impossible to check in a brute-force attack.

Second, the public key needs to be constructed such that it cannot be told whether the parity-check matrix came from of a random code or from a particular family. The parity-check matrix $\mathbf{H}$ is constructed using the Goppa polynomial and the code locators which constitute the secret key. Therefore, the secret information is contained in the parity-check matrix and must be destroyed for publishing the public key. The public key then looks as if it came from a random code. In the original McEliece cryptosystem the destruction is done by left multiplying a permutation matrix and right multiplying a scrambling matrix to the parity-check matrix $\mathbf{H}$. In Classic McEliece this essential security relevant step is done via the Gaussian elimination in Line 5 of Algorithm 3 resulting in $\mathbf{H}_{\text{sys}}$ [BCL+20].

There exist many proposals to replace Goppa codes with other families of codes, mostly in order to reduce the public key size. However, many of them are shown to be insecure or inefficient. Table 3.4 represents some of the proposals and corresponding breaks. These attacks use algorithms that run in polynomial time and allow fast recovery of the secret key from the public key due to special structure of the public key in these cryptosystems.

The original McEliece cryptosystem based on Goppa codes, which are subfield subcodes, still withstands cryptanalysis. It is known to be resistant against structural attacks in polynomial time, as Goppa codes share many characteristics with random codes. However, Goppa codes with a high rate close to one (see Eq. (2.4)) can be distinguished from random codes [FGO+11]. Thus, the degree t of the Goppa polynomial must be large enough for the distinguisher to stop working [FGO+11, Tab. 1], which is the case for all parameters in Table 3.3. In 2017, Couvreur et al. [COT17] found a method to attack Goppa codes that are constructed on field sizes $\mathbb{F}_q^2$ with $m = 2$.

Third, it is needed to make sure that no part of the Goppa code of the secret key is leaked. If the support or the Goppa polynomial is revealed then the Goppa polynomial can be easily found and vice-versa.

Support Set is Known: In the case the support set $\mathcal{L}$ is known to the attacker, a corresponding Goppa polynomial can be computed in polynomial time using codewords calculated from the public key $\mathbf{T}$ [BBD09, p. 125]. Some codewords $\mathbf{c}$ are generated from the public key using $\mathbf{G} = [-\mathbf{T}^\top \mid \mathbf{I}_{\text{n-k}}]$ and $\mathbf{c} = \mathbf{uG}$ for any non-zero word $\mathbf{u}$. Then, the binary irreducible Goppa polynomial $g(x)$ can be reconstructed using the fact that $g(x)$ divides

$$\sum_{i \in \text{supp}(c)} \prod_{\substack{j \in \text{supp}(c) \\ j \neq i}} (x - \alpha_j) \tag{3.5}$$

with α_j the code locators of the Goppa code. A concrete algorithm is given in Chapter 6, Proposition 6.17.

Goppa Polynomial is Known: In the case the Goppa polynomial $g(x)$ is known to the attacker, the corresponding code locators $\alpha_1, \ldots, \alpha_n$ can be computed in polynomial time using the public key $\mathbf{T}$. From the Goppa polynomial $g(x)$ a parity-check matrix $\mathbf{H}$ using an arbitrary set $\mathcal{L}_0$ is constructed. The secret support $\mathcal{L}$ is then obtained by applying the support splitting algorithm to the parity-check matrix $\mathbf{H}$ and $\mathbf{H}_{\text{pub}}$ [BBD09, p. 125]. The support splitting algorithm computes the permutation between two permutation-equivalent codes (see [BBD09, Sec. 4.2], [Sen00]). The permutation between $\mathcal{L}_0$ and $\mathcal{L}$ is the same as the permutation between $\mathbf{H}$ and $\mathbf{H}_{\text{pub}}$.

3.5.2 Information Set Decoding - Message Attack

A message attack tries to reconstruct the secret message from the ciphertext using the known public key. There exist computationally intensive techniques for solving the cryptographic decoding problem in Definition 2.9. An understanding and measure of the complexity of the best decoding techniques is necessary to select secure cryptosystem parameters. The most relevant decoding technique in this regard is Information Set Decoding (ISD). The first idea for ISD traces back to Prange in 1962 [Pra62]. A more concrete analysis was mentioned in [McE78] and given in [RN87; AM87]. Since then many further improvements have been made.

The general idea consists of an attacker that repeatedly selects k symbols at random from a received word $\mathbf{r}$ of length n in hope that none of the k chosen symbols are in error. The k chosen symbols are denoted by $\mathbf{r}_k \in \mathbb{F}_2^k$ and let $\mathcal{R} = \{i_1, \ldots, i_k\}$ be the set of index positions $i_1, \ldots, i_k \in \{1, \ldots, n\}$ that were randomly selected. The received word includes at most w_e errors. Then, the attacker takes the public key $\mathbf{G}' \in \mathbb{F}_2^{k \times n}$ and selects the same k columns indexed by $\mathcal{R}$ to receive a $k \times k$ matrix $\mathbf{G}'_k$. She computes the inverse of $\mathbf{G}'_k$ and the secret message candidate $\tilde{\mathbf{u}} = \mathbf{r}_k \cdot {\mathbf{G}'_k}^{-1}$. If the Hamming weight $\text{wt}(\mathbf{r} - \tilde{\mathbf{u}}\mathbf{G}') \leqslant w_e$ then $\tilde{\mathbf{u}}$ is our searched secret message. The vector $\mathbf{r}_k$ is then our error-free information set. For separable Goppa codes with $d \geqslant 2t + 1$ at most t errors can be corrected and thus $w_e = t$. A summary of the described algorithm is given in Algorithm 10.

Algorithm 10 Prange ISD [Pra62; AM87]

Input: received word $\mathbf{r}$, public key $\mathbf{G}'$
Output: secret message $\tilde{\mathbf{u}}$
1: **repeat**
2: Choose randomly k unique indices $i_a \in \{1, \ldots, n\}$ with $i_a < i_{a+1}$ for $a = 1, \ldots, k$
3: Select the symbols r_{i_a} of $\mathbf{r}$ and construct $\mathbf{r}_k = (r_{i_1}, \ldots, r_{i_k})$
4: Select the columns $\mathbf{g}'^{\top}_{i_a}$ of $\mathbf{G}'$ to construct the $k \times k$ matrix $\mathbf{G}'_k = [\mathbf{g}'^{\top}_{i_1} \cdots \mathbf{g}'^{\top}_{i_k}]$
5: Compute the inverse ${\mathbf{G}'_k}^{-1}$
6: Calculate the secret message candidate $\tilde{\mathbf{u}} = \mathbf{r}_k \cdot {\mathbf{G}'_k}^{-1}$
7: **until** $\mathrm{wt}(\mathbf{r} - \tilde{\mathbf{u}}\mathbf{G}') \leq w_e$

The number of operations needed for the matrix inversion is $\mathcal{O}(k^a)$, where a is between 2 and 3. Assuming there are exactly w_e errors in the received word $\mathbf{r}$, then the number of different possibilities of choosing k out of n is $\binom{n}{k}$ and the number of possibilities to choose error-free symbols is $\binom{n-w_e}{k}$. Therefore, the probability of no error in k randomly selected symbols of n symbols with w_e errors is $\frac{\binom{n-w_e}{k}}{\binom{n}{k}}$. The total expected work factor is then [AM87]

$$\mathrm{WF}_{\mathrm{Prange}} = k^a \frac{\binom{n}{k}}{\binom{n-w_e}{k}} , \tag{3.6}$$

with n the code length, k the dimension, w_e the number of errors and $2 < a \leqslant 3$. For simple schoolbook matrix inversion e.g. Gauss-Jordan elimination, the factor is $a = 3$. For more elaborated matrix inversion algorithms a factor of $a < 3$ can be achieved [Str69].

The above described ISD algorithm directly applies to the original McEliece cryptosystem, whose public key is a scrambled generator matrix. But, this attack can also be transformed to the Niederreiter cryptosystem that has a parity-check matrix as public key. For the Niederreiter cryptosystem the syndrome $\mathbf{s} = \mathbf{c_0}$ is a public ciphertext. From the parity-check matrix $\mathbf{K}$ there are w_e columns randomly selected to form an $n - k \times n - k$ matrix. These columns are summed up and checked if they give the syndrome $\mathbf{s}$.

The problem of generic decoding can be transformed to the problem of finding a codeword of low weight in a slightly larger code. Algorithms addressing that problem have been presented by Canteaut, Chabanne in 1994 [CC94], Canteaut, Chabaud in 1998 [CC98], Canteaut, Sendrier in 1998 [CS98], Engelbert et al. in 2007 [EOS07] and Bernstein in 2008 [BLP08].

Based on the naive Prange ISD, many more elaborate ISD algorithms exist and ISD algorithms have still today a broad attention for research. Lee, Brickell and van Tilbourg [LB88; Til88] improved the work factor of Eq. (3.6) in 1988. The current best known ISD algorithms are all variants and build upon Stern's algorithm [Ste89]

and Dumer [Dum91] in 1989/91, that combine the algorithm of Lee, Brickell with birthday decoding.

The birthday decoding principle applied to the Niederreiter cryptosystem splits the public key $\mathbf{K}$ into two equally sized parts $\mathbf{K}_1$ and $\mathbf{K}_2$ such that $\mathbf{K} = [\mathbf{K}_1\mathbf{K}_2]$ and enumerates through the two following sets

$$\mathcal{A}_1 = \left\{\mathbf{e}_1\mathbf{K}_1^\top \mid \mathrm{wt}_\mathrm{H}(\mathbf{e}_1) = \frac{w_e}{2}\right\} \tag{3.7}$$

$$\mathcal{A}_2 = \left\{\mathbf{e}_2\mathbf{K}_2^\top + \mathbf{s} \mid \mathrm{wt}_\mathrm{H}(\mathbf{e}_2) = \frac{w_e}{2}\right\} \tag{3.8}$$

where $\mathbf{e}_1, \mathbf{e}_2 \in \mathbb{F}_2^{n/2}$ are the splits of $\mathbf{e} = (\mathbf{e}_1\mathbf{e}_2)$, which needs to be found. The goal is to find w_e columns of $\mathbf{K}$ that add to the syndrome $\mathbf{s}$. The vectors $\mathbf{e}_1$ and $\mathbf{e}_2$ for which the intersection of the two sets is not the empty set $\mathcal{A}_1 \cap \mathcal{A}_2 \neq \emptyset$, are solutions of the syndrome decoding problem with $\mathbf{s} + \mathbf{e}_1\mathbf{K}_1^\top + \mathbf{e}_2\mathbf{K}_2^\top = \mathbf{0} \mod 2$. To obtain most of the solutions, the principle needs to be applied to different splits of the public key $\mathbf{K}$ e.g. by randomly picking the columns for the two halves, or by allowing the two halves to overlap such that some columns of $\mathbf{K}$ are part of both $\mathbf{K}_1$ and $\mathbf{K}_2$.

For the Stern/Dumer algorithm the public key $\mathbf{K}$ need to be transformed such that $\mathbf{K}' \in \mathbb{F}_2^{(n-k)\times n}$ in the following form

$$\mathbf{K}' = \begin{bmatrix} \mathbf{I}_{n-k-l} & \mathbf{K}'_\mathrm{top} \\ \mathbf{0} & \mathbf{K}'_\mathrm{bottom} \end{bmatrix} \tag{3.9}$$

is achieved, with $\mathbf{K}'_\mathrm{top} \in \mathbb{F}_2^{(n-k-l)\times(k+l)}$ and $\mathbf{K}'_\mathrm{bottom} \in \mathbb{F}_2^{l\times(k+l)}$. For this, the public key matrix is multiplied by an invertible matrix $\mathbf{U}$ and a permutation matrix $\mathbf{P}$. Then, $\mathbf{K}' = \mathbf{UKP}$ and the syndrome transforms to $\mathbf{s}' = \mathbf{s}\mathbf{U}^\top$. The searched error vector $\mathbf{e}$ is also transformed to $\mathbf{e}' = \mathbf{eP}$ and to receive the plaintext by knowing $\mathbf{e}'$, a back transformation to $\mathbf{e} = \mathbf{e}'\mathbf{P}^{\top^{-1}}$ is needed. Corresponding to the above form in Eq. (3.9) the syndrome is separated into $\mathbf{s}'^\top = (\mathbf{s}'_\mathrm{top}\mathbf{s}'_\mathrm{bottom})$ with $\mathbf{s}'_\mathrm{top} \in \mathbb{F}_2^{n-k-l}$ and $\mathbf{s}'_\mathrm{bottom} \in \mathbb{F}_2^{l}$. The first step of the algorithm is then to apply the birthday decoding principle to $\mathbf{s}'_\mathrm{bottom} = \mathbf{e}'_\mathrm{bottom}\mathbf{K}'^{\top}_\mathrm{bottom}$ to find $\mathbf{e}'_\mathrm{bottom}$ with $\mathrm{wt}_\mathrm{H}(\mathbf{e}'_\mathrm{bottom}) = a$ and $0 < a < w_e$. The second step applies the principle of Lee, Brickell and enumerates through all solutions of $\mathbf{e}'_\mathrm{bottom}$ and checks if the Hamming weight of $\mathbf{e}'_\mathrm{top} = \mathbf{e}'_\mathrm{bottom}\mathbf{K}'^{\top}_\mathrm{top} + \mathbf{s}'_\mathrm{top}$ corresponds to $w_e - a$. If $\mathrm{wt}_\mathrm{H}(\mathbf{e}'_\mathrm{top}) = w_e - a$, then $\mathbf{e} = (\mathbf{e}'_\mathrm{top}\mathbf{e}'_\mathrm{bottom})\mathbf{P}^{\top^{-1}}$ In order to minimize the complexity of the algorithm a good value of the parameters l and a must be found.

Bernstein et al. in 2011 [BLP11] use ball collision decoding. The MMT algorithm introduced in 2011 by [MMT11] improves the algorithm by Dumer using improved birthday decoding [HJ10], while BJMM in 2012 by [BJMM12] use further improved birthday decoding. May-Ozerov [MO15] and Both May [BM18] again reduce the work factor.

Esser, Bellini in 2021 [EB22] established a framework that analyses the complexity of all these algorithms in a unified and practical model. They compare the different variants and give concrete hardness estimations. The improvements compared to the algorithm of Prange are mostly based on taking less computation operations at

the cost of more memory use. Memory use also takes time and may slow down the algorithm, which needs to be added to the work factors above. Thus, a trade-off between operations and memory consumption needs to be taken into account. Memory usage can be modeled. Prominent models take a logarithmic or cube-root penalty factor, that is costs of $T \cdot \log M$ or $T \cdot \sqrt[3]{M}$, respectively, with time complexity T and memory usage M [EB22]. Prange is a memory-free algorithm.

For cyclic codes there exists a speed-up for ISD algorithms in a factor of roughly $\sqrt{k}$ with k the dimension of the code [Sen11]. This speed-up is called Decoding One Out of Many (DOOM), but that does not apply for Classic McEliece.

In 2010, Bernstein [Ber10] investigated ISD for quantum computers and found out that they bring roughly a square-root speed-up.

Chapter 4

Investigating Generalized Goppa Codes

In this chapter Generalized Goppa Codes (GGC) are introduced. These codes are defined by a Goppa polynomial $g(x)$ and a set $\mathcal{L}_\mathsf{r}$ of fractions of polynomials in $\mathbb{F}_q[x]$. In the special case where all numerators of $\mathcal{L}_\mathsf{r}$ are 1 and all denominators of $\mathcal{L}_\mathsf{r}$ are monic polynomials of degree 1, this corresponds to classical Goppa codes introduced in Section 2.1.3. Therefore, GGC are an extension of classical Goppa codes to a new class of codes. Generalized Goppa codes are also called generalized $(\mathcal{L}, g)$-codes.

In Section 4.1 generalized Goppa codes and binary generalized Goppa codes are investigated. A parity-check matrix for binary GGC with a set $\mathcal{L}_\mathsf{p}$ of polynomials of any degree is developed. It is shown that a careful selection of the polynomials in $\mathcal{L}_\mathsf{p}$ leads to a lower bound on the minimum Hamming distance of generalized Goppa codes which improves upon previously known bounds. A syndrome-based unique decoder for generalized Goppa codes which can decode errors up to half of the minimum distance is presented.

Section 4.2 investigates the application of GGCs in a Niederreiter cryptosystem. For this, binary GGCs are used. They have the advantage that the code length n can be chosen bigger than the 2^m allowed by classical Goppa codes. This allows to reduce the field size, albeit at the cost of a reduced security level or a bigger public key size.

The content of Section 4.1.1 and Section 4.1.2 is a joint work with Hedongliang Liu and has been published in [LPZW21, Sec. 3-4] [1]. The code parameters in Section 4.2 have been published in [LPZW21, Sec. 6].

4.1 Generalized Goppa Codes

Generalized Goppa Codes are introduced by Shekhunova, Mironchikov and Bezzateev in [SM81; BS97] and defined as follows.

Definition 4.1 (q-ary Generalized Goppa Codes [BS97; BS11]).
Let be given a Goppa polynomial $g(x) \in \mathbb{F}_q[x]$ of degree t and a support $\mathcal{L}_\mathsf{r}$ of fractions

[1] Interleaved Generalized Goppa Codes in Section 5 of [LPZW21] are not part of this thesis.

with polynomials $\varphi_i(x), \psi_i(x) \in \mathbb{F}_q[x]$ *and* $\deg(\varphi_i(x)) < \deg(\psi_j(x))$ *such that*

$$\mathcal{L}_{\mathsf{r}} = \left(\frac{\varphi_1(x)}{\psi_1(x)}, \frac{\varphi_2(x)}{\psi_2(x)}, \ldots, \frac{\varphi_n(x)}{\psi_n(x)} \right) \tag{4.1}$$

with $\gcd(\varphi_i(x), \psi_j(x)) = 1$ *for* $i \neq j$ *and* $\forall i, j \in \{1, \ldots, n\}$*. Then, the q-ary GGC is defined by*

$$\Gamma(\mathcal{L}_{\mathsf{r}}, g) := \left\{ \mathbf{c} \in \mathbb{F}_q^n \mid \sum_{i=1}^{n} c_i \frac{\varphi_i(x)}{\psi_i(x)} = 0 \mod g(x) \right\}, \tag{4.2}$$

where $\gcd(\varphi_i(x), g(x)) = 1$ *and* $\gcd(\psi_i(x), g(x)) = 1$ *for* $i \in \{1, \ldots, n\}$*.*

All $\varphi_i(x)$ and $\psi_i(x)$ are relatively prime to each other and to the Goppa polynomial $g(x)$.

Theorem 4.2 (q-ary GGC Minimum Distance).
The minimum distance of a generalized Goppa code satisfies ([BS11] without proof)

$$d \geqslant \frac{t+1}{\ell} \tag{4.3}$$

and dimension $k \geqslant n - t$ *for* $\ell = \max(\deg(\psi_i(x)))$ *with* $i \in \{1, \ldots, n\}$*.*

4.1.1 Binary Generalized Goppa Codes

For binary GGC we have $q = 2^m$ with $t = \deg(g(x))$. From definitions in Eq. (4.1) and Eq. (4.2) a subclass of binary GGC [BS13, Chap. 3], denoted by $\Gamma(\mathcal{L}_{\mathsf{p}}, g)$, can be defined.

Definition 4.3 (Binary Generalized Goppa Codes).
Let be given a Goppa polynomial $g(x) \in \mathbb{F}_q[x]$ *of degree* t *with* $q = 2^m$ *and a support* $\mathcal{L}_{\mathsf{p}}$ *of irreducible polynomials* $f_i(x) \in \mathbb{F}_q[x]$ *such that*

$$\mathcal{L}_{\mathsf{p}} = \big(f_1(x), f_2(x), \ldots, f_n(x) \big) \tag{4.4}$$

with $\deg(f_i(x)) \leqslant \deg(g(x))$ *and* $\gcd(f_i(x), f_j(x)) = 1, \forall i \neq j$*, and* $\gcd(f_i(x), g(x)) = 1$*,* $\forall i, j \in \{1, \ldots, n\}$*. Then,*

$$\Gamma(\mathcal{L}_{\mathsf{p}}, g) := \left\{ \mathbf{c} \in \mathbb{F}_2^n \mid \sum_{i=1}^{n} c_i \frac{f_i'(x)}{f_i(x)} = 0 \mod g(x) \right\}, \tag{4.5}$$

is a generalized Goppa code, where $f_i'(x)$ *is the formal derivative of* $f_i(x)$*.*

The degree of $f_i(x)$ is denoted by l_i and ℓ is the maximum degree of all $f(x)$ with

$$\ell = \max\{l_i\} = \max_{f(x) \in \mathcal{L}_{\mathsf{p}}} \{\deg(f(x))\}. \tag{4.6}$$

Properties of Binary Generalized Goppa Codes:

Length:	$n \geqslant m \cdot \deg(g(x))$	(4.20)
	$n \leqslant \sum_{i=1}^{\ell} \mathcal{I}_q(i)$	(4.8)
Dimension:	$k = n - \operatorname{rank}(\mathbf{H}^{\text{bin}}) \geqslant n - m \cdot \deg(g(x))$	(4.19)
Minimum distance:	$d \geqslant d_{\text{g}} := \frac{\deg(g(x)) + 1}{\ell}$	(4.21)
	$d_{\text{sep}} := \frac{2 \cdot \deg(g(x)) + 1}{\ell}$	(4.29)
Error-Correction Capability:	$w_e \leqslant w_{e_{\text{sep}}} := \left\lfloor \frac{\deg(g(x))}{\ell} \right\rfloor = \left\lfloor \frac{d_{\text{sep}}}{2} \right\rfloor$	(4.39)

For $\ell = 1$ we have a classical binary Goppa code that corresponds to Eq. (2.15) in Section 2.1.3.

The code length n is bounded by the number of irreducible polynomials in $\mathbb{F}_{2^m}[x]$ existing with degree smaller or equal to $\deg(g(x)) = t$. For classical Goppa codes the code length is limited by the field size $q = 2^m$. The number $\mathcal{I}_q(\tau)$ of irreducible polynomials of a degree τ over $\mathbb{F}_q$ can be calculated ([Rot06, p. 225]) by

$$\mathcal{I}_q(\tau) = \frac{1}{\tau} \sum_{k|\tau} \mu(k) \cdot q^{\frac{\tau}{k}} \tag{4.7}$$

where $\mu(k)$ is the Möbius function (cf. [Rot06, p. 224])

$$\mu(k) = \begin{cases} 1 & \text{if } k = 1 \\ (-1)^s & \text{if } k \text{ is a product of } s \text{ distinct primes} \\ 0 & \text{otherwise.} \end{cases}$$

Thus, the code length n is upper bounded with the following theorem.

Theorem 4.4 (Code Length [NB20a]).
Let $q = 2^m$ for some positive integer m and denote $\ell = \max_{f(x) \in \mathcal{L}_\mathsf{p}} \deg f(x)$. The length of $\Gamma(\mathcal{L}_\mathsf{p}, g)$ is limited by

$$n \leqslant \sum_{\tau=1}^{\ell} \mathcal{I}_q(\tau), \tag{4.8}$$

where $\mathcal{I}_q(\tau)$ is the number of irreducible polynomials of degree τ in the polynomial ring $\mathbb{F}_q[x]$.

Then, the maximum possible code length n for given degree t of the Goppa polynomial

$g(x)$ is

$$\sum_{\tau=1}^{t} \mathcal{I}_q(\tau). \tag{4.9}$$

By definition in Eq. (4.4), the code locators in $\mathcal{L}_\mathsf{p}$ are $\mathbb{F}_q$-irreducible polynomials, which can be represented by their roots in the splitting field such that (see also Theorem 2.5 in Chapter 2)

$$f_i(x) = \prod_{j=0}^{l_i-1} \left(x - \gamma_i^{q^j}\right), \ \forall i = 1, \dots, n \tag{4.10}$$

of degree l_i, where $\gamma_i^{q^j} \in \mathbb{F}_{q^{l_i}}$ are the roots of $f_i(x)$ in the splitting field $\mathbb{F}_{q^{l_i}}$.

A parity-check matrix for binary GGC with code locators of first and second degree is specified without a proof in [NB20b]. In the following theorem a parity-check matrix for generalized Goppa codes with code locators of arbitrary degree is constructed.

Theorem 4.5 (Parity-Check Matrix).
Given a binary generalized Goppa code $\Gamma(\mathcal{L}_\mathsf{p}, g)$ as in Eq. (4.5) with polynomials of Eq. (4.10) in $\mathcal{L}_\mathsf{p}$. Let $t = \deg(g(x))$ be the Goppa polynomial degree and $n = |\mathcal{L}_\mathsf{p}|$ the code length. A parity-check matrix $\mathbf{H}$ of $\Gamma(\mathcal{L}_\mathsf{p}, g)$ is

$$\mathbf{H} = [\mathbf{h}_1^\top \ \mathbf{h}_2^\top \ \cdots \ \mathbf{h}_n^\top] \in \mathbb{F}_q^{t \times n} \tag{4.11}$$

with $\mathbf{h}_i = (h_{i,1} \ h_{i,2} \ \cdots \ h_{i,t})$, where

$$h_{i,j} = \sum_{\iota=0}^{l_i-1} \frac{\gamma_i^{(j-1)q^\iota}}{g\left(\gamma_i^{q^\iota}\right)}, \ \forall i = 1, \dots, n \text{ and } j = 1, \dots t$$

such that $\mathbf{H}\mathbf{c}^\top = \mathbf{0}, \ \forall \mathbf{c} \in \Gamma(\mathcal{L}_\mathsf{p}, g)$.

Proof. From Eq. (4.4), $f_i(x)$ and $g(x)$ are relatively prime such that $\gcd(g(x), f_i(x)) = 1$. That implies that the roots of $f_i(x)$ are not roots of $g(x)$ and $g(\gamma_i^j) \neq 0, \forall j = 0, \dots, l_i - 1$ and $i = 1, \dots, n$. Using the relation in Eq. (2.32) an inverse $f_i^{-1}(x) \bmod g(x)$ can be found by the Extended Euclidean Algorithm (EEA)

$$1 \equiv f_i(x) f_i^{-1}(x) \mod g(x) \tag{4.12}$$

with $v(x) = f_i^{-1}(x)$ the output and $a(x) = g(x), b(x) = f_i(x)$ the input of Algorithm 1 in Section 2.2. Let the Goppa polynomial be

$$g(x) = g_0 + g_1 x + \cdots + g_t x^t$$

with $g_t \neq 0$. Then, by using the EEA Algorithm 1 in Section 2.2 for calculating the

inverse we obtain

$$\frac{f_i'(x)}{f_i(x)} \mod g(x) = \left(\prod_{j=0}^{l_i-1} g\left(\gamma_i{}^{q^j}\right)\right)^{-1} \cdot \\ \sum_{\tau=0}^{t-1} x^\tau \left(\sum_{k=\tau+1}^{t} g_k \left(\sum_{j=0}^{l_i-1} \gamma_i^{(k-1-\tau)q^j} \prod_{\substack{\xi=0,\\ \xi\neq j}}^{l_i-1} g\left(\gamma_i{}^{q^\xi}\right)\right)\right). \tag{4.13}$$

The above polynomial can also be written in vector notation where the $(t-\tau)$-th element of the vector $\mathbf{v}_i$ contains the polynomial coefficient of $x^\tau, \forall \tau \in [0, t-1]$. The vector $\mathbf{v}_i^\top$ forms the i-th column of the matrix $\mathbf{C} \cdot \mathbf{H}$ of size $(t \times n)$ where

$$\mathbf{C} = \begin{bmatrix} g_t & 0 & \dots & 0 \\ g_{t-1} & g_t & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ g_1 & g_2 & \dots & g_t \end{bmatrix}. \tag{4.14}$$

Plugging the vectors $\mathbf{v}_i$ for all $i = 1, \dots, n$ into Eq. (4.5), it can be verified that $\mathbf{c} \in \Gamma(\mathcal{L}_\mathsf{p}, g)$ if and only if $[\mathbf{v}_1^\top, \mathbf{v}_2^\top, \dots, \mathbf{v}_n^\top] \cdot \mathbf{c}^\top = \mathbf{CH} \cdot \mathbf{c}^\top = \mathbf{0}$. Therefore,

$$\mathbf{CH} = \widetilde{\mathbf{H}} \tag{4.15}$$

is a parity-check matrix of $\Gamma(\mathcal{L}_\mathsf{p}, g)$. Since $\mathbf{C}$ is invertible, $\mathbf{H}$ is also a parity-check matrix. □

For example, for classical Goppa codes Eq. (4.13) simplifies to

$$\frac{1}{(x-\alpha_i)} \mod g(x) = g(\alpha_i)^{-1} \cdot \left(\sum_{\tau=0}^{t-1} x^\tau \left(\sum_{k=\tau+1}^{t} g_k \alpha_i^{k-1-\tau}\right)\right) \tag{4.16}$$

and for GGC with a code locator of second degree it reads

$$\frac{\gamma_j + \gamma_i^q}{(x-\gamma_i)(x-\gamma_i^q)} \mod g(x) = (g(\gamma_i) g(\gamma_i^q))^{-1} \cdot \\ \cdot \left(\sum_{\tau=0}^{t-1} x^\tau \left(\sum_{k=\tau+1}^{t} g_k \left(\gamma_i^{k-1-\tau} g(\gamma_i^q) + \gamma_i^{q(k-1-\tau)} g(\gamma_i)\right)\right)\right). \tag{4.17}$$

In matrix notation a parity-check matrix of arbitrary degrees l_i can be written as

$$\mathbf{H} = \begin{bmatrix} \frac{1}{g(\gamma_1)} + \cdots + \frac{1}{g(\gamma_1^{(l_1-1)q})} & \cdots & \frac{1}{g(\gamma_n)} + \cdots + \frac{1}{g(\gamma_n^{(l_n-1)q})} \\ \frac{\gamma_1}{g(\gamma_1)} + \cdots + \frac{\gamma_1^{(l_1-1)q}}{g(\gamma_1^{(l_1-1)q})} & \cdots & \frac{\gamma_n}{g(\gamma_n)} + \cdots + \frac{\gamma_n^{(l_n-1)q}}{g(\gamma_n^{(l_n-1)q})} \\ \vdots & \cdots & \vdots \\ \frac{\gamma_1^{t-1}}{g(\gamma_1)} + \cdots + \frac{\gamma_1^{(t-1)(l_1-1)q}}{g(\gamma_1^{(l_1-1)q})} & \cdots & \frac{\gamma_n^{t-1}}{g(\gamma_n)} + \cdots + \frac{\gamma_n^{(t-1)(l_n-1)q}}{g(\gamma_n^{(l_n-1)q})} \end{bmatrix} \tag{4.18}$$

Using the relation in Eq. (2.2) the above parity-check matrix with $l_1 = 1$ and $l_2 = 2$ corresponds to the parity-check matrix presented in [NB20b] with code locators of degree 1 and 2.
A binary parity-check matrix $\mathbf{H}^{\text{bin}} \in \mathbb{F}_2^{tm \times n}$ of $\Gamma(\mathcal{L}_{\mathsf{p}}, g)$ can be obtained by replacing every entry in $\mathbf{H}$ from Eq. (4.11) or Eq. (4.18) with a length-m column vector representation over $\mathbb{F}_2$ according to some fixed basis of $\mathbb{F}_q$ over $\mathbb{F}_2$.

Theorem 4.6 (Dimension).
For a binary generalized Goppa code $\Gamma(\mathcal{L}_{\mathsf{p}}, g)$ as in Eq. (4.5), the dimension is

$$k = n - \text{rank}(\mathbf{H}^{\text{bin}}) \geqslant n - tm \tag{4.19}$$

where $\mathbf{H}^{\text{bin}} \in \mathbb{F}_2^{tm \times n}$ is the $\mathbb{F}_2$-representation of $\mathbf{H} \in \mathbb{F}_{2^m}^{t \times n}$ and $t = \deg(g(x))$.

From the above theorem it follows that

$$n \geqslant mt. \tag{4.20}$$

Theorem 4.7 (Minimum Distance).
The minimum Hamming distance d of $\Gamma(\mathcal{L}_{\mathsf{p}}, g)$ is

$$d \geqslant d_{\text{g}} := \frac{t+1}{\ell}, \tag{4.21}$$

where $t = \deg(g(x))$ and $\ell = \max_{f(x) \in \mathcal{L}_{\mathsf{p}}} \deg(f(x))$.

Proof. For all codewords $\mathbf{c} \in \Gamma(\mathcal{L}_{\mathsf{p}}, g)$ we have $\mathbf{H}\mathbf{c}^\top = \mathbf{0} \iff \mathbf{H}^{\text{bin}}\mathbf{c}^\top = \mathbf{0}$. Consider one codeword $\mathbf{c}$, the product of all $f_i(x)$ for $i \in \operatorname{supp} \mathbf{c}$ is denoted as

$$F_{\mathbf{c}}(x) := \prod_{i \in \text{supp}(\mathbf{c})} f_i(x),$$

where its formal derivative is denoted as

$$F'_{\mathbf{c}}(x) := \sum_{i \in \text{supp}(\mathbf{c})} f'_i(x) \prod_{\substack{j \in \text{supp}(\mathbf{c}) \\ j \neq i}} f_j(x).$$

Furthermore, let

$$R_{\mathbf{c}}(x) := \sum_{i \in \operatorname{supp}(\mathbf{c})} \frac{f_i'(x)}{f_i(x)} = \frac{F_{\mathbf{c}}'(x)}{F_{\mathbf{c}}(x)}, \tag{4.22}$$

where $f_i'(x)$ is the formal derivative of $f_i(x)$. Since all $f_i(x)$ have distinct roots by Eq. (4.10), they are relatively prime to its formal derivative $f_i'(x)$ and therefore also $\gcd(F_{\mathbf{c}}'(x), F_{\mathbf{c}}(x)) = 1$ holds. By definition in Eq. (4.4) it applies $\gcd(f_i(x), g(x)) = 1$ and thus also $\gcd(F_{\mathbf{c}}(x), g(x)) = 1$. From these properties it follows

$$R_{\mathbf{c}}(x) \equiv 0 \mod g(x) \quad \Longleftrightarrow \quad g(x)|F_{\mathbf{c}}'(x).$$

Since we are working over a field of characteristic 2, the formal derivative $F_{\mathbf{c}}'(x)$ has only even powers and is a perfect square. Let $\bar{g}(x)$ be the lowest degree perfect square which is divisible by $g(x)$, then

$$g(x)|F_{\mathbf{c}}'(x) \quad \Longleftrightarrow \quad \bar{g}(x)|F_{\mathbf{c}}'(x)$$

and

$$\begin{aligned} \mathbf{c} \in \Gamma \quad &\Longleftrightarrow \quad R_{\mathbf{c}}(x) = 0 \mod g(x) \\ &\Longleftrightarrow \quad \bar{g}(x)|F_{\mathbf{c}}'(x). \end{aligned} \tag{4.23}$$

With $l_i = \deg(f_i(x))$ the degree of $F_{\mathbf{c}}(x)$ is $\deg(F_{\mathbf{c}}(x)) = \sum_{i \in \operatorname{supp}(\mathbf{c})} l_i$ and

$$\deg(F_{\mathbf{c}}'(x)) \leqslant \deg(F_{\mathbf{c}}(x)) - 1 = \sum_{i \in \operatorname{supp}(\mathbf{c})} l_i - 1. \tag{4.24}$$

Consider a vector $\mathbf{v}_m \in \mathbb{F}_2^n$ whose non-zero index positions $\operatorname{supp} \mathbf{v}_m$ correspond to index positions i for which $f_i(x)$ have highest degree ℓ, then

$$\deg(F_{\mathbf{v}_m}'(x)) \leqslant \operatorname{wt}(\mathbf{v}_m) \cdot \ell - 1. \tag{4.25}$$

Note that $\mathbf{v}_m$ is not necessarily a codeword. Assume that $\deg(F_{\mathbf{v}_m}'(x)) \overset{!}{\geqslant} \deg(\bar{g}(x))$ then $\operatorname{wt}(\mathbf{v}_m) \geqslant \frac{\deg(\bar{g}(x))+1)}{\ell}$. To have Eq. (4.23) fulfilled we require

$$\deg(F_{\mathbf{c}}'(x)) \geqslant \deg(\bar{g}(x)) \quad \forall\ \mathbf{c} \in \Gamma. \tag{4.26}$$

For any $\mathbf{c}$ with $\operatorname{wt}(\mathbf{c}) < \operatorname{wt}(\mathbf{v}_m)$ it follows that $\deg(F_{\mathbf{c}}'(x)) < \deg(F_{\mathbf{v}_m}'(x))$. In other words, we cannot find a codeword $\mathbf{c}$ with $\operatorname{wt}(\mathbf{c}) < \operatorname{wt}(\mathbf{v}_m)$ and $\deg(F_{\mathbf{c}}'(x)) \geqslant \deg(F_{\mathbf{v}_m}'(x))$. Therefore, to fulfill the relation in Eq. (4.26) it follows

$$d(\Gamma) = \min_{\mathbf{c} \in \Gamma} \operatorname{wt}(\mathbf{c}) \geqslant \operatorname{wt}(\mathbf{v}_m) \tag{4.27}$$

$$\geqslant \frac{\deg(\bar{g})(x) + 1}{\ell} \geqslant \frac{\deg(g(x)) + 1}{\ell} = d_{\mathrm{g}}. \tag{4.28}$$

□

Corollary 4.8 (Separable GGC Minimum Distance).
A binary separable generalized Goppa code $\Gamma(\mathcal{L}_{\mathsf{p}}, g)$ *with Goppa polynomial* $g(x)$ *whose roots are all distinct, is the same code as* $\Gamma(\mathcal{L}_{\mathsf{p}}, g^2)$*. Thus, the minimum distance is*

$$d \geqslant d_{\text{sep}} := \frac{2t+1}{\ell}. \tag{4.29}$$

Proof. Since $g(x)$ is separable, all roots of $g(x)$ are distinct. Therefore, the proof of Theorem 4.7 applies with $\bar{g}(x) = g(x)^2$. □

For separable GGC whose code locators have only even degrees, the lower bound on the minimum distance is increased by $1/\ell$ compared to Eq. (4.29).

Corollary 4.9 (Separable Even-Degree Code Locators Minimum Distance).
A binary separable GGC with the support $\mathcal{L}_{\mathsf{p}_{\text{even}}}$ *of even-degree polynomials has at least a minimum distance* d *with*

$$d \geqslant d_{\text{even}} := \frac{2t+2}{\ell}. \tag{4.30}$$

Proof. Since $\deg(f_i(x))$ is even for all $f_i(x) \in \mathcal{L}_{\mathsf{p}}$, so is $\deg F_{\mathbf{c}}(x)$ even. Then, for a binary field it applies that $\deg F'_{\mathbf{c}}(x) \leqslant \sum_{i \in \text{supp}(\mathbf{c})} l_i - 2$ in Eq. (4.24) and $\deg F'_{\mathbf{v}_m}(x) \leq \text{wt}(\mathbf{v}_m) \cdot l - 2$ in Eq. (4.25). Using Corollary 4.8 and the proof in Theorem 4.7, the statement follows. □

4.1.2 Decoding of Binary Generalized Goppa Codes

For decoding binary GGC an explicit decoding algorithm and a new decoding radius are presented. The unique decoding radius for binary GGC is $\left\lfloor \frac{d}{2} \right\rfloor$, which is slightly different compared to the common form of $\left\lfloor \frac{d-1}{2} \right\rfloor$ for other codes. In the following we apply the relations for decoding Goppa codes in Section 2.1.4 of Chapter 2 to binary GGC.

Definition 4.10 (Syndrome Polynomial, Error Locator Polynomial (ELP), Error Evaluator Polynomial (EEP)).
Let $\mathbf{e} \in \mathbb{F}_2^n$ *be an error vector and* $\mathcal{E} = \text{supp}(\mathbf{e})$ *for a binary GGC of code length* n *with* $\mathbf{c} \in \Gamma(\mathcal{L}_{\mathsf{p}}, g)$*. The syndrome polynomial is defined by*

$$s(x) := \sum_{i \in \mathcal{E}} e_i \frac{f'_i(x)}{f_i(x)} \mod g(x). \tag{4.31}$$

The Error Locator Polynomial (ELP) is defined by

$$\sigma(x) := \prod_{i \in \mathcal{E}} f_i(x) \tag{4.32}$$

and the Error Evaluator Polynomial (EEP) is defined by

$$\eta(x) := \sum_{i\in\mathcal{E}} e_i f_i'(x) \prod_{j\in\mathcal{E}\setminus\{i\}} f_j(x). \tag{4.33}$$

Let $\mathbf{c} \in \Gamma(\mathcal{L}_\mathsf{p}, g)$ be a codeword and $\mathbf{r} = \mathbf{c} + \mathbf{e} \in \mathbb{F}_2^n$ the received word from a noisy channel. Then, the syndrome polynomial is calculated from $\mathbf{r} = (r_1, \dots, r_n)$ by

$$s(x) = \sum_{i=1}^{n} r_i \frac{f_i'(x)}{f_i(x)} \mod g(x). \tag{4.34}$$

The syndrome polynomial $s(x) = \sum_{i=1}^{t} s_i x^{t-i}$ can also be written in vector notation as $\mathbf{s} = (s_1, \dots, s_t)$ and corresponds to $\mathbf{s} = \mathbf{r}\widetilde{\mathbf{H}}^\top$ with $\widetilde{\mathbf{H}}$ from Eq. (4.15).

In Algorithm 11 a syndrome-based decoder for $\Gamma(\mathcal{L}_\mathsf{p}, g)$ is presented. The main step of decoding is to determine the ELP $\sigma(x)$ and the EEP $\eta(x)$ given the syndrome $s(x)$ by solving the key equation.

Lemma 4.11 (Key Equation).
For a binary generalized Goppa code $\Gamma(\mathcal{L}_\mathsf{p}, g)$ with error $\mathbf{e}$ of weight w_e a key equation is

$$\eta(x) = \sigma(x)s(x) \mod g(x) \tag{4.35}$$

with

$$\gcd(\sigma(x), \eta(x)) = 1 \tag{4.36}$$

$$\deg \eta(x) < \deg \sigma(x) \leqslant w_e \cdot \ell \tag{4.37}$$

where $\ell = \max_{f(x)\in\mathcal{L}_\mathsf{p}} \deg f(x)$.

Proof. With $\mathcal{E} = \mathrm{supp}(\mathbf{e})$ and $w_e = |\mathcal{E}|$ the Eq. (4.35) follows from Eq. (4.31), Eq. (4.32), and Eq. (4.33) since

$$s(x) = \frac{\sum_{i\in\mathcal{E}} e_i f_i'(x) \prod_{j\in\mathcal{E}\setminus\{i\}} f_j(x)}{\prod_{i\in\mathcal{E}} f_i(x)} = \frac{\eta(x)}{\sigma(x)} \mod g(x). \tag{4.38}$$

Since all $f_i(x)$ have distinct roots by definition in Eq. (4.10) it applies that $\gcd(f_i(x), f_i'(x)) = 1$ and Eq. (4.36) follows. From the definition of an ELP in Eq. (4.32) the degree of $\sigma(x)$ is upper bounded with the maximum degree of $f_i(x)$ such that $\deg \sigma(x) = \sum_{i\in\mathcal{E}} \deg(f_i(x)) \leqslant w_e \cdot \max_{i\in\mathcal{E}}\{\deg(f_i(x))\} \leqslant w_e \cdot \ell$. From the definition of an EEP in Eq. (4.33) the degree of $\eta(x)$ is the degree of the formal derivative of $\sigma(x)$ such that $\deg(\eta(x)) = \deg(\sigma'(x)) < \deg \sigma(x)$. Thus, Eq. (4.37) follows. □

Theorem 4.12 (Separable Binary GGC Unique Decoding Radius).
For a binary separable *generalized Goppa code $\Gamma(\mathcal{L}_\mathsf{p}, g)$ with $d \geqslant d_{\mathrm{sep}}$ the Algorithm 11*

using the Extended Euclidean Algorithm (EEA) can uniquely decode any error $\mathbf{e}$ *of weight* w_e *with*

$$w_e \leqslant w_{e_{\text{sep}}} := \left\lfloor \frac{t}{\ell} \right\rfloor = \left\lfloor \frac{d_{\text{sep}}}{2} \right\rfloor, \tag{4.39}$$

where $t = \deg(g(x))$ *and* $\ell = \max_{f(x)\in\mathcal{L}_{\mathsf{p}}} \deg f(x)$*.*

Proof. For syndrome-based unique decoding the EEA in Algorithm 2 can be used to solve the key equation. A unique solution for the ELP $\sigma(x)$ and EEP $\eta(x)$ can be found with the stopping condition in Line 5 of Algorithm 2 such that $\deg(r_\mu(x)) < \frac{\deg g(x)}{2} \leqslant \deg(r_{\mu-1}(x))$. Then, $\sigma(x) = v_\mu(x)/c$ and $\eta(x) = r_\mu(x)/c$. From Eq. (2.31) and $\frac{\deg g(x)}{2} \leqslant \deg(r_{\mu-1}(x))$ it follows $\deg v_\mu(x) \leqslant \frac{\deg(g(x))}{2}$ and thus $\deg \sigma(x) \leqslant \frac{\deg(g(x))}{2}$.
Since the separable generalized Goppa code $\Gamma(\mathcal{L}_{\mathsf{p}}, g)$ is the same code as $\Gamma(\mathcal{L}_{\mathsf{p}}, g^2)$ according to Corollary 4.8, we can apply Algorithm 11 on $\Gamma(\mathcal{L}_{\mathsf{p}}, g^2)$ to decode $\Gamma(\mathcal{L}_{\mathsf{p}}, g)$. Then, the degree constraint for uniquely decoding becomes $\frac{\deg\left(g(x)^2\right)}{2}$. Since $\deg(\sigma(x)) \leqslant w_e\ell$ we know that we can uniquely decode an error $\mathbf{e}$ of weight w_e as long as $w_e\ell \leqslant \deg(g(x)^2)/2$. Thus,

$$\deg \sigma(x) \leqslant w_e\, \ell \overset{!}{\leqslant} \frac{\deg(g(x)^2)}{2} = t.$$

It holds that $\frac{t}{\ell} < \frac{t}{\ell} + \frac{1}{(2\ell)} = \frac{d_{\text{sep}}}{2}$. In particular, $\left\lfloor \frac{t}{\ell} \right\rfloor < \left\lfloor \frac{t}{\ell} + \frac{1}{(2\ell)} \right\rfloor$ only if $2\ell \mid (2t+1)$, which is impossible for positive integers t and ℓ. Therefore, $\left\lfloor \frac{t}{\ell} \right\rfloor = \left\lfloor \frac{t}{\ell} + \frac{1}{(2\ell)} \right\rfloor$. □

Algorithm 11 Syndrome-based Decoding Algorithm for binary separable GGC

Input: received word $\mathbf{r} \in \mathbb{F}_2^n$, Goppa polynomial $g(x)$ and support $\mathcal{L}_{\mathsf{p}}$ of $\Gamma(\mathcal{L}_{\mathsf{p}}, g)$

1: Calculate the syndrome polynomial $s(x)$ according to Eq. (4.34) or from $\mathbf{s} = \mathbf{r}\mathbf{H}^\top$
2: Execute the EEA in Algorithm 2 with inputs $g(x), s(x)$ and outputs $\eta(x), _, \sigma(x)$
3: Search for the roots of $\sigma(x)$ among any root γ_i of $f_i(x)$ in the splitting field $\mathbb{F}_{q^{l_i}}$ (e.g. using Chien Search) and get $\mathcal{E} \leftarrow \{i \mid \sigma(\gamma_i) = 0\}$
4: Initialize $\mathbf{e} = \mathbf{0}$ and set $e_i \leftarrow 1,\ \forall i \in \mathcal{E}$

Output: codeword $\hat{\mathbf{c}} \leftarrow \mathbf{r} - \mathbf{e}$

4.2 Niederreiter Cryptosystem with Binary Generalized Goppa Codes

After having introduced generalized Goppa codes, we investigate a Niederreiter-based cryptosystem using binary generalized Goppa codes. A cryptosystem using binary GGC is constructed in the same way as described in Section 3.2, but with different code parameters for the code length n, the field size m, the Goppa polynomial degree

m,τ	1	2	3	4	5	6
1	2	1	2	3	6	9
2	4	6	20	60	204	670
3	8	28	168	1008	6552	43.596
4	16	120	1360	16.320	209.712	279.548
5	32	496	10.912	261.888	6.710.880	178.951.344
6	64	2016	87.360	4.193.280	214.748.352	1.14532018e+10
7	128	8128	699.008	67.104.768	6.87194765e+09	7.33007400e+11
8	256	32.640	5.592.320	1.07372544e+09	2.19902326e+11	4.69124933e+13
9	512	130.816	44.739.072	1.71798036e+10	7.03687442e+12	3.00239973e+15
10	1024	52.3776	357.913.600	2.74877645e+11	2.25179981e+14	1.92153584e+17
11	2048	2.096.128	2.86331085e+09	4.39804546e+12	7.20575940e+15	1.22978294e+19
12	4096	8.386.560	2.29064909e+10	7.03687400e+13	2.30584301e+17	7.87061080e+20
13	8192	33.550.336	1.83251935e+11	1.12589989e+15	7.37869763e+18	5.03719092e+22
14	16.384	134.209.536	1.46601550e+12	1.80143984e+16	2.36118324e+20	3.22380219e+24
15	32.768	536.854.528	1.17281240e+13	2.88230376e+17	7.55578637e+21	2.06323340e+26

Table 4.1: Number of irreducible polynomials of degree τ over $\mathbb{F}_{2^m}$

t and the error-correction capability w_e. According to Eq. (4.8) the code length n for binary GGC is upper bounded by the number of irreducible polynomials existing for a given field size. Table 4.1 shows the number of irreducible polynomials of degree τ over $\mathbb{F}_{2^m}$ according to Eq. (4.7). To receive the maximum possible code length for a given support $\mathcal{L}_\mathsf{p}$, these numbers need to be summed up according to Eq. (4.8). Classical Goppa codes (binary GGC with $\ell = 1$) are using code locators of degree $\tau = 1$ therefore the code length n is restricted to 2^m for a field size of $\mathbb{F}_{2^m}$ with the extension degree m a positive integer (see Table 4.1 Column 1). Using binary GGC with code locators of degree greater than 1, the code length n can be much higher. This makes it possible to construct much longer codes for a smaller field size with binary GGC than with binary Goppa codes. In other words, for $\ell > 1$ a smaller m compared to classical Goppa codes can be used to obtain the same code length (see Table 4.1 Column 2-6). For example, with parameters $m = 5$ and $\ell = 3$ and $l_i \leqslant 3$ for $i = 1, \ldots, n$ a maximum code length of $n_{\max_{5,3}} = 32 + 496 + 10.912 = 11.440$ is possible.

In Table 4.2, we show some examples of code parameters ($k \geqslant$, m, ℓ, t, d_{sep}) of binary separable Goppa codes and binary separable generalized Goppa codes $\Gamma(\mathcal{L}_\mathsf{p}, g(x))$, denoted by GC and GGC-$\ell$ respectively, for several values of length n.

For GGCs with a fixed code length n and fixed degree t of the Goppa polynomial, the lower bound on the minimum distance d_{sep} is reduced by the factor of ℓ, according to Corollary 4.8. For a fixed d_{sep}, the degree t must be increased to achieve the same error-correction capability. The lower bound on the dimension k is calculated by $n - mt$ and is therefore smaller for a higher degree of t.

Table 4.2 also shows the corresponding public key sizes of Classic McEliece and the corresponding expected work factor for an ISD attack by Eq. (3.6) of [AM87]. The public key size is determined by the systematic form of $\mathbf{H}^{\text{bin}}$ with $\mathbf{H}_{\text{sys}} = (\mathbf{I}_{n-k} \mid \mathbf{T})$

Code	n	$k \geqslant$	m	ℓ	t	d_{sep}	$\text{WF}_{\text{Prange}} \approx$	$|\mathsf{pk}|$ [bytes]
GC	3488	2720	12	1	64	128	2^{177}	261.120
GGC-2	3488	3040	7	2	64	64	2^{162}	170.240
GGC-2	3488	2585	7	2	129	129	2^{130}	291.782
GC	6960	5413	13	1	119	239	2^{301}	1.047.319
GGC-2	6960	6127	7	2	119	119	2^{222}	637.974
GGC-3	6960	5170	5	3	358	239	2^{275}	1.156.788
GC	8192	6528	13	1	128	257	2^{339}	1.357.824
GGC-2	8192	7296	7	2	128	128	2^{245}	817.152
GGC-8	8192	6528	2	8	832	208	2^{281}	1.357.824

Table 4.2: Code parameters for binary separable GGCs with corresponding work factor of Prange (Eq. (3.6)) and public key size for Classic McEliece. Table adapted from [LPZW21] and extended.

according to Eq. (3.2) and the size of $\mathbf{T}$ is

$$|\mathsf{pk}| = (nmt - m^2t^2)/8 \tag{4.40}$$

bytes. Thus, a higher n leads to a larger public key for fixed m and t. To meet the NIST security requirements, Classic McEliece needs to choose code parameters that result in a large public key size compared to other NIST KEM submissions. That is a drawback in computation, storage and transmission. Therefore, it is desirable to reduce the public key size while retaining the requested security strength. As Classic McEliece contains big public keys compared to other PQC KEMs we want to search for code parameters in GGC that can reduce the public key size by retaining the security levels chosen by the submitters. We investigate how generalized Goppa codes can help in reducing this drawback.

For estimating the security level, e.g. work factor of ISD (see also Section 3.5.2) we take the memory-free algorithm of Prange [Pra62] with the work factor specified in Eq. (3.6) [AM87].[2] The security level is represented by $\text{SL}_{\text{Prange}} = \log_2(\text{WF}_{\text{Prange}})$ with $\text{WF}_{\text{Prange}} = k^3\binom{n}{k}/\binom{n-w_e}{k}$ the work factor for $a = 3$ in Eq. (3.6) and $w_e = \lfloor d_{sep}/2 \rfloor$. It can be seen from the formula that a higher dimension k and a higher error-correction capability $w_{e_{\text{sep}}}$ result in a higher work factor. For CAT-1 parameters in Table 3.3 the security level is $\text{SL}_{\text{Prange}} \approx 177$ and for CAT-3 parameters it is $\text{SL}_{\text{Prange}} \approx 220$. The goal is to find code parameters for GGC that improve the public key size.

We focused on the CAT-1 security level and calculated the public key size for all possible t and n given the field size m and the maximum degree ℓ for GGC that achieve the security level of at least 177. As a smaller n brings a smaller public key according to Eq. (4.40) thus, we stop the calculation if we found code parameters achieving the security level for fixed t. Fig. 4.1 shows the smallest public key sizes

[2] As introduced in Section 3.5.2, improved ISD algorithms exist. They are not memory-free, making comparisons more complicated, as both runtime and memory usage need to be taken into account. At their core, they rely on the same idea as Prange, hence we expect similar behaviour.

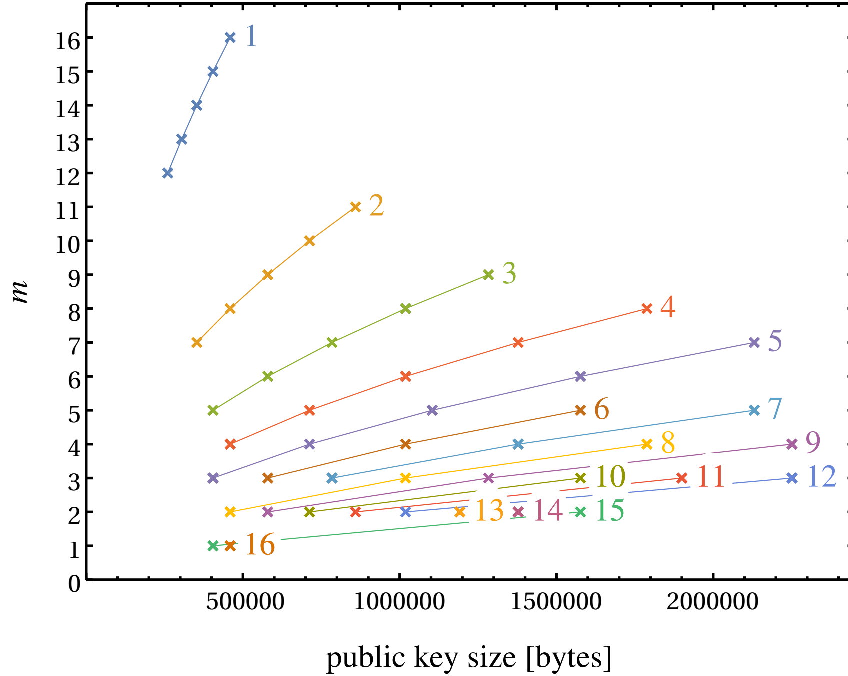


Figure 4.1: Smallest public key sizes for given ℓ (in different colours) and m achieving a security level of at least 177. (Corresponding code parameters are listed in Table 4.3)

possible for given extension degree m of $\mathbb{F}_{q^m}$ and maximum degree ℓ that achieve at least the security level of CAT-1 parameters of 177. It can be seen that the smallest public key for the given security level of 177 is reached with $\ell = 1$ and $m = 12$ and that corresponds to classical Goppa codes. The smallest possible public key has 260.190 bytes. That differs from the parameters chosen by the submitters for CAT-1 with a size of 261.120 bytes, because the submitters chose the parameters such that they are a multiple of 2, 8 or 32 and $t = 64$ can be better implemented on classical binary computers than $t = 59$. The observation that the smallest public key sizes can be achieved with classical Goppa codes $\ell = 1$ is also valid for other security levels. Thus, GGC do not give an advantage in reducing the public key size for given security level. Classical Goppa codes are still the best code parameters in terms of smallest public key size for given security level. However, the field size $\mathbb{F}_{2^m}$ can be reduced using separable GGC by tolerating a small increase of the public key size. So, to minimize the field size it is best to choose either $\ell = 2$ with $m = 7$ for a public key size of 353.012 bytes or to construct a cryptosystem with $m = 3$ and $\ell = 5$ for a public key size of 404.550 bytes. For $m = 1$ and $m = 2$ [COT17] there exist breaks for classical Goppa codes distinguishing the code from a random code that can most likely be applied also to GGC. Note that parameters n and t must also be big enough to prevent brute-force key attacks on the public key as specified in Section 3.5.1.

Looking at Fig. 4.1 raises the question of why increasing ℓ by 1 sometimes makes smaller public key sizes possible. The answer lies in Eq. (4.8) and the values given in Table 4.1. The given security level for $\ell = 1$ is reached with a code length $n = 3648$.

To achieve this code length the minimal extension degree must be $m = 12$ (Table 4.1 Column 1, Row 12 gives a value of 4096 which is the first value in this column that is bigger than 3648). For higher $\ell > 1$ it can be estimated which minimal m can bring a code length of at least 3648. For $\ell = 2$ this is $m = 7$ as the sum of Row 7, Column 1 and Row 7, Column 2 in Table 4.1. An $m = 6$ would be too small to achieve the security level as the maximum code length is restricted to $64 + 2016 = 2080$ in Row 6. For $\ell = 3$, the smallest m for which we can achieve an $n > 3648$ is $m = 5$. The same applies for $\ell = 4$, the smallest m achieving the security level is with $m = 4$. Thus, to achieve an $m = 1$ we need at least $\ell = 15$ to construct codes with a length of at least 3648. Codes with smaller n than 3648 do not achieve the security level of 177 based on the work factor specified above.

Table 4.3 shows the code parameters corresponding to the plot in Fig. 4.1 representing the smallest public key for given ℓ and m.

Conclusion In this chapter, Generalized Goppa Codes (GGC) have been discussed and investigated for use in conjunction with a Niederreiter cryptosystem. We found out that these codes do not bring a smaller public key size for the same security level compared to classical Goppa codes. We developed a construction of parity-check matrices for binary GGC with code locators of arbitrary degrees of polynomials. We derived a bound on the minimum distance for binary GGC and separable binary GGC. For separable GGC and even degree polynomials of the support the lower bound on the minimum distance is slightly increased. A decoding algorithm for GGC has been presented. By using the EEA for decoding separable GGC, it has been shown that the unique decoding radius is increased to $\left\lfloor \frac{d_{\text{sep}}}{2} \right\rfloor$. We set up Classic McEliece with GGC and searched for code parameters that decrease the public key size for the same security level as for the parameters chosen for classical Goppa codes using the ISD work factor of Prange. We showed that it is not possible to find parameters for binary GGC that improve the public key size compared to classical Goppa codes. Classical Goppa codes still show the best performance regarding public key size for given security level. However, by tolerating a slightly larger public key size, the field size can be reduced appreciably.

For future work, it is interesting to look into q-ary GGCs that operate on $\mathbb{F}_{q^m}$ for $q > 2$. A further direction of research is to look for supercodes of GGCs, i.e. a code that contains GGCs as subcodes, if any exist.

ℓ	m	t	n	$\mathrm{SL}_{\mathrm{Prange}}$	|pk| [bytes]
1	12	59	3648	177,023	260.190
1	13	59	3947	177,024	304.883
1	14	59	4245	177,027	353.012
1	15	58	4590	177,031	404.550
1	16	58	4888	177,017	459.360
2	7	118	4245	177,027	353.012
2	8	116	4888	177,017	459.360
2	9	116	5483	177,024	579.290
2	10	116	6074	177,019	712.530
2	11	116	6662	177,015	859.067
3	5	174	4590	177,031	404.550
3	6	174	5483	177,024	579.290
3	7	174	6368	177,012	784.088
3	8	174	7248	177,019	1.018.944
3	9	171	8210	177,022	1.283.334
4	4	232	4888	177,017	459.360
4	5	232	6074	177,019	712.530
4	6	232	7248	177,019	1.018.944
4	7	228	8503	177,020	1.377.947
4	8	228	9670	177,016	1.788.888
5	3	290	4590	177,031	404.550
5	4	290	6074	177,019	712.530
5	5	290	7540	177,020	1.103.813
5	6	285	9087	177,013	1.576.834
5	7	285	10541	177,018	2.131.159
6	3	348	5483	177,024	579.290
6	4	348	7248	177,019	1.018.944
6	5	342	9087	177,013	1.576.834
7	3	399	6438	177,013	784.185
7	4	399	8503	177,020	1.377.947
7	5	399	10541	177,018	2.131.159
8	2	464	4888	177,017	459.360
8	3	464	7248	177,019	1.018.944
8	4	456	9670	177,016	1.788.888
9	2	522	5483	177,024	579.290
9	3	513	8210	177,022	1.283.334
9	4	513	10830	177,015	2.251.557
10	2	580	6074	177,019	712.530
10	3	570	9087	177,013	1.576.834
10	4	560	12117	177,012	2.765.560
11	2	638	6662	177,015	859.067
11	3	638	9856	177,018	1.900.124
12	2	696	7248	177,019	1.018.944
12	3	684	10830	177,015	2.251.557
13	2	754	7831	177,016	1.191.886
13	3	741	11696	177,016	2.632.310
14	2	798	8503	177,020	1.377.947
15	1	870	4590	177,031	404.550
15	2	855	9087	177,013	1.576.834
16	1	928	4888	177,017	459.360
16	2	912	9670	177,016	1.788.888
17	1	986	5186	177,022	517.650

Table 4.3: Code parameters corresponding to Fig. 4.1

Chapter 5

Accelerations using the RISC-V Vector Extension

The security of cryptosystems is the foundation for establishing secure communication channels between multiple parties. For employing the cryptosystems on real devices the performance of the algorithms are also important, especially computational complexity and resource consumption. The reduction of the memory consumption due to the big public key used in Classic McEliece was investigated in Chapter 4. In this chapter, the reduction of the computational complexity is researched. For this, it is necessary to look at the actual implementations of Classic McEliece. These are introduced in Section 5.1.

We examined the different implementations and there exists source code of the algorithms specially written for readability and algorithms specially designed for performance. In particular there are vectorized accelerations for x86/AMD64 ISA processors. Vectorization is a general strategy to speed up computations whenever the same operation is applied to large chunks of data. Processors that support the necessary vector operations can apply them to a whole vector of data at once, instead of cycling through the data points individually. A performance profiling of these implementations showed that the Gaussian Elimination Algorithm (GEA) is the most time consuming algorithm of the cryptosystem. This is reported in Section 5.2. The GEA is investigated in detail in Section 5.3. We find that it is possible to accelerate the GEA by vectorization to which the rest of the chapter is dedicated.

The fast execution of cryptographic algorithms is important on any system and the vectorization of the GEA is expected to result in speed-up on processors integrating any Instruction Set Architecture (ISA), e.g. x86 ISA. An ISA is a description of a processor architecture which defines the machine code that a processor core can understand and also how instructions are interpreted and executed. Based on the ISA, the core is developed and its layout defined. Assembly code contains ISA specific instructions that are readable by humans. Therefore, source code that was compiled to a binary executable for a specific ISA can usually only be understood by a core integrating the same ISA.

We demonstrate the vectorization of the GEA on the RISC-V ISA. RISC-V is a simple and easy to understand ISA because of its reduced set of instructions, compared

to the x86 and AMD64 ISAs which is complex. We use RISC-V and RISC-V Vector Extension (RVV) to vectorize the GEA. In Section 5.4 and in Section 5.4.1 the RISC-V ISA and its vector extension are presented. The RVV was under development at the time this study was conducted. Only a few industrial architectures providing RVV had been announced, e.g. Andes' NX27V, SiFive's VIU75 and Alibaba's Xuantie-910 chipsets. As these platforms were yet to become available, this work showed how these new RVV-extended processor generations can be used to accelerate Classic McEliece using a rapid prototyping approach. In Section 5.5 the GEA using RVV is presented. In Section 5.6 we simulate and verify the correctness of our vectorized implementation. As simulator the Extendable Translating Instruction Set Simulator (ETISS) [MDG+17] is used. We evaluate the performance for an architecture with and without RVV for different memory interfaces.

The content of Section 5.2, Section 5.3, Section 5.5, Section 5.6 is published in [PGZM21]. Section 5.6 as well as the numerical results of Section 5.5 are a joint work with Johannes Geier. He developed the "SoftVector" library, the simulation environment on ETISS and obtained the final simulation results. More details on the simulation environment can be found in the paper. At the time this study was conducted, the most recent version of Classic McEliece implementation has been of Round 2 [BCL+19]. Therefore, we worked on the Round 2 implementations.

Our simulation code and environment, as well as the RVV accelerated GEA source code can be found in `https://github.com/tum-ei-eda/rvv-gauss-demo`.

5.1 Overview of Classic McEliece implementations

As required by NIST each submission needs to attach an implementation of their proposed system to the submission package. The Classic McEliece submission package of Round 2 consists of the following implementations for each parameter set specified in Table 3.3 :

- Reference/Optimized implementations: platform independent implementations in C-code
- Additional implementations[1]: x86 ISA processors specific implementations for reference and optimized implementation in C-code and assembly-code using the
  - Streaming Single Instruction Multiple Data (SIMD) Extension (SSE)
  - Advanced Vector Extension (AVX)

The reference/optimized implementation is described in Section 5.1.1. The additional implementations are presented in Section 5.1.2. Since Round 3, the submission package also includes a platform independent vectorized implementation, that uses 64 bit `long long` types.

[1]Since Round 3 of the NIST competition there is also an additional implementation that uses the VEC Extension.

For each parameter set in Table 3.3 there exist two different key generation algorithms. One calculates the systematic form of Line 5 in Algorithm 3 and the other calculates a semi-systematic form which are called as *f variants*. In the variant of the systematic form, the algorithm in Algorithm 3 is executed until a keypair is found resulting with the public key being in systematic form. This takes about three trials for Line 5 in Algorithm 3, for each trial having a newly generated Goppa polynomial. To minimize this number of trials, the *f variants* key generation algorithm swaps code locators in Line 5 of Algorithm 3 to achieve a systematic form without selecting a new Goppa polynomial and code locators, making it possible to generate valid secret/public keypairs already at the first trial. In order to revert the swapping for decryption, the swapping parameters are stored in the secret key.

5.1.1 Platform independent implementations

The platform independent implementations are written in plain C-code. The optimized implementations are an exact copy of the reference implementations.

The reference implementation integrates the following main procedures with the corresponding file names in parenthesis:

– Galois Field Operations in $\mathbb{F}_2$ and $\mathbb{F}_{2^m}$ (`gf.c`)
– Key Generation of Algorithm 3 (`operations.c`)
    - Secret key generation (`sk_gen.c`)
    - Public key generation (`pk_gen.c`)
– Encapsulation of Algorithm 5 (`operations.c`)
    - Encryption (`encrypt.c`)
– Decapsulation of Algorithm 6 (`operations.c`)
    - Decryption (`decrypt.c`)
    - Syndrome computation (`synd.c`)
    - Polynomial evaluations (`root.c`)
    - Berlekamp-Massey Decoder (`bm.c`)
– Beneš network (`benes.c`)
– Controlbits for Beneš network (`controlbits.c`)

The Beneš network is used to represent the code locators $\alpha_1, \ldots, \alpha_n$ in $\mathcal{L}$ in a different way for simpler fast constant-time decoding. The code locators are described as the controlbits of the Beneš network.

The secret key is stored as a concatenation of bits constituting of the seed used for randomness, the column selections (*f variants*) or zeros (*non-f variants*), the Goppa polynomial $g(x)$, the controlbits and the string s.

5.1.2 AVX/SSE Implementations

The AVX/SSE implementations only run on processors that integrate the x86 Instruction Set Architecture (ISA) with an AVX or SSE extension. The x86 ISA is a Complex Instruction Set Computer (CISC) and has been initially developed by Intel Corporation. CISCs processors implement an architecture where a single instruction can execute multiple functions, e.g. load from memory, execute arithmetic operation and store result back to memory. While Reduced Instruction Set Computers (RISCs) are designed to have less and simplified instructions with each instruction executing only one function, e.g. either load, arithmetic operation or store. Compared to CISCs, the RISCs may need to perform more instructions for the same task. Processors that are implementing the base x86 ISA operate on 16 bit, 32 bit or 64 bit wide General Purpose Registers (GPRs). By integrating instruction set extensions, additional registers and instructions are available on hardware for more complex computations. Usually, using instruction set extensions accelerates the computation of tasks compared to computations on base ISA only, as computations have access to additional hardware that is used concurrently with base ISA hardware. The SSE and AVX extend the base ISA with 128 bit or 256 bit registers.

The Streaming Single Instruction Multiple Data (SIMD) Extension (SSE) is an instruction set extension for x86 ISA processors developed by Intel Corporation. The first version of SSE contains eight registers holding 128 bits and instructions that operate on these registers. Using a single SSE instruction, the operation can be applied to all 128 bits simultaneously. The following versions are extensions that integrate more additional instructions and more registers than the earlier versions. Thus, higher versions include lower versions. In Classic McEliece the SSE implementation is written for version SSE4.1 having registers for 128 bits.

The Advanced Vector Extension (AVX) is also an instruction set extension for x86 ISA processors developed by Intel Corporation. But, compared to SSE, the AVX contains 16 registers holding 256 bits and defines instructions that operate on these registers. An expansion of AVX is AVX2 which integrates additional functionality, e.g. vector shifts. In Classic McEliece the AVX implementation is written for version AVX2 and automatically also implements the SSE version 4.1 such that computations are executed concurrently on 128 bit and 256 bit registers.

Intrinsics are an application programmable interface in C-code that correspond to the instruction in assembly code, but can be called in C-code. Thus, there is no need of including explicit assembly code and the programmer does not need to care about registers and can use variables instead, e.g. instead of writing and knowing the operands, the programmer can just call the function, and the intrinsics interface is taking care of input and output.

In the AVX implementations the ELP is differently represented in memory than in the platform independent implementations, which affects the simulations in Section 6.3.1 of Chapter 6.

5.2 Performance Analysis of Implementations

A performance analysis of the reference and AVX implementations of Round 2 show the time consuming parts in the algorithm. The analysis was done using the software program Callgrind and KCachegrind [Wei] on a Linux 64 bit, Intel i7-8650U @1.9GHz machine. The key generation algorithm takes much longer than encryption or decryption. Thus, we analyzed the key generation algorithm and present the important parts for two parameter sets in Table 5.1 and Table 5.2. In the reference implementation, the construction of controlbits for the Beneš network, which is needed for a simplified fast constant-time decoding algorithm, makes up a big portion of the key generation. This is more evident in the case for parameters $n = 6960, m = 13, t = 119$ with a 62% share in Table 5.2 compared to parameters $n = 8192, m = 13, t = 128$ with a 40% share in Table 5.1. In both cases it can be seen that the use of platform specific ISA extensions accelerates the construction of the controlbits to 10% and 8% share, respectively. The Galois Field multiplication (GF mul) takes about one eighth of the key generation in the reference and in the AVX accelerated implementation. In the AVX reference implementation the GEA dominates the running time in key generations and is thus the most time consuming operation in the cryptosystem.
As the AVX implementation is designed for performance, while the reference implementation is designed for clarity and easier readability, we focus on accelerating the GEA.

	Reference implementation [%]	AVX implmementation [%]
GEA	41	67
GF mul	13	18
Controlbits	40	8
Others	6	7

Table 5.1: Profiling of Keypair Generation for Classic McEliece with parameters 8192128 on Linux 64 bit, Intel i7-8650U @1.9GHz

	Reference implementation [%]	AVX implementation [%]
GEA	26	73
GF mul	8	12
Controlbits	62	10
Others	4	5

Table 5.2: Profiling of Keypair Generation for Classic McEliece with parameters 6960119 on Linux 64 bit, Intel i7-8650U @1.9GHz

5.3 Gaussian Elimination Algorithm (GEA)

The GEA is needed in Step 1 and in Step 5 of Algorithm 3. For the systematic form key generation algorithms, the $mt \times n$ parity check matrix $\hat{\mathbf{H}}$ of the Goppa code is brought into systematic form

$$(\mathbf{I}_a \mid \mathbf{T}) \tag{5.1}$$

where $\mathbf{I}_a$ is a $a \times a$ identity matrix and $\mathbf{T}$ is a $a \times (b-a)$ matrix with $a = n-k = m \cdot t$ and $b = n$. This is a row-reduced echelon form such that an identity matrix is formed in the left-most $a \times a$ block of the matrix. The systematic form reduces the key size of the public key from $(n-k) \times n$ to $(n-k) \times k$ and scrambles the matrix such that the secret keys cannot be recovered from the public key $\mathbf{T}$ (see also Section 3.5.1). Not every matrix can be brought to systematic form. If it is not possible, Step 5 of Algorithm 3 fails and the key generation process needs to start over, as described in Section 5.1.

The pivot element is an element on the diagonal of the matrix. The pivot row is the row of the current pivot element. To form the identity matrix, the algorithm needs to iterate through the rows of the matrix and perform two operations:

1. Check if pivot element is a 1. If not, form it to a 1.

2. Bring all elements below and above the pivot element to a 0.

Operation 1 is realized by dividing the whole row by the value of the pivot element to get a 1. If the pivot element is a 0, the pivot element needs to be formed to non-zero first. This is done by adding another row to the pivot row. If that does not succeed, the algorithm fails and a new matrix must be generated.

Operation 2 is realized by subtracting the pivot row from all other rows. If the matrix is not binary, the pivot row is subtracted with a factor, which sets the element to a 0. Not every parity-check matrix of size $a \times b$ with $a = m \cdot t$ and $b = n$ can be transformed into its systematic form. This is only possible if the right-most $(b-a) \times (b-a)$ block of the matrix consists of linearly independent columns. That is the case in about 1/3 of the parity-check matrices for Goppa codes. The procedure is described in Algorithm 12. It operates on a binary matrix and is constant time.

The implementation of Step 5 of Algorithm 3 operates on a binary matrix with a rows and b columns, specified in Algorithm 12. For Operation 1 every other row is added to the pivot row if the pivot element is a 0. This is needed such that the algorithm is constant time and impedes timing attacks. The complexity of Algorithm 12 is $O(a^2 \cdot b)$ with a rows and b columns. By analysing Algorithm 12 we identify the approximate number of load and store accesses on the matrix of size $a \times b$ in Eq. (5.2).

Algorithm 12 Gaussian Elimination Algorithm (GEA)

Input: binary matrix $\mathbf{A}$ of size $a \times b$
Output: binary matrix $\tilde{\mathbf{A}}$ in form $(\mathbf{I_a}|\mathbf{T})$ or $\perp$

```
 1: for i = 0 → (a+7)/8 − 1 do
 2:     for j = 0 → 7 do
 3:         row ← i·8+j
 4:         // Operation 1
 5:         for k = i + 1 → a-1 do                          ▷ k1-loop
 6:             mask ← A[row,i] ⊕ A[k,i]
 7:             mask ← mask ≫ j
 8:             mask ← mask ∧ 1
 9:             mask ← -mask
10:             for column c = i → b/8 − 1 do                ▷ c1-loop
11:                 A[i,c] ← A[i,c] ⊕ A[k,c] ∧ mask
12:         if A[i,i]≠ 1 then
13:             Failure and stop of algorithm
14:         // Operation 2
15:         for k = 0 → a-1 do                              ▷ k2-loop
16:             if k ≠ row then
17:                 mask ← A[k,i] ≫ j
18:                 mask ← mask ∧ 1
19:                 mask ← -mask
20:                 for column c = 0 → b/8 − 1 do            ▷ c2-loop
21:                     A[k,c] ← A[k,c ] ⊕ A[row,c] ∧ mask
```

$$
\begin{aligned}
N_{a,b} = {} & a \cdot \left(1 + \underbrace{(a-1)}_{\text{k2-loop}}\left(1 + \underbrace{3 \cdot \frac{b}{8}}_{\text{c2-loop}}\right)\right) + \underbrace{2 \cdot \frac{a(a-1)}{2}}_{\text{k1-loop}} + \\
& \underbrace{+\, 3 \cdot \sum_{j=1}^{\lfloor \frac{a}{8} \rfloor} \left(\frac{(8j+2)(8j+3)}{2} - \frac{(8j-6)(8j-5)}{2}\right) \cdot \left(\frac{b}{8} - \left\lfloor \frac{a}{8} \right\rfloor + j\right)}_{\text{c1-loop}} .
\end{aligned}
\tag{5.2}
$$

The first term of Eq. (5.2) describes one memory access in Line 12 and Line 17 of Algorithm 12 respectively and three memory accesses in Line 21 of Algorithm 12. The second term accounts for two memory accesses in Line 6 of Algorithm 12. The last term describes three memory accesses in Line 11 of Algorithm 12.

Example 5.1.
For system parameters $n = 8192$*,* $t = 128$ *and* $m = 13$ *the matrix is of size* $a = 1664$

and $b = 8192$ for the GEA. Then, from Eq. (5.2) we obtain

$$N_{1664,8192} = 8503 \cdot 10^6 + 2 \cdot 10^6 + 3978 \cdot 10^6 \approx 12.5 \cdot 10^9. \tag{5.3}$$

In Section 5.5 the inner loops (c1-loop and c2-loop) are replaced by a single vector arithmetic operation that reduce the number of memory accesses using the RVV. RISC-V and the vector extension are introduced in the next section.

5.4 Introduction to RISC-V

RISC-V is an open-source Instruction Set Architecture (ISA) that is based on RISC instructions. It was initially developed by Andrew Waterman, Yunsup Lee, Krste Asanović and David Patterson from University of California, Berkeley in 2010 with the idea to design the first ISA that is freely available and not protected by patents. So, the RISC-V ISA can be integrated in any processor without having to pay licence fees. Since then, the RISC-V ISA gained much attention and many developers and companies support the development of the open-source project. In 2015, the RISC-V international non-profit organization took over the worldwide management of the fast-growing community. The RISC-V ISA is known for its minimalistic, simple and flexible design. The goal of RISC-V is to be a universal ISA, that fits any kind of processor, from the smallest microcontroller to the most powerful supercomputer. It should be suitable for all kind of architectures like Field Programmable Gate Arrays (FPGAs) and Application-Specific Integrated Circuits (ASICs).

The minimal requirement for a RISC-V processor is to have hardware and gate logic that integrates a base integer ISA. A base integer ISA constitutes an architecture that supplies basic operations like additions, shifts, branches, comparisons. It can operate on either 32 bits, 64, bit or 128 bit integer values. These values can be stored in 31 general-purpose base ISA registers denoted by `x1[XLEN] - x31[XLEN]` with `XLEN` referring to the size of the register in bits. The register `x0[XLEN]` is hardwired to zero. For resource constrained embedded devices there may also only 16 registers be implemented. The base integer ISA is denoted by `RV` followed by the value of `XLEN` and an `I` for integer with 32 registers or `E` for embedded with 16 registers. For example, `RV64I` stands for the RISC-V base instruction set for 64 bit processors. To realise more complex operations like multiplications, floating point calculations or vector operations, the base architecture needs to be extended by additional instructions. These additional instructions are grouped into different extensions. For example, the instructions for integer multiplication and division are grouped in the standard extension for Integer Multiplications and Divisions, denoted by an `M`. In this work the 64 bit base integer ISA with the extension for integer multiplication and division `RV64IM` is used. The following section is occupied with the RISC-V Vector Extension (RVV), denoted with a `V`, for example `RV64IMV`.

5.4.1 RISC-V Vector Extension (RVV)

A RISC-V organised volunteer task force defines and develops the RVV. In 2020, they established the RVV version 0.9 [AKA+20] which this work rests on. The RVV is very flexible compared to SSE and AVX. The length of the registers in SSE and AVX are fixed to a length of 128 bits and 256 bits respectively, while in RVV the length of the registers can be freely chosen up to 2^{16} bits (ratified version 1.0).

The vector extension adds 32 vector registers and 7 unpriviledged Control Status Registers (CSRs) to a base integer RISC-V ISA. The vector registers are denoted by

$$\texttt{v0[VLEN] - v31[VLEN]} \tag{5.4}$$

with `VLEN` the number of bits that can be stored in the registers. The value of `VLEN` must be a power of 2 and greater or equal than the parameter `ELEN`. The parameter `ELEN` defines the maximum size of a single vector element in bits on that can be operated on. The value of `ELEN` must also be a power of 2 and additionally be greater or equal than 8. The values of `VLEN` and `ELEN` are fixed by the hardware and cannot be changed by software.

The parameter `SEW` denotes the dynamic standard element width, which is set during computations and is upper bounded by `ELEN`. By default, a vector register is viewed as being divided into `VLEN/SEW` standard-with elements. The parameter `LMUL` specifies how many vector registers are grouped together such that a single vector instruction operates on this group of registers. In other words, `LMUL` indicates on how many registers one vector is spanned. Then, a vector length `VL` can be set, which refers to the number of elements of size `SEW` bits inside one vector. The following instructions configure these values.

$$\texttt{vsetvli rd, rs1, vtypei} \tag{5.5}$$

$$\texttt{vsetvl rd, rs1, rs2} \tag{5.6}$$

`rd` and `rs1` stands for a register from the base ISA. In `rs1` the value of `VL` is stored, that should be set by the instruction. In `rd` the return value of the instruction is stored. In the normal case, where the content of `rs1` is not zero, the return value is the set `VL`. `vtypei` specifies `SEW` and `LMUL` values separated with commas. For `SEW = X` the character `eX` and for `LMUL = Y` the characters `eY` must be set. For example, for `SEW = 8` and `LMUL = 8` the instruction is `vsetvli t0, a2, e8, m8` . As alternative to `vsetvli` the instruction `vsetvl` can be invoked to set the `SEW` and `LMUL` values via a separate register `rs2`.

The vector extension supports different kind of load and store operations. In this work only the unit-strided addressing modes are used. To load and store values to and from the vector registers `v0 - v31`, the following instructions are used.

$$\texttt{vleX.v vd, (rs1), vm} \tag{5.7}$$

$$\texttt{vseX.v vs3, (rs1), vm} \tag{5.8}$$

Used RVV instructions:

vsetvli rd, rs1, vtypei	Sets SEW, LMUL and VL
vle8.v vd, (rs1), vm	Unit-strided load of VL ∗ 8 bits
vse8.v vs3, (rs1), vm	Unit-strided store of VL ∗ 8 bits
vand.vi vd, vs2, imm, vm	Bitwise logical AND with immediate parameter
vand.vx vd, vs2, rs1, vm	Bitwise logical AND with base ISA register parameter
vand.vv vd, vs2, vs1, vm	Bitwise logical AND with vector register parameter
vxor.vi vd, vs2, imm, vm	Bitwise logical XOR with vector register parameter
vxor.vx vd, vs2, rs1, vm	Bitwise logical XOR with base ISA register parameter
vxor.vv vd, vs2, vs1, vm	Bitwise logical XOR with vector register parameter

With vd the vector register destination for load operations is specified. The vector register vs3 holds the data that should be stored on memory. In rs1 the base address of the memory for loads and stores is kept. Unit-strided load and stores access elements stored continuously in memory starting form the base effective address given in rs1. The vm flag specifies whether vector masking is enabled or not. Masking is used to modify the execution of a vector instruction, i.e. to apply the instruction only on selected vector elements. The mask value is always supplied by the vector register v0, where the mask bit for the vector element i is located in bit i of v0. If vm is given as v0.t, then masking is enabled. Otherwise, when vm is missing then masking is disabled. The character X in load/store instructions is usually set to the value of SEW, e.g. for 8 it is vle8.v . Roughly said, it specifies how many bits multiplied by VL are moved to/from memory.

As vector bitwise logical instructions the XOR and AND operations are used. There exist three different instructions for each bitwise logical operation, depending on whether the value is passed directly to the instruction, or given inside a base ISA register or vector register.

$$\texttt{vand.vi vd, vs2, imm, vm} \tag{5.9}$$

$$\texttt{vand.vx vd, vs2, rs1, vm} \tag{5.10}$$

$$\texttt{vand.vv vd, vs2, vs1, vm} \tag{5.11}$$

$$\texttt{vxor.vi vd, vs2, imm, vm} \tag{5.12}$$

$$\texttt{vxor.vx vd, vs2, rs1, vm} \tag{5.13}$$

$$\texttt{vxor.vv vd, vs2, vs1, vm} \tag{5.14}$$

The instruction vand.vi calculates the bitwise logical AND operation of a vector in vs2 and a 5-bit immediate scalar that is encoded in imm. In the vxor.vx instruction the scalar value is taken from a base ISA register rs1. If XLEN>SEW, the least-significant bits of length SEW of rs1 are used for the operation. The instruction vand.vv calculates the logical AND of two vectors vs1 and vs2. The same structures

apply to the XOR instructions.

Example 5.2.
The following example code in assembly instructions for a RVV with `VLEN=1024` *calculates Line 11 to Line 13 of Algorithm 13 with vectors of length* `VL = 870` *and thus* $b = 6960$*.*

```
li a2, 870                  # set vector length
vsetvli t0, a2, e8, m8      # e8: SEW=8, m8: LMUL=8
vle8.v v8,(rs1)             # Unit-strided load of VL*8-bit
vle8.v v16,(rs2)            # Unit-strided load of VL*8-bit
lbu a5, (rs3)               # Load operand from memory
vand.vx v24,v8,a5, v0.t     # v24 = v8 & operand
vxor.vv v16,v16,v24         # v16=v16^v24
vse8.v v16,(rs2)            # Store row with VL*8-bit from v16 to
    memory address rs2
```

5.5 Acceleration of GEA with RVV

In this section the GEA is accelerated by RVV version 0.9 as specified in [AKA+20] using an assembly-level implementation (nearly all instructions used here remained unmodified also in the RVV ratified version 1.0). As no intrinsics (see Section 5.1.2) for RVV existed at the time this study was conducted, the GEA of the C-code was extracted and compiled to RISC-V 64 bit assembly using the GNU Compiler Collection (GCC). Based on this RISC-V assembly, an analysis of the GEA showed where an integration of RVV could accelerate the algorithm. The code locations marked with “Vector arithmetic” in Algorithm 13 are identified for RVV use.

We adapt the GEA in Algorithm 12 such that the inner loops of Algorithm 12 are replaced by two vector bitwise logical operations: `vxor.vv` and `vand.vv`. The resulting algorithm is shown in Algorithm 13.

For a Classic McEliece cryptosystem with system parameters of $n = 8192, t = 128$ and $m = 13$, the parity check matrix for the GEA is of size $a = 1664$ and $b = 8192$. With a vector register length of $\texttt{VLEN} = 1024$ bits, a standard element width of $\texttt{SEW} = 8$, $\texttt{LMUL} = 8$ and thus a vector length of $\texttt{VL} = 870$ one single row of the 1664×8192 bits matrix can be stored into one vector register group. Hence, one vector represents one row of the parity check matrix. Thus, load/stores and operations can be executed by one single instruction. This is possible for $\texttt{VLEN} * 8 \geqslant b$. Then, one vector register group consists of 8 vector registers of length `VLEN` and can store in total `VLEN`*8 bits. The vector extension has 32 vector registers. Therefore, for the above parameters a maximum of 4 matrix rows can be simultaneously stored in vector registers. If the mask register `v0` is used, then it reduces to 3 matrix rows in 3 vector register groups, starting at `v8`, `v16` and `v24`.

The source code of the accelerated GEA with RVV inline assembly can be found in Appendix A.

Algorithm 13 GEA with Vector Extension

Input: binary matrix $\mathbf{A}$ of size $a \times b$
Output: binary matrix $\tilde{\mathbf{A}}$ in form $(\mathbf{I_a}|\mathbf{T})$ or $\perp$

```
 1: for i = 0 → (a+7)/8 − 1 do
 2:     for j = 0 → 7 do
 3:         row ← i·8+j
 4:         Load A[row] into vector register
 5:         // Operation 1
 6:         for k = i + 1 → a-1 do                                    ▷ k1-loop
 7:             mask ← A[row,i] ⊕ A[k,i]
 8:             mask ← mask ≫ j
 9:             mask ← mask ∧ 1
10:             mask ← -mask
11:             Load A[k] into vector register
12:             A[row,i:b-1] ← A[row,i:b-1] ⊕ A[k,i:b-1] ∧ mask   ▷ Vector arithmetic
13:             Store A[row] from vector register to memory
14:         if A[i,i]≠ 1 then
15:             Failure and stop of algorithm
16:         // Operation 2
17:         for k = 0 → a-1 do                                        ▷ k2-loop
18:             if k ≠ row then
19:                 mask ← A[k,i] ≫ j
20:                 mask ← mask ∧ 1
21:                 mask ← -mask
22:                 Load A[k] into vector register
23:                 A[k] ← A[k] ⊕ A[row] ∧ mask                   ▷ Vector arithmetic
24:                 Store A[k] from vector register to memory
```

The memory access savings of the accelerated GEA can be estimated. Assuming a rows and b bits per row, the approximate number of load and store accesses using the vector extension is

$$
\begin{aligned}
N^V_{a,b,W} = a \cdot \Bigg(1 + \underbrace{(a-1)}_{\text{k2-loop}}\Bigg) + \underbrace{2 \cdot \frac{a(a-1)}{2}}_{\text{k1-loop}} \\
+ \left\lceil \frac{b}{W} \right\rceil \cdot \Bigg(\underbrace{a}_{\text{vec. i-th row}} + \underbrace{2\frac{a(a-1)}{2}}_{\text{vec. k1-loop}} + \underbrace{2a(a-1)}_{\text{vec. k2-loop}} \Bigg),
\end{aligned}
\tag{5.15}
$$

where W denotes the vector processing unit's memory port width in bits. That means, that W is the number of bits, that can be concurrently loaded/stored into/from the vector registers from/to memory by the above described vector load/store instruction.

With $a >> 1$ it follows

$$N^V_{a,b,W} = 2a^2 + \left\lceil \frac{b}{W} \right\rceil \cdot (a + 3a^2). \tag{5.16}$$

The first term of Eq. (5.15) describes one memory access to read out the pivot element in Line 14 of Algorithm 13 and one memory access in Line 19 of Algorithm 13. The second term accounts for two memory accesses in Line 7 of Algorithm 13. The third, fourth and fifth terms describe the number of memory accesses to load or store the vector registers in the case of a port width of W bit. The third term accounts for one load of the vector in Line 4 of Algorithm 13. The forth term accounts for two vector load/store operations in Line 1 and Line 13 of Algorithm 13 and the fifth term for two load/store operations in Line 22 and Line 24 of Algorithm 13.

Example 5.3.
For a matrix size of $a = 1664, b = 8192$ and a variable port width W the Eq. (5.15) is simplified to be approximately

$$N^V_{1664,8192,W} = 5.54 \cdot 10^6 + \left\lceil \frac{8192}{W} \right\rceil \cdot 8.31 \cdot 10^6.$$

Memory port width W [bit]	8	32	64	128	256
$N^V_{1664,8192,W}$ $[\cdot 10^9]$	8.51	2.13	1.07	0.537	0.271
$1 - \frac{N^V_{1664,8192,W}}{N_{1664,8192}*}$ [in %]	31.9	83.0	91.4	95.7	97.8

* from Eq. (5.3), we get $N_{1664,8192} \approx 12.5 \cdot 10^9$ individual memory accesses

Table 5.3: Estimated number of memory accesses for the vector accelerated GEA and comparison to non-accelerated scalar GEA

By increasing the memory port width W, the number of memory accesses of the GEA can be reduced. Thus, the vector processing unit benefits from a high width data port, which is shown in Table 5.3. For a reasonably sized memory port, e.g. 256 bit, a reduction in memory accesses of up to 97.8% compared to the scalar solution introduced in Section 5.3 could be gained. But this is only a theoretical value for a rough design guidance, as the baseline is a non-optimized scalar source code. The C compiler can also significantly improve memory accesses on the scalar GEA implementation.

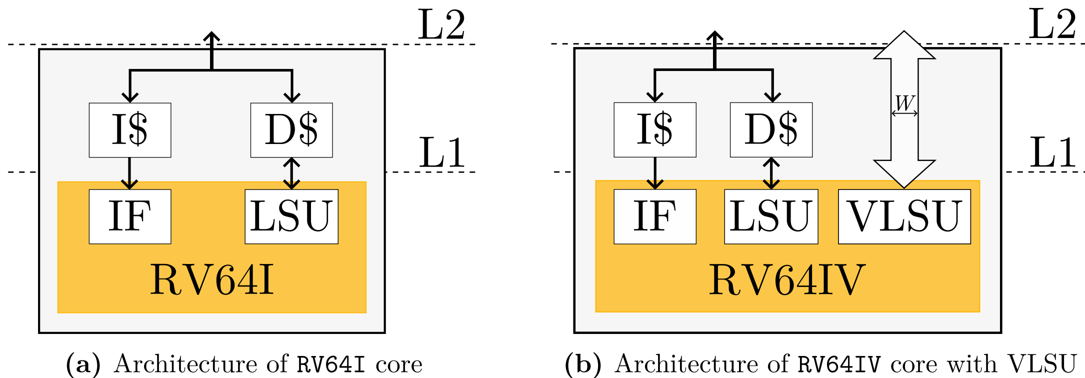
5.6 Experimental Evaluation

In this section the above described code is experimentally analysed regarding its runtime. As there was no hardware processor available integrating the RVV at the time this study was conducted, we investigated the source code on a simulator.

5.6.1 Simulation Environment

As simulator we use the ETISS [MDG+17], which is well suited for profiling and fast prototyping. It easily allows to add simulation support for RVV. ETISS uses a simple resource model and generates resource usage traces. This permits to compute the timing for different pipelines and memory interface configurations. The ISA models are specified in CoreDSL, which is a Domain-Specific Language (DSL). For the base ISA an open-source model [MIN20] is used. The RVV model is outsourced to an external library called "SoftVector". The machine encodings, micro-architectural states and exceptions are handled inside the DSL definition.

For our analysis we model a 64 bit processor with a four stage pipeline. As reference architecture the Register Transfer Level (RTL)[2] model of CVA6 (Ariane) from OpenHW Group [ZB19] is taken. The algorithm on a `RV64IM` architecture runs in a core with an internal L1 data cache. For the simulations with RVV, the `RV64IMV` architecture implements a separate vectorial pipeline with its own Vector Load/Store Unit (VLSU) and a vector arithmetic logic unit (VALU) for the vector instructions. For the vector load/store unit an own high-bandwidth memory interface is provided. Thus, these two architectures integrate different memory and cache systems, that are depicted in Fig. 5.1. Each resource in ETISS is assigned an individual execution time in cycles. This gives the possibility to get performance estimates on data sheet accuracy level.



(a) Architecture of `RV64I` core

(b) Architecture of `RV64IV` core with VLSU

Figure 5.1: Two processor architectures are used for the performance analysis of the GEA. The L1 cache consists of the instruction cache (I$) and the data cache (D$). An instruction fetch (IF) and a load/store unit (LSU) is considered for the scalar `RV64I` architecture. For the `RV64IV` architecture the `RV64I` is extended by RVV and a dedicated Vector Load/Store Unit (VLSU). Other core components, such as arithmetic units and instruction decoders, are not shown in these figures.

5.6.2 Performance Analysis

We cross-compiled the source code in Appendix A with GCC version 9.2.0 and the optimization level `-O3` for the best possible runtime optimization. The functionality

[2]For a short introduction into RTL see Section 6.3.3.

Architecture	$I[10^9]$	W[bit]	MP	$C[10^9]$	IPC	$t[s]$	G
RV64IM	4.99	64	10	7.76	0.643	15.5	-
			20	10.5	0.475	21.0	-
RV64IMV	0.064	64	10	1.27	0.0468	2.54	6.1
			20	1.36	0.0498	2.72	7.7
		128	10	0.737	0.0853	1.47	10.5
			20	0.830	0.0767	1.66	12.7
		256	10	0.472	0.132	0.944	16.4
			20	0.564	0.113	1.13	18.6

Table 5.4: GEA simulation results comparing a pure scalar (RV64IM) and vector accelerated model (RV64IMV). Table from [PGZM21].

of the vector accelerated GEA on `RV64IMV` was validated by comparing the simulation results with the simulation results of a scalar implementation of the GEA on `RV64IM` that does not integrate RVV.

The performance is substantially dependent on the memory accesses, because the modelled vector register length in RVV exceeds the base ISA registers by far. Thus, the data cache hits/misses need also be taken into account. For the `RV64IM` architecture, the cache misses are modelled via a penalty of additional 10 or 20 cycles per cache miss. For this, the L1 cache hit ratio was experimentally gathered via a RTL simulation of the above described CVA6 core [ZB19] and is 83% of all L1 cache accesses. For `RV64IMV`, a memory port warm-up penalty is introduced, which models a delay of 10 or 20 cycles before data through the memory port is streamed with a bandwidth of W. For each architecture, the two different cache miss penalties MP of 10 cycles and 20 cycles are explored.

We profiled with ETISS the runtime C and the number of instructions I of the vector accelerated GEA with RVV on `RV64IMV` and the non-accelerated GEA on `RV64IM`. These two values are gathered for a memory port width W of 64, 128 and 256 bits. The runtime C is given in clock cycles.

Table 5.4 presents the simulation results for the vector accelerated GEA with RVV on `RV64IMV` in comparison with the non-accelerated GEA on `RV64IM`. The instructions per cycle IPC are calculated by $IPC = \frac{I}{C}$. For the execution time t in seconds, a processor clock of 500 MHz is taken and thus, $t = \frac{C}{500\text{ MHz}}$. To calculate the speed-up gain G, all runtime clock cycles $C_{\text{RV64IMV},MP}$ of the same cache miss penalty on `RV64IMV` are compared to the runtime clock cycles $C_{\text{RV64IM},MP}$ of the matching cache miss penalty on `RV64IM`. Thus, $G = \frac{C_{\text{RV64IM},MP}}{C_{\text{RV64IMV},MP}}$.

For the non-accelerated GEA on `RV64IM` we can estimate a runtime of 7.76 billion cycles for an assumed MP of 10 cycles. That would be an execution time of 15.5 seconds on a 500 MHz processor. A higher MP of 20 cycles results in 10.5 billion

cycles. For different configurations of the VLSU memory port width W, we received performance gains of 6 up to 18 for very wide vector memory interfaces of 256 bits for the GEA using the RVV extension. The low IPC shows that the runtime on `RV64IMV` is strongly dependent on the memory port width W. However, these results clearly display the benefit of RVV.

Conclusion In this chapter, the implementations of Classic McEliece have been analysed and accelerated using the RISC-V Vector Extension (RVV). First, the implementations were profiled regarding their runtime on a x86-64 processor. The GEA was found as the most time-consuming algorithm in the cryptosystem. Thus, the GEA was selected for exploring the acceleration on a processor integrating the RISC-V ISA and its vector extension. Then, the GEA was investigated and the most promising code locations for vector integration were located. The RISC-V base ISA and RVV were explained in detail to understand the GEA source code modifications. Thereupon, the vector accelerated GEA source code was simulated and compared to the non-accelerated GEA. For this, a simulation environment was established using the Extendable Translating Instruction Set Simulator (ETISS). The SoftVector library on CoreDSL that connects RVV support to ETISS had to be developed. Two architectures with separate memory models and different sizes of the memory port width for the VLSU were investigated. Because the performance is substantially dependent on the memory accesses, also the data cache accesses had to be modelled. For this cache hits/misses were estimated and provided with a miss penality, which is reflected in the number of clock cycles needed for the execution of the algorithm. In our setup and on a RISC-V 64 bit Ariane core integrating RVV we received a 6 to 18 time higher runtime for memory port widths of 64 bit to 256 bits than for the GEA that does not take advantage of the vector extension. For further works different values of the maximum vector register length `VLEN` and thus the vector length `VL` can be examined. Different cores, memory and cache models can be explored. Additionally, also other algorithms used in the implementations of Classic McEliece should be investigated and further accelerated.

Chapter 6

Attacking Classic McEliece using Fault Injections

As already described in Section 3.5, Classic McEliece withstands decades of cryptanalysis and is designed to be robust to structural attacks and ISD attacks. Thus, it is exceedingly hard to attack the system from the mathematical perspective. But, it is interesting to look into attacks that target the hardware and the electronic device on which an implementation of Classic McEliece is executed, e.g. on an embedded system. A classical binary computer is built upon many transistors forming gates and storage elements connected via electrical wires. Two voltage levels are represented via a bit which can have values 0 or 1. If an attacker has access to the physical device, the cryptographic computations can be observed or disrupted in ways that compromise secret information. There exist different attack models that are interesting to look at for an embedded device. These can be divided into observing (passive) attacks and fault (active) attacks. Observing attacks passively monitor the behavior of the device or monitor some affect from its environment in some form e.g. observing voltage on electrical wires with an oscilloscope or measuring the time usage of a computation. These can be called side-channel attacks. In contrast, fault attacks actively manipulate the functionality of the device such that computations are intentionally disrupt. For example, the algorithms of a cryptosystem can be interfered by injecting light with a laser to flip bits or by shortly removing the power source (glitching). With highly focused laser beams and a good spatial and temporal precision, specific single and adjacent bits on a chip can be set and reset [SHS16]. Guo et al. [GJJ22] published a key-recovery side-channel attack on Classic McEliece. They use chosen ciphertexts and exploit a side-channel leakage in the additive Fast Fourier Transform (FFT) that evaluates the ELP during decoding. Cayrel et al. [CCD+21] present a message-recovery fault attack on Classic McEliece by attacking the syndrome computation that changes the syndrome from $\mathbb{F}_2$ to the integers $\mathbb{N}$. The resulting syndrome decoding problem in $\mathbb{N}$ can be easily solved by integer linear programming. In [CDCG22] they present a similar message-recovery attack using only side-channel information on power consumption of the chip. This attack also gathers information on the syndrome in $\mathbb{N}$ but is more tolerant to noise. Xagawa et al. [XIU+21] showed single-fault injection attacks for all NIST PQC Round 3 KEM candidates except for

Classic McEliece, as no simple key-recovery plaintext checking attacks for the underlying McEliece/Niederreiter PKE system are known. The presented single-fault injection attacks in [XIU+21] are executed by skipping instructions on a chip using glitching of the power supply. The skipping circumvents the IND-CCA2 security of the KEM and leads to execute chosen-chiphertext attacks on the vulnerable PKE.

In this chapter a key-recovery attack to the Classic McEliece Key Encapsulation Mechanism (KEM) considering fault injections is presented. The main idea is to use fault injections to retrieve the secret key of the cryptosystem from physical hardware via a chosen-chipertext attack. The fault injections target the Error Locator Polynomial (ELP) of the Goppa code and the validity checks of the decapsulation algorithm. The faulty outputs of decapsulation achieved by fault injections are used to set up a system of polynomial equations in the unknowns of the secret support elements of the Goppa code. The polynomial system of equations is solved over the extension field $\mathbb{F}_{2^m}$ and a suitable Goppa polynomial is determined via Eq. (3.5) to form an alternative secret key.

The attack is mathematically described and is based on Classic McEliece that is presented in Section 3.3 of Chapter 3. We simulated the complete attack on the implementations submitted to NIST Round 3 [BCL+20] with software in C-code and SageMath-code. We additionally investigated and simulated the fault injections on two open-source RISC-V processors.

In Section 6.1 the hardware fault model is defined and the mathematical background of the key-recovery attack is described. The details to solve the system of polynomial equations in the finite field $\mathbb{F}_{2^m}$ are given in Section 6.2. In Section 6.3 the implementation details and simulation results that validate the attack are presented. A feasibility study for two RISC-V processors using RTL simulations.

The idea of the work in this chapter has initially been presented at [PGMW22] and the content is published in [PGD+23]. Johannes Geier performed the RTL simulations presented in Section 6.3.3. For this, he developed a tool which has since been published in [GM23]. Julian Danner joined the project at a later stage and provided the algorithms in Section 6.2 and corresponding SageMath program to solve the system of polynomial equations and to calculate the alternative keys as well as contributed in the mathematical formulation of the principles of the described attack.

6.1 Mathematical Description of Key-Recovery Fault Injection Attack

We present an attack that finds an alternative secret key of the Classic McEliece KEM by adapting and combining the skipping attacks of Xagawa et al. [XIU+21] with the fault injection attack on the Niederreiter cryptosystem using Goppa codes by Danner and Kreuzer [DK20]. The attack targets the decapsulation function to find an alternative secret key. The alternative secret key can be used to gain access to sensitive information that is encrypted by a symmetric cipher whose symmetric key is generated by Classic McEliece.

6.1.1 Fault Model

It is considered that the secret key is stored in a Trusted Execution Environment (TEE) so that its memory location is well protected. Only the TEE itself has access to the secret key, i.e. the key cannot be physically accessed or retrieved by any other means.

Assumption 6.1.
The attacker has access to the input and output of the decapsulation function (Algorithm 6 in Chapter 3). She can freely choose the input of the decapsulation function (chosen ciphertext attack).

Assumption 6.2.
The attacker can inject faults on the physical device during decapsulation such that the transistor states are changed at specific positions and times. It is assumed that single and adjacent bits can be set or reset.

Such faults described in Assumption 6.2 are achievable, for example, by laser fault injections [SHS16].

On the computational level, the following is achieved by fault injections: The validity checks in Line 4 of Algorithm 7 and Line 5 of Algorithm 6 (see Chapter 3) is bypassed (VCB) and the ELP is corrupted either by setting or resetting one or more adjacent bits in a single coefficient (ELPB) or by setting a coefficient to zero (ELPZ).

To simplify the theoretic analysis of fault injections into the ELP, consider the following remark.

Remark 6.3.
We model the faults on a coefficient $a \in \mathbb{F}_{2^m}$ of the ELP as an addition in the field $\mathbb{F}_{2^m}$, i.e. write the faulty coefficient $\tilde{a} \in \mathbb{F}_{2^m}$ as $\tilde{a} = a + \xi$ for some appropriately chosen $\xi \in \mathbb{F}_{2^m}$. Note that for our attack we do not need to know the fault value ξ.

6.1.2 Implementation Specific Behaviour of Decoding

The implementations submitted to NIST [BCL+20] contain a reference implementation, as well as several hardware accelerated implementations for x86/AMD64 processors. These implementations are described in Section 5.1.1 and Section 5.1.2. The ELP $\sigma_e(x) \in \mathbb{F}_{2^m}[x]$ (see Eq. (2.23)) is represented differently in the reference and hardware accelerated code, but all implementations share the same properties that result from storing the ELP in memory. These can be mathematically modeled as follows.

Remark 6.4 (Implementation Details of ELP).
The ELP $\sigma_e(x) \in \mathbb{F}_{2^m}[x]$ of degree $\deg(\sigma_e) \leqslant t$ is stored as the polynomial $\sigma_e(x) \cdot x^{t-\deg(\sigma_e)}$ of degree t. All coefficients of x^j with $j < t - \deg(\sigma_e)$ of this polynomial are zero.
Using this alternate form of the ELP does not affect error correction as long as no α_i

is zero, or $\mathrm{wt}(\mathbf{e}) = \deg(\sigma_e(x)) = t$*. However, if there is* $i \in \{1, \dots, n\}$ *with* $\alpha_i = 0$ *and* $\mathrm{wt}(\mathbf{e}) < t$*, then the output* $\mathbf{e}' \in \mathbb{F}_2^n$ *of Line 2 in Algorithm 7 is changed, resulting in* $\mathrm{supp}(\mathbf{e}') = \mathrm{supp}(\mathbf{e}) \cup \{i\}$ *and* $\mathrm{wt}(\mathbf{e}') \leqslant \mathrm{wt}(\mathbf{e}) + 1$*. In particular,* $\mathbf{e} \neq \mathbf{e}'$ *only if there is an* $i \in \{1, \dots, n\}$ *with* $\alpha_i = 0$ *and* $\mathbf{e}_i = 0$*.*

This allows the attacker to quickly find the index $i \in \{1, \dots, n\}$ for the zero element $\alpha_i = 0$, if it is contained in $\mathcal{L}$ (details in Remark 6.7).

6.1.3 Overview of Attack

The attack targets the decapsulation function and can find an alternative secret key. The main idea is to generate chosen ciphertexts and induce faults on the decapsulation procedure in order to retrieve the secret support $\mathcal{L}$ of the Goppa code. If the support is known, the Goppa polynomial $g(x)$ of the secret key can be calculated using the public key $\mathbf{T}$ (see also Section 3.5.1).

Theorem 6.5 ([BBD09, p. 125]).
If one part of the secret key Γ *(the support set* $\mathcal{L}$ *or the Goppa polynomial* $g(x)$*) is known, the other part of the secret key* Γ *can be calculated using the public key* $\mathbf{T}$*.*

The attack consists of two fault injection steps on the decapsulation. The decryption procedure of Classic McEliece in Algorithm 6 and Algorithm 7 is illustrated in a flow chart in Figure 6.1a showing the algorithm in normal operation. Fig. 6.1b illustrates a faulty operation of decapsulation and indicates the fault positions necessary for the attack. The first step of the attack is to inject faults on the decoding procedure on the ELP coefficients so that it leaks information about the secret key. The second step is to produce faults that bypass the validity checks (VCB) ensuring the faulty decoding result is not rejected. With a wise choice of the chosen-ciphertexts the information about the secret key contained in the hashed output can be retrieved. That is demonstrated by our simulation in Section 6.3, where it is demonstrated that under given circumstances the information about the secret key contained in the hashed output can be retrieved.
There are two kinds of faults in the ELP coefficients (in Section 6.1.6):

- ELPB: single and adjacent bits of coefficients of ELP are corrupted.
- ELPZ: coefficients of ELP are corrupted to zero.

Both lead to successful key recovery, but the case of zeroing the coefficients in ELPZ is more efficient, which will be discussed in the next sections.

With access to the faulty output $\tilde{\mathbf{e}}'$ of the decoding step via the second fault of VCB in Section 6.1.4, a polynomial equation with unknowns of the secret support $\mathcal{L}$ can be obtained. The goal is to gather enough equations to solve the resulting polynomial system of equations. A solution of that (non-linear) system of polynomial equations via Gröbner basis and Buchberger's algorithm (see also Appendix B) eventually leads to an alternative support $\tilde{\mathcal{L}}$ for which there is an irreducible polynomial $\tilde{g}(x) \in \mathbb{F}_{2^m}[x]$

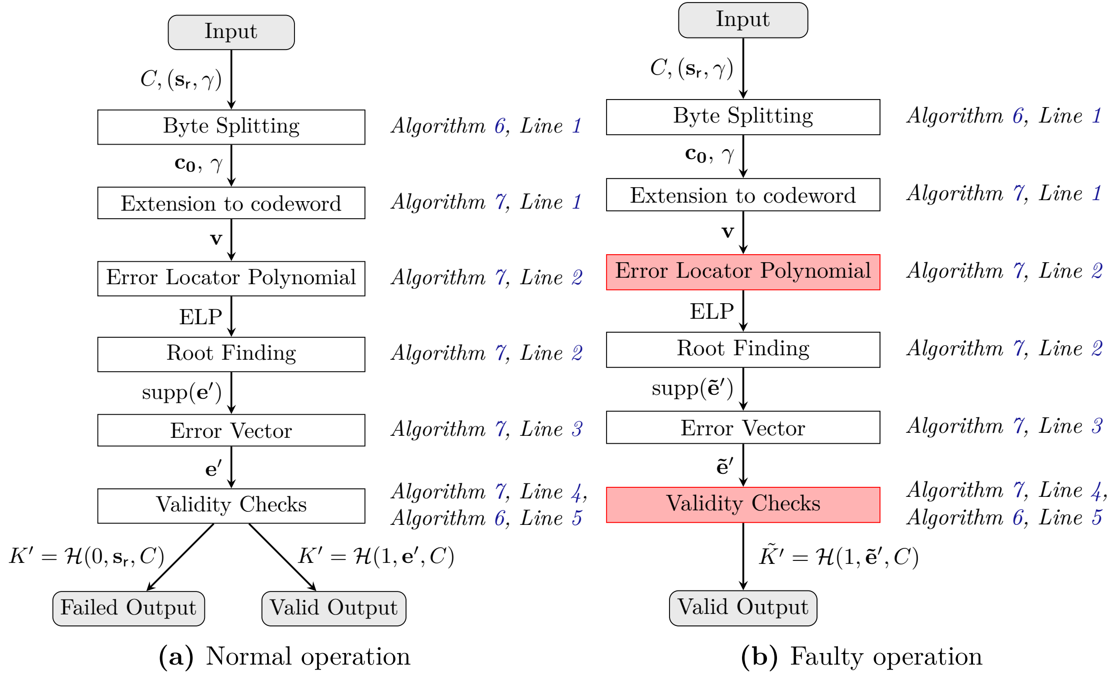


Figure 6.1: Flowchart showing the decapsulation and decoding steps indicating the differences between normal and faulty operation. Fault injections target the steps marked in red.

of degree t with $\Gamma(\mathcal{L}, g) = \Gamma(\tilde{\mathcal{L}}, \tilde{g})$. This allows efficient correction of up to t errors in the code $\Gamma(\mathcal{L}, g)$. Hence, for every $\mathbf{s}_r \in \mathbb{F}_2^n$ the tuple $(\mathbf{s}_r, (\tilde{g}, \tilde{\mathcal{L}}))$ can be used as an alternative secret key with Algorithm 6. If the zero element $\mathbf{0} \in \mathbb{F}_{2^m}$ is contained in the support $\mathcal{L}$, then its index can be easily found by 6.1.5.

6.1.4 Fault Injection on the Validity Checks (VCB)

If a faulty ELP is accomplished with the first fault injection and a faulty error vector $\tilde{\mathbf{e}}$ is computed, then the validity checks in Algorithm 7 Line 4 and Algorithm 6 Line 5 would result in a failure. The checks confirm whether the decoding function provides a valid output and compares the corresponding hashes. In general the faulty error vector $\tilde{\mathbf{e}}'$ is not equal to $\mathbf{e}$ and the decapsulation procedure would return a predefined session key with $\mathbf{s}_r$ (*Failed Output* in Figure 6.1a). Thus, this needs to be prevented. The second fault circumvents these validity checks $C_1' = C_1$ and $\mathbf{H}_{\mathrm{sys}}\tilde{\mathbf{e}}'^{\top} = \mathbf{H}_{\mathrm{sys}}\mathbf{e}^{\top}$ in the decapsulation and decoding procedure. Both validity checks can be attacked by one fault injection.

A faulty session key $\tilde{K}' = \mathcal{H}(1, \tilde{\mathbf{e}}', C)$ is a hash of the input ciphertext $C = (\mathbf{c}_0, \mathcal{H}(2, \mathbf{e}))$ and the output $\tilde{\mathbf{e}}' \in \mathbb{F}_2^n$ of the decode algorithm. According to our fault model the attacker has full control over C. It is feasible to extract $\tilde{\mathbf{e}}'$ from $\tilde{K}'$ by exhaustive search if the weight of $\tilde{\mathbf{e}}'$ is small enough.

Remark 6.6 (De-hash Session Key).

(a) If C and hash $\tilde{K}' = \mathcal{H}(1, \tilde{\mathbf{e}}', C)$ are known for some $\tilde{\mathbf{e}}' \in \mathbb{F}_2^n$ with $\mathrm{wt}(\tilde{\mathbf{e}}') \leq 2$, then

one can find $\tilde{\mathbf{e}}'$ *with less than* $\binom{n}{2}+\binom{n}{1}+\binom{n}{0}$ *hash computations and comparisons via exhaustive search.*

(b) *The statement in (a) is also true if* $\mathrm{wt}(\tilde{\mathbf{e}}') \leq 3$ *and one index* $i \in \mathrm{supp}(\tilde{\mathbf{e}}')$ *is known.*

(c) *For the parameters (Table 3.3), we have* $n \leq 2^{13}$*, this means that less than* $2^{25}+2^{12}+1$ *hash computations and comparisons are required to find the output of the decoding algorithm* $\tilde{\mathbf{e}}'$ *from a faulty session key* $\tilde{K}'$*, given that* $\mathrm{supp}(\tilde{\mathbf{e}}')$ *contains at most two unknown indices.*

Before the corruption of the coefficients of the ELP which lead to polynomial equations in the unknown support $\mathcal{L}$ of the Goppa code is discussed, it is shown that one can easily check if zero is one of the support elements and if so, find its index.

6.1.5 Locating the Zero Element in the Support

As already mentioned, the attacker can freely choose the input of the decapsulation algorithm and can read faulty outputs with VCB if the input ciphertext comes from a small weight $\mathbf{e}$ (see Section 6.1.4). Thus, it is also possible to choose an input ciphertext that is generated from the all-zero vector $\mathbf{e} = \mathbf{0} \in \mathbb{F}_2$ of weight $\mathrm{wt}(\mathbf{e}) = 0$. This allows to find out whether the zero element is contained in the support or not. If the zero element, for which $\alpha_j = 0$, is part of the support its index $j \in \{1, \ldots, n\}$ can be found without a fault injection on the ELP.

Remark 6.7 (Finding of Zero Element).
Let $\mathbf{e} = \mathbf{0} \in \mathbb{F}_2^n$*, and set* $C = (\mathbf{c_0} = \mathbf{e}\mathbf{H}_{\text{sys}}{}^{\top}, \mathcal{H}(2, \mathbf{e}))$ *as input for the decapsulation algorithm in Algorithm 6 such that* $\mathbf{c_0}$ *is the input for the decoding algorithm in Algorithm 7. If there is a* $j \in \{1, \ldots, n\}$ *with* $\alpha_j = 0$*, then the decoding algorithm evaluates the polynomial* x^t *and outputs* $\mathbf{e}' \in \mathbb{F}_2^n$ *where* $\mathrm{supp}(\mathbf{e}') = \{j\}$ *by Remark 6.4. Otherwise,* $\mathbf{e}' = \mathbf{e} = \mathbf{0}$ *holds.*
Since $\mathrm{wt}(\mathbf{e}') \leq 1$ *a VCB fault can be applied and* $\mathbf{e}'$ *from the hash output of the decapsulation function retrieved (see Remark 6.6). From* $\mathbf{e}'$ *it can be determined whether there exists a* $j \in \{1, \ldots, n\}$ *with* $\alpha_j = 0$ *and find the index* j*.*

Remark 6.8 (Probability of Existence of Zero Element).
The probability that the zero element exists in $\mathcal{L}$ *is* $\frac{n}{2^m}$*. Thus, with smaller distance between* n *and* 2^m *the probability increases.*

For the next sections, it is presumed that the attacker has knowledge of the index j for $j \in \{1, \ldots, n\}$ with $\alpha_j = 0$, if it exists. The zero element in the support can be gathered with a single execution of the decapsulation algorithm with chosen ciphertext input generated from the all-zero vector and one fault to skip the validity checks VCB.

6.1.6 Fault Injection on the ELP Coefficients

To receive information about the secret key, fault injections into certain coefficients of the Error Locator Polynomial (ELP) during the decoding process are necessary. The inputs are chosen-ciphertexts generated from chosen words $\mathbf{e} \in \mathbb{F}_2^n$ of Hamming weight 2. The ELP is defined as (see also Eq. (2.23) and Eq. (2.24))

$$\sigma_e(x) = \prod_{i \in \varepsilon} (x - \alpha_i) \quad \in \mathbb{F}_{2^m}[x] \tag{6.1}$$

where $\varepsilon = \{i \mid e_i \neq 0, i = 1, \ldots, n-1\}$ is the set of error locations. The goal is to receive a faulty output $\tilde{\mathbf{e}}' \in \mathbb{F}_2^n$ of the decoding step whose non-zero positions are the indices of the code locators that are roots of the faulty ELP. The roots are only searched among the code locators. So, the root finding leaks information about the evaluated code locators of the secret key.

The fault injections on the ELP are mainly based on the ideas of the fault attack presented in [DK20], but a handful of adjustments had to be made to accommodate the different fault model and the peculiarities of the implementation. Also the solving process was refined to decrease the number of required fault injections.

A faulty ELP is denoted as $\tilde{\sigma}_e$ with $\tilde{\mathbf{e}}'$ the faulty error vector calculated from a faulty ELP such that

$$\tilde{e}'_i = \begin{cases} 1 & \text{if } \widetilde{\sigma_e}(\alpha_i) = 0 \\ 0 & \text{otherwise} \end{cases} \tag{6.2}$$

where $i \in \{1, \ldots, n\}$. Based on the non-zero positions of the output of the decoding function, it can be read out which support element is a zero of the (faulty) ELP. If $\tilde{\mathbf{e}}' = \mathbf{0}$, then there could no root be found among the code locators in $\mathcal{L}$.

For both cases ELPB and ELPZ the syndromes corresponding to vectors $\mathbf{e} \in \mathbb{F}_2^n$ of weight 2 are chosen as input to the decapsulation procedure. Then, an ELP has degree 2 and the form $\sigma_e(x) = (x - \alpha_{i_1})(x - \alpha_{i_2}) \in \mathbb{F}_{2^m}[x]$ for $\operatorname{supp}(\mathbf{e}) = \{i_1, i_2\}$ and chosen $i_1, i_2 \in \{1, \ldots, n\}$ with $i_1 \neq i_2$.

Definition 6.9 (Fault Injection on ELP).
Let $\mathbf{e} \in \mathbb{F}_2^n$ *with* $\operatorname{wt}(\mathbf{e}) = 2$ *and* $\operatorname{supp}(\mathbf{e}) = \{i_1, i_2\} \subseteq \{1, \ldots, n\}$ *for* $i_1 \neq i_2$. *Let* $d \in \{0, 1\}$ *be the degree location for the induced fault and let* $\tilde{\mathbf{e}}' \in \mathbb{F}_2^n$ *be the faulty error vector from output of Algorithm 7 where a fault was injected during computation such that the d-th coefficient of* $\sigma_e(x)$ *is corrupted. Then, the ELP is replaced by a faulty ELP*

$$\tilde{\sigma}_e(x) = \xi x^d + \sigma_e(x) \tag{6.3}$$

with $\xi \in \mathbb{F}_{2^m}$. *The result of a fault injection on the ELP is a pair* $(\mathbf{e}, \tilde{\mathbf{e}}')$ *.*

Our fault model allows to generate arbitrary many such fault injections. Also note that the value of ξ is unknown. It is introduced in order to simplify the mathematical descriptions. In reality the implementation behaves as described in Remark 6.10. We use two types of fault injections on the ELP:

(a) Constant fault injection ($d = 0$): a fault is injected on the coefficient of the constant term of the ELP.

(b) Linear fault injection ($d = 1$): a fault is injected on the coefficient of the linear term of the ELP.

In order to calculate the unknown support set $\mathcal{L}$, constant fault injections are insufficient (see Proposition 6.18). We need additional fault injections on the ELP which we achieve by compromising the linear coefficient of the ELP with $d = 1$.

For an $\mathbf{e}$ of weight 2 and thus for an ELP of degree 2, the implementation specific output using Remark 6.4 and Remark 6.7 is summarized in the following remark, if a fault injection is given to the linear or constant coefficient. Recall that the implementation specific output of the decode algorithm is constructed not from the zeros of $\tilde{\sigma}_e(x)$ but from the zeros of $x^{t-2}\tilde{\sigma}_e(x)$.

Remark 6.10 (Implementation Specific Output for Fault on Quadratic ELP).
Let $\mathbf{e} \in \mathbb{F}_2^n$ *with* $\mathrm{supp}(\mathbf{e}) = \{i_1, i_2\}$ *and* $i_1 \neq i_2$. *Assume that a fault* $\xi \in \mathbb{F}_{2^m}$ *is injected into the* d*-th coefficient of* $\sigma_e(x)$ *with* $d \in \{0, 1\}$. *Let* $\tilde{\mathbf{e}}' \in \mathbb{F}_2^n$ *be the output of the decoding algorithm when the polynomial for the root finding is given by* $x^{t-2}\tilde{\sigma}_e(x)$ *(Remark 6.4) where* $\tilde{\sigma}_e(x) = \xi x^d + \sigma_e(x)$.

(a) If $\alpha_j \neq 0$ *for all* $j \in \{1, \ldots, n\}$, *then* $\mathrm{wt}(\tilde{\mathbf{e}}') \leq 2$, *and*

$$\mathrm{supp}(\tilde{\mathbf{e}}') = \{i \in \{1, \ldots, n\} \mid \alpha_i \textit{ is a zero of } \tilde{\sigma}_e(x)\}. \tag{6.4}$$

(b) If there is $j \in \{1, \ldots, n\}$ *with* $\alpha_j = 0$, *then* $\mathrm{wt}(\tilde{\mathbf{e}}') \leq 3$ *and* $j \in \mathrm{supp}(\tilde{\mathbf{e}}')$ *is known. Moreover, we have*

$$\mathrm{supp}(\tilde{\mathbf{e}}') = \{i \in \{1, \ldots, n\} \mid \alpha_i \textit{ is a zero of } \tilde{\sigma}_e(x)\} \cup \{j\}. \tag{6.5}$$

Using Remark 6.7 these two cases can be distinguished.

Not all fault injections lead to a polynomial equation, but only those where the faulty ELP has two zeros among the support $\mathcal{L}$. Thus, it can be seen from the weight of the faulty output $\tilde{e}'$, if a fault injection was successful and leaked information about the secret key.

Definition 6.11 (Successful Fault Injection).
A result $(\mathbf{e}, \tilde{\mathbf{e}}')$ *of a fault injection is called successful, if*

(1) for all $j \in \{1, \ldots, n\}$ *holds* $\alpha_j \neq 0$ *and* $\mathrm{wt}(\tilde{\mathbf{e}}') = 2$, *or*

(2) there is a $j \in \{1, \ldots, n\}$ *with* $\alpha_j = 0$ *and* $\mathrm{wt}(\tilde{\mathbf{e}}') = 3$.

For every successful result the set $\{i \in \{1, \ldots, n\} \mid \alpha_i \text{ is a zero of } \tilde{\sigma}_e(x)\}$ is deduced using Remark 6.10 and Definition 6.11 such that it contains exactly two elements in order to set up the polynomial equations in the following two cases ELPB and ELPZ.

Remark 6.12 (Probability Successful Fault Injection).
The probability to have a successful fault injection increases with the ratio $\frac{n}{2^m}$. This follows simply from the fact that the number of support elements increases with n and by that also the number of possible roots for the faulty ELP $\tilde{\sigma}_e(x)$ increases.

6.1.6.1 Corrupting bits of coefficients (ELPB)

In the case of ELPB, the fault injections are positioned such that single or adjacent bits in the linear or constant coefficient of an ELP are set or reset. This means, that the ELP is replaced by $\tilde{\sigma}_e(x) = \xi x^d + \sigma_e(x)$ for $d \in \{0, 1\}$ and some (unknown) $\xi \in \mathbb{F}_{2^m}$ (see also Remark 6.3).

Proposition 6.13 (Fault Equations ELPB).
Let $(\mathbf{e}, \tilde{\mathbf{e}}')$ be the result of a successful fault injection with $\text{supp}(\mathbf{e}) = \{i_1, i_2\}$ and $\{i \in \{1, \ldots, n\} \mid \alpha_i$ is a zero of $\tilde{\sigma}_e(x)\} = \{j_1, j_2\}$. Then, an equation can be found in the following forms.

(a) *Constant injection ($d = 0$):*
If $(\mathbf{e}, \tilde{\mathbf{e}}')$ is a successful result of a constant injection, then

$$\alpha_{i_1} + \alpha_{i_2} = \alpha_{j_1} + \alpha_{j_2} \tag{6.6}$$

and $(\alpha_1, \ldots, \alpha_n)$ is a zero of the linear polynomial $x_{i_1} + x_{i_2} + x_{j_1} + x_{j_2} \in \mathbb{F}_{2^m}[x_1, \ldots, x_n]$.

(b) *Linear injection ($d = 1$):*
If $(\mathbf{e}, \tilde{\mathbf{e}}')$ is a successful result of a linear injection, then

$$\alpha_{i_1}\alpha_{i_2} = \alpha_{j_1}\alpha_{j_2} \tag{6.7}$$

and $(\alpha_1, \ldots, \alpha_n)$ is a zero of the quadratic polynomial $x_{i_1}x_{i_2} + x_{j_1}x_{j_2} \in \mathbb{F}_{2^m}[x_1, \ldots, x_n]$.

Proof. The proof can be shown by comparing the coefficients of the resulting polynomials. The ELP of degree 2 of $\text{supp}(\mathbf{e}) = \{i_1, i_2\}$ is denoted by $\sigma_e(x) = (x - \alpha_{i_1})(x - \alpha_{i_2})$. The faulty ELP is denoted by $\tilde{\sigma}_e(x) = \xi x^d + \sigma_e(x)$ for $d \in \{0, 1\}$ and $\xi \in \mathbb{F}_{2^m}$. By Remark 6.10 and Definition 6.11, the roots α_{j_1} and α_{j_2} of $\tilde{\sigma}_e(x)$ are known and $\tilde{\sigma}_e(\alpha_{j_1}) = \tilde{\sigma}_e(\alpha_{j_2}) = 0$ applies. Then, $\tilde{\sigma}_e(x)$ can also be written as $\tilde{\sigma}_e(x) = (x - \alpha_{j_1})(x - \alpha_{j_2}) = x^2 + (\alpha_{j_1} + \alpha_{j_2})x + \alpha_{j_1}\alpha_{j_2}$. For both statements (a) and (b) the coefficients in $\xi x^d + \sigma_e(x) = \tilde{\sigma}_e(x)$ are compared:

$$\xi x^d + (x - \alpha_{i_1})(x - \alpha_{i_2}) = (x - \alpha_{j_1})(x - \alpha_{j_2}).$$

For (a) with $d = 0$ this result in $x^2 + (\alpha_{i_1} + \alpha_{i_2})x + (\alpha_{i_1}\alpha_{i_2} + \xi) = x^2 + (\alpha_{j_1} + \alpha_{j_2})x + \alpha_{j_1}\alpha_{j_2}$.
Comparing the linear coefficients yields $\alpha_{i_1} + \alpha_{i_2} = \alpha_{j_1} + \alpha_{j_2}$.
For (b) with $d = 1$ this result in $x^2 + (\alpha_{i_1} + \alpha_{i_2} + \xi)x + \alpha_{i_1}\alpha_{i_2} = x^2 + (\alpha_{j_1} + \alpha_{j_2})x + \alpha_{j_1}\alpha_{j_2}$.
So, $\alpha_{i_1}\alpha_{i_2} = \alpha_{j_1}\alpha_{j_2}$ follows from the constant coefficients. □

Note that $\alpha_{j_1}, \alpha_{j_2}$ are the roots of the faulty error locator polynomial $\widetilde{\sigma_e}(x) = 0$ and $\alpha_{i_1}, \alpha_{i_2}$ are the roots of the original error locator polynomial $\sigma_e(x)$, but all are elements of the support set $\mathcal{L}$ ($\alpha_{j_1}, \alpha_{j_2}, \alpha_{i_1}, \alpha_{i_2} \in \mathcal{L}$).

6.1.6.2 Zeroing Coefficients (ELPz)

From observations of the RTL simulation (see Section 6.3.3), we found out that instead of targeting individual bits of the ELP coefficients stored in GPRs, one may also target the instructions that operate on them. For example, by skipping the instruction that stores the ELP coefficient to memory, the resulting coefficient will be equal to zero. This is the case as the algorithm sets the ELP vector to zero before calculating its coefficients. Such fault injections also fit well with Definition 6.9, where the fault value $\xi \in \mathbb{F}_{2^m}$ has the same value as the targeted coefficient of the ELP such that the coefficient cancels out. Coefficient-zeroing fault injections provide polynomial equations as follows.

Proposition 6.14 (Fault Equations ELPz).
Let $(\mathbf{e}, \tilde{\mathbf{e}}')$ *be the result of a successful fault injection with* $\text{supp}(\mathbf{e}) = \{i_1, i_2\}$ *on the* d*-th coefficient of* $\sigma_e(x)$ *such that the* d*-coefficient of* $\tilde{\sigma}_e(x)$ *is set to zero. Let* $j \in \text{supp}(\tilde{\mathbf{e}}')$ *with* $\alpha_j \neq 0$*. Then, an equation can be found in the following forms.*

(a) Constant injection $(d = 0)$*:*
If $(\mathbf{e}, \tilde{\mathbf{e}}')$ *is a successful result of a constant injection, then*

$$\alpha_{i_1} + \alpha_{i_2} = \alpha_j \tag{6.8}$$

and $(\alpha_1, \ldots, \alpha_n)$ *is a zero of the linear polynomial* $x_{i_1} + x_{i_2} + x_j \in \mathbb{F}_{2^m}[x_1, \ldots, x_n]$*.*

(b) Linear injection $(d = 1)$*:*
If $(\mathbf{e}, \tilde{\mathbf{e}}')$ *is a successful result of a linear injection, then*

$$\alpha_{i_1}\alpha_{i_2} = \alpha_j^2 \tag{6.9}$$

and $(\alpha_1, \ldots, \alpha_n)$ *is a zero of the quadratic polynomial* $x_{i_1}x_{i_2} + x_j^2 \in \mathbb{F}_{2^m}[x_1, \ldots, x_n]$*.*

Proof. As in the proof above, the ELP of degree 2 of $\text{supp}(\mathbf{e}) = \{i_1, i_2\}$ is denoted by $\sigma_e(x) = (x - \alpha_{i_1})(x - \alpha_{i_2}) = x^2 + (\alpha_{i_1} + \alpha_{i_2})x + \alpha_{i_1}\alpha_{i_2}$.
For (a) with $d = 0$ it applies $\tilde{\sigma}_e(x) = x^2 + (\alpha_{i_1} + \alpha_{i_2})x$. By Remark 6.4 the implementation constructs $\tilde{\mathbf{e}}'$ from the zeros of $x^{t-2}\tilde{\sigma}_e(x) = x^{t-1}(x + \alpha_{i_1} + \alpha_{i_2})$ which has only one non-zero root $\alpha_{i_1} + \alpha_{i_2}$. Then, $j \in \text{supp}(\tilde{\mathbf{e}}')$ with $\alpha_j \neq 0$ implies that $\alpha_j = \alpha_{i_1} + \alpha_{i_2}$.
For (b) with $d = 1$ it applies $\tilde{\sigma}_e(x) = x^2 + \alpha_{i_1}\alpha_{i_2}$. For $\alpha_j \neq 0$ a zero of $\tilde{\sigma}_e(x)$ and of $x^{t-2}\tilde{\sigma}_e(x)$ the polynomial can be written as $\tilde{\sigma}_e(x) = x + \alpha_j$ and $\tilde{\sigma}_e(\alpha_j) = 0 = \alpha_j^2 + \alpha_{i_1}\alpha_{i_2}$. Thus, it follows $\alpha_j^2 = \alpha_{i_1}\alpha_{i_2}$. □

Recall that the attacker knows if there is $j \in \text{supp}(\tilde{\mathbf{e}}')$ with $\alpha_j \neq 0$ due to Remark 6.7. Hence, the above proposition can be applied directly.

Remark 6.15 (Probability for Successful Fault Injection for ELPz).
If the support elements $\alpha_1, \dots, \alpha_n$ *and* $\mathbf{e} \in \mathbb{F}_2^n$ *with* $\mathrm{wt}(\mathbf{e}) = 2$ *are chosen uniformly at random, then the probability that there exists an* α_j *with* $\alpha_j = \alpha_{i_1} + \alpha_{i_2}$ *or* $\alpha_j^2 = \alpha_{i_1}\alpha_{i_2}$ *for* $\mathrm{supp}(\mathbf{e}) = \{i_1, i_2\}$ *is* $\frac{n}{2^m}$*. This means that the success rate for obtaining a polynomial equation for a zeroing fault injection (ELPz) is about* $\frac{n}{2^m}$*.*

This probability is significantly greater than the success rate of the injections that directly target single or adjacent bits of the coefficients (ELPb), especially if $n \ll 2^m$. This observation can be confirmed with our simulations.

6.2 Computation of Alternative Secret Key

After the equations with indeterminates in the unknown support elements were gathered, the obtained polynomial system of equations needs to be solved in order to assign values to the elements. The drawback of this kind of method is, that it is not fail-safe. If the equations contain any wrong information, the secret key cannot be reliably reconstructed and the whole key-recovery attack has to start over again. To verify if the recovered secret key is indeed a valid secret key of the cryptosystem, we implemented a check that compares the public key of the original and the public key calculated from the recovered secret key. Two scenarios can be distinguished, depending on whether the obtained system of polynomial equations contain equations in all support elements or only a subset of them.

6.2.1 System of polynomial equations contains all unknowns of the support set

Fault injections on ELP together with VCB are repeated many times for different chosen ciphertext inputs to collect polynomial equations using Proposition 6.13 or Proposition 6.14. These polynomial equations can be collected and written in a fault equation system set $\mathcal{F} \subseteq \mathbb{F}_{2^m}[x_1, \dots, x_n]$ that has the code locators of the code $(\alpha_1, \dots, \alpha_n)$ as a common zero. The polynomials in $\mathcal{F}$ are either linear or quadratic due to the special choose of the input ciphertexts as described in Proposition 6.13 and Proposition 6.14. Both linear and quadratic equations are needed in order to solve the polynomial system of equations, as shown in Proposition 6.18. Denote the set of zeros of $\mathcal{F} \subseteq \mathbb{F}_{2^m}[x_1, \dots, x_n]$ by

$$\mathcal{Z}(\mathcal{F}) = \{\mathbf{a} \in \mathbb{F}_{2^m}^n \mid f(\mathbf{a}) = 0 \text{ for all } f(x_1, \dots, x_n) \in \mathcal{F}\}. \tag{6.10}$$

The goal is to find a support candidate set $\mathcal{S}_\mathcal{F} \subseteq \mathcal{Z}(\mathcal{F})$ that is a subset of the set of the common zeros of $\mathcal{F}$. The support candidate set contains a support candidate $(\tilde{\alpha_1}, \dots, \tilde{\alpha_n}) \in \mathcal{S}_\mathcal{F}$ for which an irreducible polynomial $\tilde{g}(x)$ of degree t exists with $\Gamma(\mathcal{L}, g) = \Gamma(\tilde{\mathcal{L}}, \tilde{g})$ and $\tilde{\mathcal{L}} = \{\tilde{\alpha_1}, \dots, \tilde{\alpha_n}\}$. Such a support candidate set is found by solving the system of polynomial equations using Gröbner basis and Buchberger's algorithm (see also Appendix B). The solving process of the fault attack in [DK20,

Section 6] is used and summarized as follows.

Proposition 6.16 (Solving Fault Equations).
Let $\mathcal{F} \subseteq \mathbb{F}_{2^m}[x_1, \ldots, x_n]$ be a fault equation system. A support candidate set $\mathcal{S}_{\mathcal{F}}$ for $\mathcal{F}$, is computed by the following steps:

(1) Reduce the linear polynomials in $\mathcal{F}$ to h remaining indeterminates by Gaussian elimination.

(2) Substitute the leading terms in the quadratic polynomials with the remaining indeterminates. This set of reduced quadratic equations is denoted as $\mathcal{F}_{red} \in \mathbb{F}_{2^m}[x_{i_1}, \ldots, x_{i_h}]$.

(3) Fix one of the remaining indeterminates to 1 (the equation $x_i - 1$ is added to $\mathcal{F}_{red}$ for some $i \in \{i_1, \ldots, i_h\}$).

(4) Find the set of zeros $\mathcal{Z}(\mathcal{F}_{red}) \subseteq \mathbb{F}_{2^m}^h$ of $\mathcal{F}_{red}$ via Gröbner basis and Buchberger's algorithm (see also Appendix B).

(5) Extend the zeros in $\mathcal{Z}(\mathcal{F}_{red})$ to elements of $\mathcal{Z}(\mathcal{F} \cup \{x_i - 1\}) \subseteq \mathbb{F}_{2^m}^n$ using the linear polynomials. Then, the support candidate set $\mathcal{S}_{\mathcal{F}}$ is

$$\mathcal{S}_{\mathcal{F}} = \{(\tilde{\alpha_1}, \ldots, \tilde{\alpha_n}) \in \mathcal{Z}(\mathcal{F} \cup \{x_i - 1\}) \mid \tilde{\alpha}_{j_1} \neq \tilde{\alpha}_{j_2} \text{ for } j_1 \neq j_2 \text{ and } j_1, j_2 = 1, \ldots, n\}.$$

For every support candidate in $\mathcal{S}_{\mathcal{F}}$ it is checked if a corresponding irreducible Goppa polynomial $\tilde{g}(x)$ can be calculated to generate the Goppa code $\Gamma(\mathcal{L}, g)$. This is done via Eq. (3.5) and the generated code is compared to $\Gamma(\mathcal{L}, g)$ using the public key $\mathbf{T}$. In this work a corresponding Goppa polynomial is computed by the following steps.

Proposition 6.17 (Finding Goppa Polynomial).
Let $(\tilde{\alpha}_1, \ldots, \tilde{\alpha}_n) \in \mathbb{F}_{2^m}^n$ be a set of code locators with $\tilde{\alpha}_i \neq \tilde{\alpha}_j$ for $i \neq j$. For the number of codewords $s \geq 1$, do the following sequence of instructions to calculate a suitable Goppa polynomial.

(1) Choose codewords $\mathbf{c}_1, \ldots, \mathbf{c}_s \in \Gamma(\mathcal{L}, g)$ (obtained using the public key $\mathbf{T}$) and set $\tilde{g}(x) = 0$.

(2) Set $f_j(x) = \sum_{i \in \text{supp}(\mathbf{c_j})} \prod_{k \in \text{supp}(\mathbf{c_j}) \setminus \{i\}} (x - \alpha_k)$ for $j \in \{1, \ldots, s\}$ and compute $h(x) = \gcd(f_1(x), \ldots, f_s(x))$ using the Euclidean algorithm.

(3) Factorize $h(x)$ and collect all irreducible factors of degree t in a set $\mathcal{G}$.

(4) For every $\hat{g}(x) \in \mathcal{G}$, check if $\Gamma(\tilde{\mathcal{L}}, \hat{g}) = \Gamma(\mathcal{L}, g)$ by comparing their parity-check matrices in systematic form. If they match, then $\tilde{g}(x) = \hat{g}(x)$.

(5) Return $\tilde{g}(x)$.

This algorithm returns a non-zero $\tilde{g}(x)$ if and only if there exists an irreducible polynomial $g'(x) \in \mathbb{F}_{2^m}[x]$ with $\Gamma(\tilde{\mathcal{L}}, g') = \Gamma(\mathcal{L}, g)$. In that case holds $\Gamma(\tilde{\mathcal{L}}, \tilde{g}) = \Gamma(\mathcal{L}, g)$.

Proof. If $\tilde{g}(x)$ is non-zero, by Item (4), it applies $\Gamma(\tilde{\mathcal{L}}, \tilde{g}) = \Gamma(\mathcal{L}, g)$. Conversely, if there is an irreducible $g'(x) \in \mathbb{F}_{2^m}[x]$ of degree t with $\Gamma(\tilde{\mathcal{L}}, g') = \Gamma(\mathcal{L}, g)$, then $g'(x)$ is an irreducible factor of $h(x)$, i.e. $g'(x) \in \mathcal{G}$. This $g'(x)$ is processed in Item (4) and ensures $\tilde{g}(x) \neq 0$. □

With $s = 5$ our simulations showed that, in practice, we always have two cases in Item (3): either $\deg(h(x)) = 2t$ and $\mathcal{G}$ contains exactly one element, or $h(x) = 1$ and $\mathcal{G} = \emptyset$. Our implementation is optimized for this observation.

The following proposition indicates the necessity of the quadratic equations in the solving procedure by showing that it is impossible to find a small set of support candidates only from linear fault equations.

Proposition 6.18 (Number of Linear Independent Polynomials).
Assume that $n > 2^{m-1}$. Let $\mathcal{F} \subseteq \mathbb{F}_{2^m}[x_1, \dots, x_n]$ be a fault equation system consisting only of linear polynomials. Then $\mathcal{F}$ contains less than $n - m$ linearly independent polynomials. Proof see [PGD+23, Prop. 18].

Note that the condition $n > 2^{m-1}$ is satisfied by all proposed parameter sets (see Table 3.3). So for the set of reduced quadratic polynomials $\mathcal{F}_{red} \in \mathbb{F}_{2^m}[x_{i_1}, \dots, x_{i_s}]$ in Item (2) of Proposition 6.16 we have $s \geq m$.

6.2.2 System of polynomial equations contains only a subset of unknowns

In the previous subsection, it was assumed that the system of polynomial equations obtained from the successful fault injections contains all unknown support elements. Many fault injections may be necessary to obtain equations in every single support element. But, an alternative secret key can be obtained even if the equation system does not determine every single support element. This may permit faster completion of the fault injection procedure.

Note that Proposition 6.17 (up to Item (3)) does not require the full set of support elements to determine a Goppa polynomial candidate, since only support elements $\tilde{\alpha}_k$ where $k \in \operatorname{supp}(c_i)$ with $i \in \{1, \dots, s\}$ are used. Hence, using suitable codewords $\mathbf{c}_i$, a Goppa polynomial candidate can be constructed from a subset of the support set $\mathcal{L}$ [PGD+23; KM22]. Before Item (4) of Proposition 6.17 can be executed for each returned Goppa polynomial candidate, the missing code locators need to be computed. This can be done using the Goppa polynomial candidate and the public key like described in Section 3.5.1 or [PGD+23, Remark 14]. The obtained set of code locators and its corresponding Goppa polynomial candidate are then checked for validity by comparing public keys (see Item (4) of Proposition 6.17). The full mathematical description is not part of this thesis and can be found in [PGD+23, Sec. 3.4].

6.3 Simulation

In this section, we demonstrate the viability of the key-recovery attack. We first use a C-implementation to simulate the attack (Section 6.3.1). For this we inject faulty variable values directly in software. We simulate the inputs and corresponding hashed outputs of the faulty decapsulation procedure. The de-hashing of these is described separately in Section 6.3.2, leading to the system of polynomial equations. This is solved to obtain an alternative secret key as described in Section 6.2 by a program written in Python3 using SageMath [Sag22].

An attacker cannot directly modify the software execution. Instead, there are different ways to conduct a fault attack, e.g. using a laser to corrupt hardware memory elements in a processor. To investigate whether this allows to inject the specific faults required for the presented attack as were identified at software-level, we execute the cryptosystem as software on a VP (Section 6.3.3). The VP implements the RISC-V ISA and allows us to inject faults into the hardware of the processor in order to study how they impact the executing software. For our software-error-model-based fault injection attack, we first analyze the binary to find the program sections that calculate the ELP and process the validity checks. The disassembly and its required alteration gives us the fault positions necessary to produce the identified faulty variable values. The necessary hardware fault attacks are then simulated on two levels; First, a fast ISA-level simulation assures that the hardware faults produce exploitable output. Second, an RTL simulation yields practicability of the fault attack with respect to a real CPU core's micro-architecture.

For our software simulation of the attack, we adapt the hardware accelerated implementation that makes use of vector arithmetics on the processor for faster runtime. To simulate the fault injections on RISC-V cores, we use the reference implementation (see Section 5.1.1 of Chapter 5).

6.3.1 Key-Recovery Simulation

We simulated the fault injections of Section 6.1.3 in C code and the solving of Section 6.2 in SageMath code. To speed up that simulation at C-level we work on AMD64 machines with the vector-accelerated AVX2 implementation. The organization of the implementation files and their names can be found in Section 5.1. An overview of the software simulation is given in Algorithm 14.

To model the attack, we adapt the implementation of the cryptosystem to include the effects of the ELPB, ELPZ and VCB faults. For the fault injections on the ELP, we have identified the following lines of code as injection points. The fault injection on the ELP happens between the function calls `bm(locator, s)` and `root(images, locator, L)` in `decrypt.c`. Fault injections on the ELP are modelled as bitwise operations on one of its coefficients. This way, the ELPB fault injection that sets two adjacent bits is implemented by setting one coefficient $a \to a \,\mathrm{OR}\, \zeta$, where ζ is an m-bit array containing only zeros except for two adjacent entries. The ELPZ fault injection is implemented by replacing all entries of one ELP coefficient with zeroes. Note that the fault value ξ corresponding to these injections as defined

Algorithm 14 Attack Simulation on C-level

1: Specify a ciphertext C as input for the decapsulation function as follows:
   i. Choose a plaintext $\mathbf{e}$ of Hamming weight $\mathrm{wt}(\mathbf{e}) = 2$.
   ii. Calculate the ciphertext $C = (\mathbf{c_0}, C_1) = (\mathbf{e}\mathbf{H}_{\mathrm{sys}}{}^{\top}, \mathcal{H}(2, \mathbf{e}))$
2: Inject a fault into the Error Locator Polynomial (ELP) as follows:
   i. Fix a fault value of $\zeta \in \mathbb{F}_{2^m}$.
   ii. Start the decapsulation process and let the Berlekamp-Massey algorithm calculate the ELP (in file `decrypt.c`).
   iii. Inject a constant or quadratic fault into the ELP (see Definition 6.9).
3: Inject a fault and reset the variable called m during decapsulation such that the following comparisons are bypassed (in file `operations.c`)
   a) Skip the comparison $\tilde{\mathbf{e}}'\mathbf{H}_{\mathrm{sys}}{}^{\top} = \mathbf{e}\mathbf{H}_{\mathrm{sys}}{}^{\top}$ (Alternative: in file `decrypt.c` clear 8-bit variable `ret_decrypt` during decapsulation).
   b) Skip the comparison $C_1' = \mathcal{H}(2, \tilde{\mathbf{e}}') = \mathcal{H}(2, \mathbf{e}) = C_1$ (Alternative: in file `operations.c` clear 8-bit variable `ret_confirm`).
4: Reconstruct $\tilde{\mathbf{e}}'$ from the output $\tilde{K}' = \mathcal{H}(1, \tilde{\mathbf{e}}', (\mathbf{e}\mathbf{H}_{\mathrm{sys}}{}^{\top}, \mathcal{H}(2, \mathbf{e})))$ of the decapsulation function as described in Section 6.3.2.
5: Calculate an alternative secret key as described in Section 6.2.

in Remark 6.3 is unknown, as it depends on the value of a. To skip the validity checks the variable `m` in file `operations.c` and in function `crypto_kem_dec_faulty` in the line `m = ret_decrypt | ret_confirm` is forced to 0. This gives $\mathcal{H}(1, \tilde{\mathbf{e}}', C)$ as output of the C-code for further analysis (see next Section 6.3.2).[1]

The simulation code is called repeatedly for different chosen ciphertexts and faults ζ to build a system of equations using Propositions 6.13 and 6.14, that can be solved with the methods from Section 6.2 to obtain an (alternative) secret key. To obtain linear equations in the support elements, faults are injected into the constant term of the ELP. In ELPB mode, we start with ζ having the two least significant bits non-zero. Then we generate faulty session keys from ciphertexts corresponding to plaintext vectors $\mathbf{e}$ with $\mathrm{wt}(\mathbf{e}) = 2$ and $\mathrm{supp}(\mathbf{e}) \in \{\{n-1, 0\}, \{0, 1\}, \{1, 2\}, \dots\}$. This is repeated for faults ζ with non-zero bits in other adjacent positions, until the resulting system of equations contains equations involving all the support elements. In ELPZ mode, there is only one way of injecting a fault, so that instead of different fault values ζ, ciphertexts corresponding to plaintext vectors $\mathbf{e}$ with $\mathrm{wt}(\mathbf{e}) = 2$ with increasing distance between the non-zero support elements $\mathrm{supp}(\mathbf{e}) \in \{\{n-1, 1\}, \{0, 2\}, \{1, 3\}, \dots\}$ are used to obtain a sufficiently large system of equations (this is also done in the ELPB-case if the number of possible ζ is exhausted before finding sufficiently many equations). The same procedure is used to inject faults on the linear term of the

[1] For the purposes of verifying the fault attack, the simulator also directly gives $\tilde{\mathbf{e}}'$ as output, sparing us the computational effort of de-hashing $\mathcal{H}(1, \tilde{\mathbf{e}}', C) \to \tilde{\mathbf{e}}'$.

	CAT I $n = 3488,\ m = 12,\ t = 64$	CAT V $n = 6688,\ m = 13,\ t = 128$	CAT V $n = 8192,\ m = 13,\ t = 128$
# of constant injections	31759	70700	56991
# of linear equations	8627	17649	21343
# of linear injections	293	564	266
# of quadratic equations	80	140	100

Table 6.1: Arithmetic Mean out of 100 simulations for ELPB.

	CAT I $n = 3488,\ m = 12,\ t = 64$	CAT V $n = 6688,\ m = 13,\ t = 128$	CAT V $n = 8192,\ m = 13,\ t = 128$
# of constant injections	8030	16516	8944
# of linear equations	6836	13482	8941
# of linear injections	94	121	100
# of quadratic equations	80	100	100

Table 6.2: Arithmetic Mean out of 100 simulations for ELPZ.

ELP in order to obtain quadratic equations in the support elements, finishing after an empirically determined fixed number of equations has been obtained. To confirm that the attack is working, we ran simulations of 100 random public/private key pairs, for several sets of parameters where $n \in \{3488, 6688, 8192\}$ (see Table 3.3). The average number of required fault injections for a successful attack on the different parameter sets are shown in Tables 6.1 and 6.2 for the fault modes ELPB and ELPZ respectively. The ELPZ-mode requires significantly fewer fault injections to complete the attack (compare with Remark 6.15). Parameter sets with smaller ratio $\frac{n}{2^m}$ also require more injections, as indicated by Remark 6.12. We find that the SageMath code usually takes only minutes to obtain an alternative secret key from the system of polynomial equations on an office computer.

6.3.2 De-hashing: Obtaining the faulty error vector from hash output

As output of the simulation in Section 6.3.1 we generate two files containing hashes $\tilde{K}' = \mathcal{H}(1, \tilde{\mathbf{e}}', C)$ with $(\mathbf{e}, \tilde{\mathbf{e}}')$ defining linear or quadratic equations in the support elements. Thanks to the small weight of the error vectors $\tilde{\mathbf{e}}'$, we can determine them from the hashes in a brute-force manner as follows.

First, we determine whether the zero element is part of the support set (Remark 6.4) and determine its index if present, according to Remark 6.7. This requires only one fault injection, giving the output $\mathcal{H}(1, \mathbf{e}', C)$ for $C = (\mathbf{0}, \mathcal{H}(2, \mathbf{0}))$, with $\mathrm{wt}(\mathbf{e}') \in \{0, 1\}$. The support $\mathrm{supp}(\mathbf{e}')$ specifies the index of the zero element in the support set, if it is present. It is determined from the hash by calculating all $n + 1$ possible hashes until the match with the output is found. Next, for every $\tilde{K}'$ we calculate the hashes $\mathcal{H}(1, \mathbf{v}, C)$ for the chosen ciphertext $C = (\mathbf{c_0}, \mathcal{H}(2, \mathbf{e}))$ and all possible $\tilde{\mathbf{e}}'$, as described

in Remark 6.6. When a match is found, $\tilde{\mathbf{e}}'$ has been determined.

We run the de-hashing on a computer with AMD EPYC 7543P processor running up to 3.3 GHz using 32 cores (64 threads) on Arch Linux with kernel 5.19.7. For the ELPB case we have about $\binom{n}{2}$ different $\tilde{\mathbf{e}}'$ to check for every hash output (number of constant injections plus number of linear injections in Table 6.1). Depending on the cryptosystem and its parameters, the total runtime on our system spans a few hours up to a few days. For the ELPZ case we have about $\binom{n}{1} = n$ different $\tilde{\mathbf{e}}'$ to check for every hash output (number of constant injections plus number of linear injections in Table 6.2). The running time on our system is a few seconds.

6.3.3 Simulation at Register Transfer Level

The Register Transfer Level (RTL) is an abstraction level in modeling the hardware of integrated circuits. On this level, the system is represented as digital signal flows between hardware registers and logical operations performed on the signals. The RTL is the abstraction level that is used in Hardware Description Languages (HDLs) such as Verilog and Very High Speed Integrated Circuits Hardware Description Langauge (VHDL) to describe an electronic circuit on the basis by data flow and timing models. The description in HDL can be converted to a gate-level representation, also called netlist, with a logic synthesis tool to derive the actual wiring. Placement and rooting tools then create the physical hardware layout that can be deployed to an FPGA or an ASIC.

We evaluated the vulnerability of two open source RISC-V cores, the OpenHW-Group's CV32E40P [Opea], formerly known as RI5CY [GST+17; DCR+17], and its security oriented derivative CV32E40S [Opeb]. For this we simulated fault injections on these specific micro-architectures using its hardware descriptions in Verilog at RTL. These simulations provide information about which fault injections trigger errors on these micro-architectures that lead to our exploit. In particular, a simulation can be used to find the fault locations and injection timings that lead to the intended attack. In order to find these fault locations, we try all possible bits (single bit set/reset) at all possible simulation steps (clock cycles) if they give us our desired exploit in an exhaustive search campaign.

A simulation at RTL is in general quite complex and a big limiting factor is the simulation time. Thus, it is desirable to minimize the simulation effort by simulating only specific time windows/frames of the decapsulation algorithm. The intended attacks that lead to an exploit described on a mathematical level, as in Proposition 6.13 or Proposition 6.14 and Section 6.1.4 are associated to the programming lines in the source code of the software implementation in C-code. First, the programming lines in the software implementation on C-code, which need to be corrupted in order to lead to the intended attack, are localized, as in Line 2 and Line 3 of Algorithm 14. These programming lines are converted to RISC-V instructions in assembly via a compiler. Second, we simulate the assembly instructions on an ISS implementing the RISC-V ISA to identify interesting instructions to further narrowing the time frame for a consecutive RTL simulation. Third, the selected time frame of instructions on

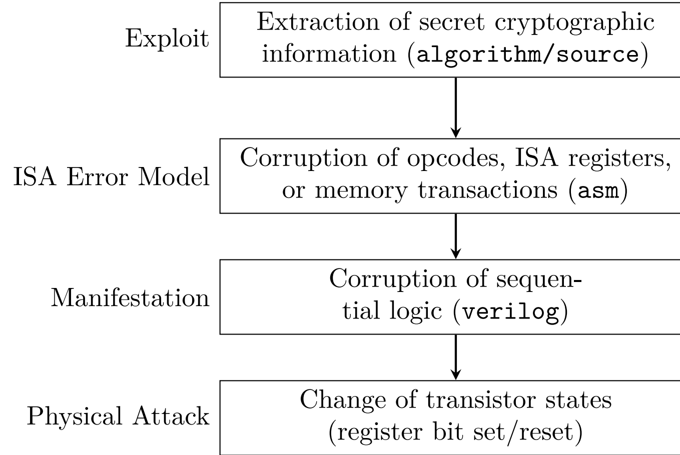


Figure 6.2: Abstraction Levels for narrowing the RTL Fault Injection Simulation.

Core: CV32E40-	P	S
clock cycles at ISA simulation	301	301
total architectural bits	6.001	16.586
# of checks	1.806.301	4.992.386
# of exploits for validity check bypass N_{VCB}	114	57
# of unique faulted bits	93	39

Table 6.3: Single-Bit RTL fault injection results for validity check fault scenario VCB.

ISA-level are then fed into the RTL fault simulation of the specific micro architectures of CV32E40P and CV32E40S to search for all possible single faults that lead to the desired exploit.

This approach is shown in Fig. 6.2. It identifies the rough fault injection points to further evaluate the feasibility of a physical attack in an exhaustive search campaign at RTL. The benefit of this approach is to focus only on the program sections that are critical on the source code to apply the exhaustive search on bit level on a narrow time frame to reduce RTL simulation time.

To simulate the fault injections on RTL, the original SystemVerilog hardware descriptions of CV32E40P/S in [Opea; Opeb] are transferred to a cycle accurate C++/ SystemC environment using the synthesis tool Verilator [Sny] and are then modified with an LLVM-based automated source code transformation tool [GM23] to integrate the fault injections. The clock cycle accurate simulation is executed in the C++/SystemC environment with a fault injection capability into sequential storage elements, e.g. flip-flops.

Core: CV32E40-	P	S
clock cycles at ISA simulation	521	521
total achitectural bits	6.001	16.586
# of checks	3.126.521	8.641.306
# of exploits for coeff. bit corruption N_{ELPB}	69	57
# of unique faulty bits	35	29
# of exploits for coeff. zeroing N_{ELPZ}	508	212
# of unique faulted bits	225	94

Table 6.4: Single-Bit RTL fault injection results for ELP coefficient fault scenario ELPB and ELPZ.

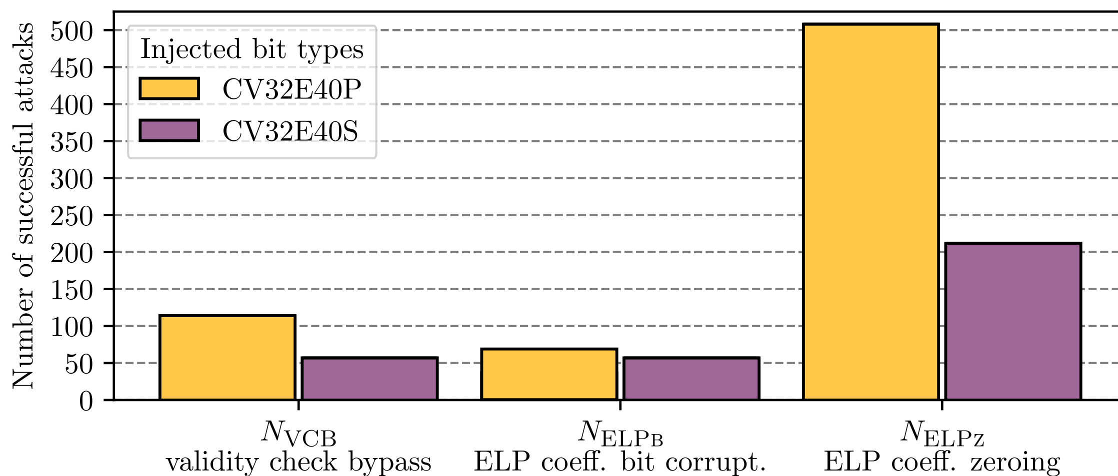


Figure 6.3: Single-Bit RTL Fault Injection Results.

In total 6001 bits for CV32E40P and 16586 bits for CV32E40S were faulted for each clock cycle and checked if the desired exploit can be achieved. The simulation results of the VCB fault (see Section 6.1.4) to bypass the validity checks are shown in Table 6.3. For VCB the ISA simulation resulted in 301 clock cycles for further RTL simulation. For CV32E40P, in 114 cases a single bit fault could achieve the VCB exploit having 93 unique injections points identified. While for the CV32E40S the exploit could be achieved in only 57 cases, although the security oriented architecture contains more injectable bits. Most of the VCB exploits are achieved by faulting bits that result in instruction skips. The simulation results of the ELPB and ELPZ faults (see Section 6.1.6) that corrupt the ELP coefficients are shown in Table 6.4. For ELPB and ELPZ the ISA simulation resulted in 521 clock cycles for further RTL simulation. The ELPB exploit is achieved in 69 cases for CV32E40P having 35 unique injections points identified and in 57 cases for CV32E40S having 29 unique injections points identified. A small number of ELPB exploits are achieved by directly faulting bits in the register storing the ELP value. The ELPZ exploit is achieved in 508 cases

for CV32E40P having 225 unique injections points identified and in 212 cases for CV32E40S having 94 unique injections points identified. Most of the ELPz exploits are achieved by faulting bits in the memory store instructions. These results are summarized in Fig. 6.3. The security oriented CV32E40S core is more resistant against fault attacks compared to CV32E40P due to its hardware based countermeasures.

Furthermore, the VCB and ELPz exploits can be jointly achieved with one single bit fault. This was observed for 47 bits in CV3240P and for 17 bits in CV32E40S, making the complete attack significantly less complex by requiring only one fault injection per ciphertext input and requiring e.g. only a single laser beam source setup.

Conclusion In this chapter a fault attack on Classic McEliece KEM that finds an alternative secret key has been described and simulated. The attack exploits that at decoding the roots of an ELP are searched among the code locators of the secret key. The main idea is to inject faults into the physical device during the decoding computation in order to generate faulty ELPs. These faulty ELPs are evaluated and the indices of the code locators are returned. With the indices of the code locators, a system of polynomial equations with the values of the code locators as unknowns can be set up. This system of polynomial equations can be solved using Gröbner basis and Buchberger's algorithm. The solution gives multiple possible values of the code locators, which all are checked whether they meet the requirements of being part of an alternative secret key by comparing its public key with the public key of the attacked cryptosystem.

We found two different kinds of fault injection methods: ELPb and ELPz. In ELPb single and adjacent bits of a coefficient of the ELP get corrupted. In ELPz all bits of a coefficient of the ELP are corrupted to zero.

As an extension of the attack, it is also possible to solve a system of polynomial equations that contains only a subset of the indices of the unknown code locators. The remaining indices, that are not part in the polynomial equations, can be found by brute-forcing possible solutions, calculating Goppa polynomial candidates and checking if they constitute a valid secret key.

To ensure the result of the faulty computation is output from the decapsulation in the KEM, another fault injection on the VCB is necessary. The output is the hash of the error vector. Thus, if the faulty ELPs have a degree of 1 and 2 the hash can be dehashed by brute-force. This can be achieved by choosing ciphertext inputs of the decapsulation that are generated from plaintext with Hamming weights of 2. For the ELPb case the brute-force dehashing runtime spans a few hours up to a few days on a 3.3GHz processor with 64 threads. For the ELPz case the brute-force dehashing runtime takes a few seconds on the same processor.

We simulated the fault injections on two RISC-V processors at RTL. It shows the possibilities for successful fault injections for ELPb and ELPz. In particular, we found that ELPz and VCB can be jointly achieved by a single fault injection.

The attack described in this chapter is based on the implementations of Classic McEliece that were submitted for the NIST PQC competition Round 3. For the current version of Round 4 the authors removed the check in Line 5 of Algorithm 6

in their implementations. This makes the VCB fault injections of our attack easier as we anticipate that there would be more possible successful fault injection locations on the processor.

For future works, it remains to execute our whole attack on real hardware. The challenge in here could be especially to get the right injection timings. Additionally, it also includes to find the right physical injection locations which are dependent on the layout of the chip.

The current method of solving the system of polynomial equations is not robust against mistakes in the fault injection procedure. Thus, the solving of the system could be improved by developing a tolerance against incorrectly obtained polynomial equations.

Chapter 7

Conclusion

This thesis dealt with code-based post-quantum cryptosystems, in particular the McEliece cryptosystem, which is based on classical Goppa codes, and variants of it. A broad range of results ranging from their mathematical foundations, their efficient implementation on hardware, to possible hardware attacks were presented. The code class of binary Generalized Goppa Codess (GGCs) was examined for a reduced public key size with equivalent security level compared to classical binary Goppa codes. Next, the performance of Classic McEliece implementations was investigated and the computational complexity (time and resource consumption) of the key generation was reduced via vectorization of the Gaussian Elimination Algorithm (GEA). Finally, a hardware fault attack on Classic McEliece was presented by compromising the Error Locator Polynomial (ELP) during decryption.

The thesis starts by giving background knowledge on code-based cryptosystems, introducing some key concepts of finite fields and coding theory. Syndrome-based decoding was explained and it was illustrated how a cryptosystem can be built upon codes that are otherwise used for channel coding. We looked into the security of code-based cryptosystems and its dependence on the syndrome-decoding problem. The McEliece cryptosystem and its dual version, the Niederreiter cryptosystem as a public-key encryption system and as well as Classic McEliece as a Key Encapsulation Mechanism (KEM) were presented. The best known attacks on the McEliece and Niederreiter cryptosystem with Goppa codes were given. Key attacks are structural attacks that target the revelation of the secret key, while a message attack targets the revelation of the message that should be kept secret. Message attacks for McEliece go back to Prange as the first idea for an Information Set Decoding (ISD) algorithm. Since then, further algorithms have been developed, which reduce the number of operations needed, usually at the cost of more memory use. For a KEM, it is also important to fulfil the requirements of Indistinguishability under Chosen Plaintext Attacks (IND-CPA) and Indistinguishability under Chosen Ciphertext Attacks (IND-CCA2), which were explained.

In Chapter 4 the code class of GGC were studied. It was shown how binary GGC are constructed, and their properties and error-correction capabilities were derived. GGC are defined via a Goppa polynomial of degree t and code locators that are fractions of polynomials. For binary GGC, the code locators are defined by polynomials with

degree at most ℓ and their formal derivatives. Classical binary Goppa codes are binary GGC with $\ell = 1$. A construction of parity-check matrices for binary GGC with code locators of arbitrary degree was developed. The maximum code length of binary GGC is bounded by the number of irreducible polynomials in $\mathbb{F}_{2^m}[x]$ for given degrees. For classical binary Goppa codes with $\ell = 1$ the code length is at most 2^m, while for GGC with higher degree polynomials the maximum possible code length is larger. The lower bound of the minimum distance is $\frac{t+1}{\ell}$, while for separable binary GGC it is $\frac{2t+1}{\ell}$. Using the Extended Euclidean Algorithm (EEA) for decoding separable GGC, the unique decoding radius can be increased to $\left\lfloor \frac{d_{\text{sep}}}{2} \right\rfloor$. For separable binary GGC and even-degree code locators the lower bound of the minimum distance can be improved to $\frac{2t+2}{\ell}$. Binary GGC were investigated for use in a Niederreiter cryptosystem, with focus on the security level based on the work factor of Prange [Pra62; AM87] and the public-key size. No code parameters of separable binary GGC with $\ell > 1$ could be found, such that the public key size would be improved over classical Goppa codes (where $\ell = 1$) for a security level based on the work factor $\mathrm{WF}_{\text{Prange}}$. However, the field size can be reduced at the cost of a smaller security level or a larger public key size to improve the complexity of all calculations, including the construction of a parity-check matrix. Note that the extension degree m must not be chosen too small, as structural attacks are known for binary classical Goppa codes with $m = 1$ and $m = 2$, which are expected to also apply to GGC.

In Chapter 5 the implementations of the Classic McEliece KEM and their performance were examined. First, the different implementations of Classic McEliece were detailed and profiled regarding their runtime on a x86-64 processor. The GEA was identified as the most time-consuming algorithm. An analysis of the GEA showed that it can be accelerated with Instruction Set Architecture (ISA) vector extensions. Next, the RISC-V ISA and its vector extension have been introduced and examples given. Assembly instructions of the RISC-V Vector Extension (RVV) were integrated into the GEA. The vectorized and non-vectorized versions of the GEA have been simulated for RISC-V on the instruction set simulator Extendable Translating Instruction Set Simulator (ETISS). It was demonstrated that by using the ISA vector extension the number of load and store accesses could be reduced and thus also the overall runtime decreased. For the GEA with RVV integration, an acceleration by a factor of 6 to 18 for a memory port width W of 64 bits to 256 bits, respectively, could be achieved.

Chapter 6 described and simulated a fault attack on the Classic McEliece KEM. The central idea is to induce faults on the physical device during decoding in order to output an error vector of a corrupt ELP and thus gather information about the secret support elements of the Goppa code. For this, chosen ciphertexts coming from small-weight plaintexts and also faults on the validity checks during decapsulation are needed. Repeating the fault injections with a series of chosen ciphertexts results in a system of polynomial equations that can be solved to obtain an alternative secret key. Two fault injection methods on the ELP were found, either corrupting single and adjacent bits of a coefficient of the ELP (ELPB), or setting all bits of an ELP coefficient to zero (ELPZ). The decoding procedure determines which code locators

are roots of the faulty ELP and produces a binary error vector with non-zeros at the indices of the found root code locators. For each error vector (indices of non-zero positions) an equation with indeterminates in the unknown code locators can be constructed. Faulting the constant or the linear coefficient of the ELP, a linear or quadratic equation is obtained. To receive the error vector as output of the decapsulation procedure more actions are necessary. The decapsulation procedure calculates a hash of the error vector containing the indices of the code locators that are roots of the faulty ELP. To output this hashed error vector another fault (VCB) is needed to bypass the validity checks of the decapsulation. Although the output is in form of a hash, its contents can be retrieved by brute force if the error vector has a small weight. This is achieved by choosing ciphertexts generated from plaintexts with Hamming weights of 2. The runtime of brute-forcing the hashed error vectors was found to be a few seconds (ELPz) to a few days (ELPb) on a 3.3 GHz processor with 64 threads. The attack is simplified if the zero element is among the support set. It can be found by choosing a ciphertext generated from the all-zero vector and a single VCB fault. The probability of the zero element existing in the support set is $\frac{n}{2^m}$. Thus, a smaller difference between code length n and field size 2^m increases the probability. The equations obtained through fault injections for different chosen ciphertexts are gathered into a system of polynomial equations with the code locators as indeterminates. This system needs to contain not only linear equations, but also quadratic equations in order to be solved. The number of fault injections needed to obtain enough linear equations and quadratic equations for solving the system were simulated using a C-code and SageMath implementation. The ELPz fault injection method is much more efficient than the ELPb fault injection method. Having the code locators, corresponding Goppa polynomial candidates are calculated using codewords that are calculated from the public key. Each Goppa polynomial candidate and the obtained support set are checked whether they comprise a valid alternative secret key by calculating its public key and comparing it with the known public key. This is also possible having only a subset of the code locators, as the remaining code locators can be restored using the Goppa polynomial candidate. The fault injections were simulated at the Register Transfer Level (RTL) of two RISC-V processors, confirming the applicability of the attack. There are more possible injection points that lead to ELPz than to ELPb. Thus, the ELPz fault injection method is easier to achieve. The simulation also revealed that ELPz and VCB faults can be jointly achieved by a single fault injection per chosen ciphertext.

In this thesis, research towards an efficient and secure implementation of McEliece-like code-based post-quantum cryptosystems was done. The work presented in this thesis illustrates the maturity of the Classic McEliece KEM, at a time where it is close to becoming standardized. Work on the underlying code set out to address the public-key size by substituting classical Goppa codes with GGC. Among the GGC, classical Goppa codes were found to have the smallest public-key size for given security level. As widespread adoption comes closer, efficient implementation also on embedded devices comes into focus. The matrix operations inherent to the cryptosystem are well-suited for vectorization, as demonstrated in this thesis for processors implementing RISC-V,

an open source ISA. The McEliece cryptosystem based on classical Goppa codes is an old cryptosystem that has a long history of research and thus there is high confidence in its security. However, fault attacks such as the one described in this thesis need to be considered as threats to implementations on highly secured devices.

Appendix A

Source Code of Vectorized GEA

The following source code is the vector accelerated GEA using RVV version 0.9. This code was simulated and used for the performance analysis in Section 5.6 of Chapter 5.

```
#include <stdlib.h>
#include <stdio.h>

#define GFBITS 13
#ifdef MCELIECE_8192128
#define  SYS_T 128
#define  SYS_N 8192
#else
#ifdef MCELIECE_6960119
#define  SYS_T 119
#define  SYS_N 6960
#else //MCELIECE_6688128
#define  SYS_T 128
#define  SYS_N 6688
#endif
#endif

typedef uint8_t mat_t[GFBITS * SYS_T][SYS_N/8];

int gaussian_elim(mat_t mat){
   unsigned char mask=0;
   unsigned char * pt=0;
   unsigned char ret=0;
   int arg1=128;
   // clear vector register v0 (masking register) with vector
      register length of VLEN=1024
   asm volatile("vsetvli %[ret], %[arg1], e8, m1 \n" // set SEW=8,
       LMUL=1, VL=128
   :[ret] "=r" (ret)
   :[arg1] "r"(arg1));
   asm volatile("vand.vi v0, v0, 0\n" //set all bits to 0
   "vxor.vi v0, v0, -1 \n"); //set all bits to 1
   arg1=SYS_N/8;
   asm volatile("vsetvli %[ret], %[arg1], e8, m8 \n" // set SEW=8,
       LMUL=8, VL=870
   :[ret] "=r" (ret)
```

```
:[arg1] "r"(arg1));

printf("Gauss Elimination Algorithm for %d%d: \n", SYS_N, SYS_T
    );
// Gaussian elimination algorithm from Classical McEliece NIST
    submission Round 2, see: https://classic.mceliece.org/
for (int i = 0; i < (GFBITS * SYS_T + 7) / 8; ++i){
   for (int j = 0; j < 8; ++j){
      int row = i*8 + j;

      if (row >= GFBITS * SYS_T)
      break;

      //Load matrix row into vector register group
      pt=&(mat[row][0]); // set address to current row of
          matrix
      asm volatile("vle8.v v8, (%[pt]) \n" // Load into vector
          register group v8
      :[pt] "+r"(pt));

      for (int k = row + 1; k < GFBITS * SYS_T; ++k)
      {
         mask = mat[ row ][ i ] ^ mat[ k ][ i ];
         mask >>= j;
         mask &= 1;
         mask = -mask;

         //Vector extension replaces loop: for (c = i; c <
             SYS_N/8; c++) mat[ row ][ c ] ^= mat[ k ][ c ] &
             mask;
         pt=&(mat[k][0]);
         asm volatile("vle8.v v16, (%[pt]) \n" // Load k-th row
              into vector register group v16
         :[pt] "+r"(pt));

         asm volatile("vand.vx v24,v16,%[mask],v0.t\n" // v24 =
              v16 & mask
         :[mask] "+r"(mask));
         asm volatile("vxor.vv v8,v8,v24,v0.t \n"); // v8=v8^
             v24

         pt=&(mat[row][0]);
         asm volatile("vse8.v v8, (%[pt]) \n" // Store current
             row (row of pivot) from vector register group to
             memory
         :[pt] "+r"(pt));

      }

      if ( ((mat[ row ][ i ] >> j) & 1) == 0 ) // return if not
          systematic
      {
         return -1;
```

```
        }

        for (int k = 0; k < GFBITS * SYS_T; ++k)
        {
            if (k != row)
            {
                mask = mat[ k ][ i ] >> j;
                mask &= 1;
                mask = -mask;

                // Vector Extension replaces loop: for (c = 0; c <
                    SYS_N/8; c++) mat[ k ][ c ] ^= mat[ row ][ c ] &
                     mask;
                pt=&(mat[k][0]);
                asm volatile("vle8.v v16, (%[pt]) \n" // Load k-th
                    row of matrix into vector register group v16
                :[pt] "+r"(pt));

                asm volatile("vand.vx v24,v8,%[mask]\n" // v24 = v8
                     & mask
                "vxor.vv v16,v16,v24 \n" // v16=v16^v24
                :[mask] "+r"(mask));

                asm volatile("vse8.v v16, (%[pt]) \n" // Store k-th
                     row from vector register group v16 to memory
                :[pt] "+r"(pt));
            }
        }
    }

    // Set Mask Register to replace loop: for (c = i; c < SYS_N
        /8; c++)
    arg1=1+i/8;
    asm volatile("vsetvli %[ret], %[arg1], e8, m1 \n" // Switch
        to SEW=8, LMUL=1, VL=1+i/8
    :[ret] "=r" (ret)
    :[arg1] "r"(arg1));
    arg1=255<<(1+i%8);
    asm volatile("vand.vx v0, v0, %[arg1] \n"
    :[arg1] "+r"(arg1)); // Shift left for 1 bit position
    arg1=SYS_N/8;
    asm volatile("vsetvli %[ret], %[arg1], e8, m8 \n" // Switch
        back to SEW=8, LMUL=8, VL=870
    :[ret] "=r" (ret)
    :[arg1] "r"(arg1));
  }

}

int main()
{
    mat_t mat = {
        #ifdef MCELIECE_8192128
```

```
        #include "matrix_begin8192128.data"
        #else
        #ifdef MCELIECE_6960119
        #include "matrix_begin6960119.data"
        #else //MCELIECE_6688128
        #include "matrix_begin6688128.data"
        #endif
        #endif
    };

    gaussian_elim(mat);

    for (int i=0; i<GFBITS * SYS_T; ++i){
        for(int j=0; j<(SYS_N/8); ++j){
            printf("%02X\t",mat[i][j]);
        }
        printf("\n");
    }
}
```

Appendix B

Gröbner Basis and Buchberger's Algorithm

This appendix introduces the Gröbner basis and its use in solving systems of multivariate polynomial equations. It is used extensively in Section 6.2, and the most important concepts are outlined here for completeness. This appendix is based on the references [Stu05] and [Buc01].

B.1 Background of Gröbner Basis

The general idea of the method of Gröbner basis is to transform a given set $\mathcal{F}$ of multivariate polynomials $\mathrm{K}[x_1, \dots, x_n]$ into another set $\mathcal{G}$ of multivariate polynomials with certain "nice" properties. The set $\mathcal{G}$ is called Gröbner basis. In the following, these properties are introduced before defining the Gröbner basis. Its utility in solving systems of multivariate polynomial equations is demonstrated in the subsequent section.

Definition B.1 (Ideal).
For a finite set $\mathcal{F} \subseteq \mathrm{K}[x_1, \dots, x_n]$, the ideal $\mathrm{Ideal}(\mathcal{F})$ generated by $\mathcal{F}$ is the set of all polynomial linear combinations such that

$$\mathcal{I} = \mathrm{Ideal}(\mathcal{F}) = \langle \mathcal{F} \rangle = \{p_1 f_1 + \cdots + p_r f_r \mid f_1, \dots, f_r \in \mathcal{F} \text{ and } p_1, \dots, p_r \in \mathrm{K}[x_1, \dots, x_r]\}.$$

The ideal of $\mathcal{F}$ is also denoted by $\langle \mathcal{F} \rangle$. The set $\mathcal{F}$ and its Gröbner basis generate the same ideal $\langle \mathcal{F} \rangle = \langle \mathcal{G} \rangle$. Next, we define an ordering of sets of multivariate polynomials by their terms.

Definition B.2 (Term Order).
A term order $\prec$ on $\mathrm{K}[x_1, \dots, x_n]$ is a total order on the set of monomials $x^a = x_1^{a_1} \cdot x_2^{a_2} \cdots x_n^{a_n}$, that has the following properties:

(a) multiplicativity, $x^a \prec x^b$ implies $x^{a+c} \prec x^{b+c} \;\; \forall a, b, c, \in \mathbb{N}^n$

(b) the constant monomial is the smallest monomial, $1 \prec x^a \;\; \forall a \in \mathbb{N}^n \backslash \{0\}$.

Different term orders exist. For example, the degree lexicographic order for $n = 2$ is $1 \prec x_1 \prec x_2 \prec x_1^2 \prec x_1x_2 \prec x_2^2 \prec x_1^3 \prec x_1^2x_2 \prec \cdots$. For a fixed term order, every polynomial $f \in \mathcal{F}$ has a unique initial term.

Definition B.3 (Initial Term).
Let $f \in \mathcal{F} \subseteq \mathrm{K}[x_1, \ldots, x_n]$ with $\prec$ a fixed term order. Then, every polynomial f has an unique initial term $\mathrm{in}_\prec(f) = x^a$. The initial term is the largest term with non-zero coefficient in the expansion of f.

For example, the following is a polynomial $f \in \mathbb{N}[x_1, x_2]$ written in decreasing degree lexicographic order, with the initial term first: $f = 5x_2^2 + 3x_1x_2 + 4x_1^2 + 7x_2 + 11x_1 + 13$. Using the initial terms, an initial ideal can be defined.

Definition B.4 (Initial Ideal).
The initial ideal $\mathrm{in}_\prec(\mathcal{I})$ is the ideal generated by the initial terms of all the polynomials in $\mathcal{I}$.

$$\mathrm{in}_\prec(\mathcal{I}) = \langle \mathrm{in}_\prec(f) : f \in \mathcal{I} \rangle$$

Then, the Gröbner basis of $\mathcal{F}$ can be defined.

Definition B.5 (Gröbner basis).
A finite subset $\mathcal{G}$ of an ideal $\mathcal{I}$ is a Gröbner basis with respect to the term order $\prec$ if the initial terms of the elements in $\mathcal{G}$ generate the initial ideal.

$$\mathrm{in}_\prec(\mathcal{I}) = \langle \mathrm{in}_\prec(g) : g \in \mathcal{G} \rangle$$

If $\mathcal{G}$ is a Gröbner basis for $\mathcal{I}$, then any finite subset of $\mathcal{I}$ that contains $\mathcal{G}$ is also a Gröbner basis.

Definition B.6 (Reduced Gröbner basis).
A reduced Gröbner basis is defined if

(a) for each $g \in \mathcal{G}$ the coefficient of $\mathrm{in}_\prec(g)$ in g is 1,

(b) $\mathrm{in}_\prec(\mathcal{I}) = \mathrm{in}_\prec(\mathcal{G})$ and no smaller subset of $\mathcal{G}$ generates the initial ideal of $\mathcal{I}$,

(c) no non-initial term of any $g \in \mathcal{G}$ lies in $\mathrm{in}_\prec(I)$.

Theorem B.7 (Unique reduced Gröbner basis).
For a given term order $\prec$, every ideal $\mathcal{I} \in K[x_1, \ldots, x_n]$ has a unique reduced Gröbner basis.

B.2 Solving Systems of Polynomial Equations

The Gröbner basis can be used to solve systems of polynomial equations as in Section 6.2. Let $\mathrm{K} \subseteq \mathbb{F}_q$ and $\mathcal{F}$ be a finite set of polynomials in $\mathrm{K}[x_1, x_2, \ldots, x_n]$, then the variety of $\mathcal{F}$ can be defined.

Algorithm 15 Buchberger's algorithm [Buc01]

Input: Finite set $\mathcal{F}$ of multivariate polynomials
Output: Gröbner basis $\mathcal{G}$
1: Set $\mathcal{G} = \mathcal{F}$
2: **for** any pair of polynomals $f_1, f_2 \in \mathcal{G}$ **do**
3: Compute the S-polynomial of f_1, f_2.
4: Reduce the S-polynomial to a reduced form h with respect to $\mathcal{G}$.
5: **if** $h = 0$ **then** continue
6: **else if** $h \neq 0$ **then** add h to $\mathcal{G}$.

Definition B.8 (Variety).
The variety of $\mathcal{F}$ is the set of all common zeros.

$$\mathcal{V}(\mathcal{F}) = \{(z_1, \ldots, z_n) \in \mathbb{F}_q^n \mid f(z_1, \ldots, z_n) = 0 \;\; \forall f \in \mathcal{F}\}$$

The variety does not change when replacing $\mathcal{F}$ by another set of polynomials which generate the same ideal in $\mathrm{K}[x_1, \ldots, x_n]$. Hence, the reduced Gröbner basis $\mathcal{G}$ for the ideal $\langle \mathcal{F} \rangle$ has the same variety as $\mathcal{F}$.

$$\mathcal{V}(\mathcal{F}) = \mathcal{V}(\langle \mathcal{F} \rangle) = \mathcal{V}(\langle \mathcal{G} \rangle) = \mathcal{V}(\mathcal{G})$$

The advantage of $\mathcal{G}$ is that it reveals properties of the variety that are not directly visible from $\mathcal{F}$. As an example, it can be shown that the variety is only empty if and only if $\mathcal{G} = \{1\}$.

To solve a system of polynomial equations it is handy to compute the Gröbner basis, as the roots of the polynomials of the Gröbner basis are also the roots of the polynomial system of equations.

To construct a Gröbner basis $\mathcal{G}$ of an arbitrary finite set $\mathcal{F}$, Buchberger's algorithm is used. It makes use of the S-polynomial.

Definition B.9 (S-polynomial).
Consider two polynomials g_1 and g_2. Their S-polynomial is given by

$$m_2 g_1 - m_1 g_2,$$

where m_1 and m_2 are monomials of smallest possible degree such that $m_2 \mathrm{in}_{\prec}(g_1) = m_1 \mathrm{in}_{\prec}(g_2)$.

The S-polynomial of two polynomials is computed by [Buc01]:

1. Multiplication of the polynomials by such monomial factors that the leading power term of the resulting products becomes equal, namely the least common multiple of the leading power terms of the two polynomials.

2. Substraction of one product from the other such that the leading power term cancels.

Example B.10.
Let $g_1 = xy - 2y$ *and* $g_2 = 2y^2 - x^2$*. Then, the S-polynomial of* g_1 *and* g_2 *is* $m_2 g_1 - m_1 g_2 = \text{S-polynomial}[g_1, g_2]$*. This results in*

$$2xy^2 - 4y^2 - 2xy^2 + x^3 = x^3 - 4y^2 \tag{B.1}$$

and the $\text{S-polynomial}[g_1, g_2] = x^3 - 4y^2$*.*

With the Buchberger's algorithm in Algorithm 15 the Gröbner basis can be computed. During the algorithm, the set $\mathcal{G}$ grows and thus also the number of possible S-polynomials grows during the algorithm. However, the algorithm does always terminate. The Buchberger's algorithm has an exponential time complexity.

Bibliography

[AM87] Carlisle M. Adams and Henk Meijer. "Security-Related Comments Regarding McEliece's Public-Key Cryptosystem". In: *Advances in Cryptology — CRYPTO '87*. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1987, pp. 224–228. DOI: 10.1007/3-540-48184-2_20 (cit. on pp. 32, 33, 47, 48, 92).

[AKA+20] Alon Amid, Krste Asanovic, Allen Baum, Alex Bradbury, Tony Brewer, Chris Celio, Aliaksei Chapyzhenka, Silviu Chiricescu, Ken Dockser, Bob Dreyer, Roger Espasa, Sean Halle, John Hauser, David Horner, Bruce Hoult, Bill Huffman, Constantine Korikov, Ben Korpan, Hanna Kruppe, Yunsup Lee, Guy Lemieux, Filip Moc, Rich Newell, Albert Ou, David Patterson, Colin Schmidt, Alex Solomatnikov, Steve Wallach, Andrew Waterman, and Jim Wilson. *RISC-V Vector Extension, v0.9*. May 2020. URL: https://github.com/riscv/riscv-v-spec/releases/tag/0.9 (visited on 07/16/2023) (cit. on pp. 61, 63).

[ABB+22] Nicolas Aragon, Paulo S. L. M. Barreto, Slim Bettaieb, Loic Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Santosh Gosh, Shay Gueron, Tim Güneysu, Carlos Aguilar Melchor, Rafael Misoczki, Edoardo Persichetti, Nicolas Sendrier, Jean-Pierre Tillich, Valentin Vasseur, and Gilles Zemor. *BIKE - Bit Flipping Key Encapsulation*. 2022. URL: https://bikesuite.org/ (visited on 09/10/2023) (cit. on p. 3).

[BCGM07] M. Baldi, F. Chiaraluce, R. Garello, and F. Mininni. "Quasi-Cyclic Low-Density Parity-Check Codes in the McEliece Cryptosystem". In: *2007 IEEE International Conference on Communications*. ISSN: 1938-1883. June 2007, pp. 951–956. DOI: 10.1109/ICC.2007.161 (cit. on pp. 20, 31).

[Bar97] Alexander Barg. *Complexity Issues in Coding Theory*. en. Tech. rep. TR97-046. ISSN: 1433-8092. Electronic Colloquium on Computational Complexity (ECCC), Oct. 1997. URL: https://eccc.weizmann.ac.il/eccc-reports/1997/TR97-046/index.html (visited on 02/05/2023) (cit. on p. 17).

[Bar94] Barg, Alexander. "Some New NP-Complete Coding Problems". ru. In: *Problemy Peredachi Informatsii* 30.3 (1994), pp. 23–28. URL: https://www.mathnet.ru/eng/ppi241 (visited on 02/05/2023) (cit. on p. 17).

[BJMM12] Anja Becker, Antoine Joux, Alexander May, and Alexander Meurer. "Decoding Random Binary Linear Codes in 2(n/20): How 1+1=0 Improves Information Set Decoding". en. In: *Advances in Cryptology – EUROCRYPT 2012*. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2012, pp. 520–536. ISBN: 978-3-642-29011-4. DOI: 10.1007/978-3-642-29011-4_31 (cit. on p. 34).

[BL05] Thierry P. Berger and Pierre Loidreau. "How to Mask the Structure of Codes for a Cryptographic Use". In: *Designs, Codes and Cryptography* 35.1 (Apr. 2005), pp. 63–79. ISSN: 1573-7586. DOI: 10.1007/s10623-003-6151-2 (cit. on p. 31).

[Ber73] E. Berlekamp. "Goppa codes". In: *IEEE Transactions on Information Theory* 19.5 (Sept. 1973), pp. 590–592. ISSN: 1557-9654. DOI: 10.1109/TIT.1973.1055088 (cit. on p. 10).

[Ber68] Elwyn R. Berlekamp. "Nonbinary BCH decoding (Abstr.)" In: *IEEE Transactions on Information Theory* 14.2 (1968), pp. 242–242. ISSN: 0018-9448. DOI: 10.1109/TIT.1968.1054109 (cit. on p. 26).

[BMT78] Elwyn R. Berlekamp, Robert J. McEliece, and Henk C. A. van Tilborg. "On the Inherent Intractability of Certain Coding Problems". In: *IEEE Transactions on Information Theory* 24.3 (May 1978), pp. 384–386. ISSN: 0018-9448. DOI: 10.1109/TIT.1978.1055873 (cit. on p. 16).

[Ber10] Daniel J. Bernstein. "Grover vs. McEliece". In: *Post-Quantum Cryptography*. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2010, pp. 73–80. ISBN: 978-3-642-12929-2. DOI: 10.1007/978-3-642-12929-2_6 (cit. on p. 35).

[BBD09] Daniel J. Bernstein, Johannes Buchmann, and Erik Dahmén, eds. *Post-quantum cryptography*. en. Berlin: Springer, 2009. ISBN: 978-3-540-88701-0 (cit. on pp. 32, 72).

[BCL+19] 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: conservative code-based cryptography - Round 2*. Tech. rep. Mar. 2019. URL: https://classic.mceliece.org/nist/mceliece-20190331.pdf (visited on 02/15/2023) (cit. on pp. 3, 20, 54).

[BCL+20] Daniel J. Bernstein, Tung Chou, Tanja Lange, Ingo von Maurich, Rafael Misoczki, Ruben Niederhagen, Edoardo Persichetti, Christiane Peters, Peter Schwabe, Nicolas Sendrier, Jakub Szefer, Wen Wang, Martin R. Albrecht, Carlos Cid, Jan Gilche, Varun Maram, Kenneth G. Paterson, Cen Jung Tjhai, and Martin Tomlinson. *Classic McEliece: conservative code-based cryptography - Round 3*. Tech. rep. Oct. 2020. URL: https://classic.mceliece.org/nist/mceliece-20201010.pdf (visited on 02/15/2023) (cit. on pp. 3, 19, 20, 23, 28, 31, 70, 71).

[BCL+22] Daniel J. Bernstein, Tung Chou, Tanja Lange, Ingo von Maurich, Rafael Misoczki, Ruben Niederhagen, Edoardo Persichetti, Christiane Peters, Peter Schwabe, Nicolas Sendrier, Jakub Szefer, Wen Wang, Martin R. Albrecht, Carlos Cid, Jan Gilche, Varun Maram, Kenneth G. Paterson, Cen Jung Tjhai, and Martin Tomlinson. *Classic McEliece: conservative code-based cryptography: modifications for round 4*. Tech. rep. Oct. 2022. URL: https://classic.mceliece.org/nist/mceliece-mods3-20221023.pdf (visited on 02/15/2023) (cit. on p. 30).

[BLP08] Daniel J. Bernstein, Tanja Lange, and Christiane Peters. "Attacking and Defending the McEliece Cryptosystem". en. In: *Post-Quantum Cryptography*. Vol. 5299. 2008, pp. 31–46. DOI: 10.1007/978-3-540-88403-3_3 (cit. on p. 33).

[BLP11] Daniel J. Bernstein, Tanja Lange, and Christiane Peters. "Smaller Decoding Exponents: Ball-Collision Decoding". In: *Advances in Cryptology – CRYPTO 2011*. Ed. by Phillip Rogaway. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2011, pp. 743–760. ISBN: 978-3-642-22792-9. DOI: 10.1007/978-3-642-22792-9_42 (cit. on p. 34).

[BS97] S.V. Bezzateev and N.A. Shekhunova. "One Generalization of Goppa Codes". In: *Proceedings of IEEE International Symposium on Information Theory*. June 1997, p. 299. DOI: 10.1109/ISIT.1997.613221 (cit. on p. 37).

[BS11] Sergey Bezzateev and N.A. Shekhunova. "One Class of Generalized Quasi-cyclic (L,G) Codes". In: 2011. URL: https://www.researchgate.net/publication/235632351_One_Class_of_Generalized_Quasi-cyclic_LG_Codes (visited on 01/05/2023) (cit. on pp. 37, 38).

[BS13] Sergey Bezzateev and N.A. Shekhunova. "Class of Binary Generalized Goppa Codes Perfect in Weighted Hamming Metric". In: *Designs Codes and Cryptography* 66 (Aug. 2013), pp. 391–399. DOI: 10.1007/s10623-012-9739-6 (cit. on p. 38).

[BR22] Mainak Bhattacharyya and Ankur Raina. "A quantum algorithm for syndrome decoding of classical error-correcting linear block codes". In: *2022 IEEE/ACM 7th Symposium on Edge Computing (SEC)*. Dec. 2022, pp. 456–461. DOI: 10.1109/SEC54971.2022.00069 (cit. on p. 17).

[BM18] Leif Both and Alexander May. "Decoding Linear Codes with High Error Rate and Its Impact for LPN Security". In: *Post-Quantum Cryptography*. Vol. 10786. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 25–46. DOI: 10.1007/978-3-319-79063-3_2 (cit. on p. 34).

[Buc01] Bruno Buchberger. "Gröbner Bases: A Short Introduction for Systems Theorists". en. In: *Computer Aided Systems Theory — EUROCAST 2001*. Vol. 2178. Springer Berlin Heidelberg, 2001, pp. 1–19. ISBN: 978-3-540-42959-3. DOI: 10.1007/3-540-45654-6_1 (cit. on pp. 99, 101).

[CC98] A. Canteaut and F. Chabaud. “A new algorithm for finding minimum-weight words in a linear code: application to McEliece’s cryptosystem and to narrow-sense BCH codes of length 511”. In: *IEEE Transactions on Information Theory* 44.1 (Jan. 1998), pp. 367–378. ISSN: 1557-9654. DOI: 10.1109/18.651067 (cit. on p. 33).

[CC94] Anne Canteaut and Hervé Chabanne. *A further improvement of the work factor in an attempt at breaking McEliece’s cryptosystem*. en. Research Report RR-2227. INRIA, 1994. URL: https://hal.inria.fr/inria-00074443 (cit. on p. 33).

[CS98] Anne Canteaut and Nicolas Sendrier. “Cryptanalysis of the Original McEliece Cryptosystem”. In: *Advances in Cryptology — ASIACRYPT ’98*. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1998, pp. 187–199. ISBN: 978-3-540-49649-6. DOI: 10.1007/3-540-49649-1_16 (cit. on p. 33).

[CD22] Wouter Castryck and Thomas Decru. *An efficient key recovery attack on SIDH*. Report Number: 975. 2022. URL: https://eprint.iacr.org/2022/975 (cit. on p. 3).

[CCD+21] Pierre-Louis Cayrel, Brice Colombier, Vlad-Florin Drăgoi, Alexandre Menu, and Lilian Bossuet. “Message-Recovery Laser Fault Injection Attack on the Classic McEliece Cryptosystem”. In: *Advances in Cryptology – EUROCRYPT 2021*. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2021, pp. 438–467. ISBN: 978-3-030-77886-6. DOI: 10.1007/978-3-030-77886-6_15 (cit. on p. 69).

[CDE21] André Chailloux, Thomas Debris-Alazard, and Simona Etinski. “Classical and Quantum Algorithms for Generic Syndrome Decoding Problems and Applications to the Lee Metric”. In: *Post-Quantum Cryptography*. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2021, pp. 44–62. DOI: 10.1007/978-3-030-81293-5_3 (cit. on p. 17).

[CB13] I. V. Chizhov and M. A. Borodin. *The failure of McEliece PKC based on Reed-Muller codes*. Report Number: 287. 2013. URL: https://eprint.iacr.org/2013/287 (cit. on p. 31).

[CDCG22] Brice Colombier, Vlad-Florin Dragoi, Pierre-Louis Cayrel, and Vincent Grosso. *Message-recovery Profiled Side-channel Attack on the Classic McEliece Cryptosystem*. Cryptology ePrint Archive, Paper 2022/125. 2022. URL: https://eprint.iacr.org/2022/125 (cit. on p. 69).

[CMP17] Alain Couvreur, Irene Márquez-Corbella, and Ruud Pellikaan. “Cryptanalysis of McEliece Cryptosystem Based on Algebraic Geometry Codes and Their Subcodes”. In: *IEEE Transactions on Information Theory* 63.8 (Aug. 2017), pp. 5404–5418. ISSN: 1557-9654. DOI: 10.1109/TIT.2017.2712636 (cit. on p. 31).

[COT17] Alain Couvreur, Ayoub Otmani, and Jean-Pierre Tillich. "Polynomial Time Attack on Wild McEliece Over Quadratic Extensions". In: *IEEE Transactions on Information Theory* 63.1 (Jan. 2017). Conference Name: IEEE Transactions on Information Theory, pp. 404–427. ISSN: 1557-9654. DOI: 10.1109/TIT.2016.2574841 (cit. on pp. 31, 49).

[DK20] Julian Danner and Martin Kreuzer. "A Fault Attack on the Niederreiter Cryptosystem using Binary Irreducible Goppa Codes". In: *Journal of Groups, Complexity, Cryptology* 12.1 (Mar. 2020), 2:1–2:20. DOI: 10.46298/jgcc.2020.12.1.6074. URL: https://arxiv.org/abs/2002.01455 (cit. on pp. 70, 75, 79).

[DCR+17] Pasquale Davide Schiavone, Francesco Conti, Davide Rossi, Michael Gautschi, Antonio Pullini, Eric Flamand, and Luca Benini. "Slow and steady wins the race? A comparison of ultra-low-power RISC-V cores for Internet-of-Things applications". In: *International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*. Vol. 27. 2017, pp. 1–8. DOI: 10.1109/PATMOS.2017.8106976 (cit. on p. 85).

[Dum91] Ilya Dumer. "On minimum distance decoding of linear codes". In: vol. 1. 1991, pp. 50–52. URL: https://www.researchgate.net/publication/296573348_On_minimum_distance_decoding_of_linear_codes#citations (cit. on p. 34).

[DBN+01] Morris J. Dworkin, Elaine B. Barker, James R. Nechvatal, James Foti, Lawrence E. Bassham, E. Roback, and James F. Dray Jr. *Advanced Encryption Standard (AES)*. Standard 197. NIST, Nov. 2001. URL: https://www.nist.gov/publications/advanced-encryption-standard-aes (cit. on p. 1).

[EOS07] D. Engelbert, R. Overbeck, and A. Schmidt. "A Summary of McEliece-Type Cryptosystems and their Security". en. In: *Journal of Mathematical Cryptology* 1.2 (Apr. 2007), pp. 151–199. ISSN: 1862-2984, 1862-2976. DOI: 10.1515/JMC.2007.009. URL: https://www.degruyter.com/view/journals/jmc/1/2/article-p151.xml (visited on 06/15/2020) (cit. on p. 33).

[EB22] Andre Esser and Emanuele Bellini. "Syndrome Decoding Estimator". In: *Public-Key Cryptography – PKC 2022*. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2022, pp. 112–141. DOI: 10.1007/978-3-030-97121-2_5 (cit. on pp. 34, 35).

[EMZ22] Andre Esser, Alexander May, and Floyd Zweydinger. "McEliece Needs a Break – Solving McEliece-1284 and Quasi-Cyclic-2918 with Modern ISD". en. In: *Advances in Cryptology – EUROCRYPT 2022*. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2022, pp. 433–457. ISBN: 978-3-031-07082-2. DOI: 10.1007/978-3-031-07082-2_16 (cit. on p. 27).

[FGO+11] Jean-Charles Faugère, Valérie Gauthier-Umanã, Ayoub Otmani, Ludovic Perret, and Jean-Pierre Tillich. “A distinguisher for high rate McEliece cryptosystems”. In: *2011 IEEE Information Theory Workshop*. Oct. 2011, pp. 282–286. DOI: 10.1109/ITW.2011.6089437 (cit. on p. 31).

[FM08] Cedric Faure and Lorenz Minder. “Cryptanalysis of the McEliece cryptosystem over hyperelliptic codes”. In: *11th International Workshop on Algebraic and Combinatorial Coding Theory*. 2008, pp. 99–107. URL: http://www.moi.math.bas.bg/acct2008/b17.pdf (visited on 02/26/2023) (cit. on p. 31).

[FO13] Eiichiro Fujisaki and Tatsuaki Okamoto. “Secure Integration of Asymmetric and Symmetric Encryption Schemes”. en. In: *Journal of Cryptology* 26.1 (Jan. 2013), pp. 80–101. ISSN: 1432-1378. DOI: 10.1007/s00145-011-9114-1 (cit. on p. 29).

[Gab95] Ernst Gabidulin. “Public-Key Cryptosystems Based on Linear Codes”. In: *Faculty of Technical Mathematics and Informatics, Delft University of Technology*. Report 95-30 (1995). ISSN: 0922-5641. URL: https://www.researchgate.net/publication/2359950_Public-Key_Cryptosystems_Based_on_Linear_Codes (visited on 02/07/2023) (cit. on p. 31).

[GST+17] Michael Gautschi, Pasquale Davide Schiavone, Andreas Traber, Igor Loi, Antonio Pullini, Davide Rossi, Eric Flamand, Frank K. Gürkaynak, and Luca Benini. “Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25.10 (2017), pp. 2700–2713. DOI: 10.1109/TVLSI.2017.2654506 (cit. on p. 85).

[GM23] Johannes Geier and Daniel Mueller-Gritschneder. “vRTLmod: An LLVM based Open-source Tool to Enable Fault Injection in Verilator RTL Simulations”. In: *Proceedings of the 20th ACM International Conference on Computing Frontiers*. CF ’23. Association for Computing Machinery, 2023, pp. 387–388. DOI: 10.1145/3587135.3591435. URL: https://github.com/tum-ei-eda/vrtlmod (visited on 09/01/2023) (cit. on pp. 70, 86).

[Gib96] Keith Gibson. “The Security of the Gabidulin Public Key Cryptosystem”. In: *Advances in Cryptology — EUROCRYPT ’96*. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1996, pp. 212–223. ISBN: 978-3-540-68339-1. DOI: 10.1007/3-540-68339-9_19 (cit. on p. 31).

[Gop70] Valerii Denisovich Goppa. “A New Class of Linear Correcting Codes”. ru. In: *Problemy Peredachi Informatsii* 6.3 (1970), pp. 24–30. URL: https://www.mathnet.ru/eng/ppi1748 (visited on 01/02/2023) (cit. on pp. 5, 10, 11).

[Gop71] Valerii Denisovich Goppa. "A Rational Representation of Codes and (L, g)-Codes". ru. In: *Problemy Peredachi Informatsii* 7.3 (1971), pp. 41–49. URL: https://www.mathnet.ru/eng/ppi1647 (cit. on p. 11).

[GZ61] Daniel Gorenstein and Neal Zierler. "A Class of Error-Correcting Codes in p^m Symbols". In: *Journal of the Society for Industrial and Applied Mathematics* 9.2 (1961), pp. 207–214. ISSN: 0368-4245. DOI: 10.1137/0109020 (cit. on p. 12).

[GLRS16] Markus Grassl, Brandon Langenberg, Martin Roetteler, and Rainer Steinwandt. "Applying Grover's Algorithm to AES: Quantum Resource Estimates". en. In: *Post-Quantum Cryptography*. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2016, pp. 29–43. ISBN: 978-3-319-29360-8. DOI: 10.1007/978-3-319-29360-8_3 (cit. on p. 2).

[Gro96] Lov K. Grover. "A fast quantum mechanical algorithm for database search". In: *Proceedings of the 28th annual ACM symposium on Theory of Computing*. STOC '96. New York, NY, USA: Association for Computing Machinery, July 1996, pp. 212–219. ISBN: 978-0-89791-785-8. DOI: 10.1145/237814.237866 (cit. on p. 2).

[Gro97] Lov K. Grover. "Quantum Mechanics helps in searching for a needle in a haystack". In: *Physical Review Letters* 79.2 (July 1997). arXiv:quant-ph/9706033, pp. 325–328. ISSN: 0031-9007, 1079-7114. DOI: 10.1103/PhysRevLett.79.325. URL: http://arxiv.org/abs/quant-ph/9706033 (cit. on p. 2).

[GJJ22] Qian Guo, Andreas Johansson, and Thomas Johansson. *A Key-Recovery Side-Channel Attack on Classic McEliece*. Cryptology ePrint Archive, Paper 2022/514. 2022. URL: https://eprint.iacr.org/2022/514 (cit. on p. 69).

[HJ10] Nick Howgrave-Graham and Antoine Joux. "New Generic Algorithms for Hard Knapsacks". In: *Advances in Cryptology – EUROCRYPT 2010*. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2010, pp. 235–256. ISBN: 978-3-642-13190-5. DOI: 10.1007/978-3-642-13190-5_12 (cit. on p. 34).

[HP03] W. Cary Huffman and Vera Pless. *Fundamentals of Error-Correcting Codes*. Cambridge: Cambridge University Press, 2003. ISBN: 978-0-521-78280-7. DOI: 10.1017/CBO9780511807077 (cit. on pp. 5, 14).

[ISO16] ISO/IEC 15946-1:2016. *Information technology — Security techniques — Cryptographic techniques based on elliptic curves — Part 1: General*. Standard. July 2016, p. 31. URL: https://www.iso.org/standard/65480.html (cit. on p. 2).

[ISO22] ISO/IEC 15946-5:2022. *Information technology — Security techniques — Cryptographic techniques based on elliptic curves — Part 5: Elliptic curve generation*. en. Standard ISO/IEC 15946-5:2022. ISO/IEC JTC 1/SC 27, International Organization for Standardization / International Electrotechnical Commission. Feb. 2022, p. 35. URL: https://www.iso.org/standard/80241.html (cit. on p. 2).

[ISO06] ISO/IEC 18033-2:2006. *Information technology — Security techniques — Encryption algorithms — Part 2: Asymmetric ciphers*. Standard. May 2006, p. 125. URL: https://www.iso.org/standard/37971.html (cit. on p. 2).

[ISO10] ISO/IEC 18033-3:2010. *Information technology — Security techniques — Encryption algorithms — Part 3: Block ciphers*. Standard. Dec. 2010, p. 78. URL: https://www.iso.org/standard/54531.html (cit. on p. 1).

[ISO] ISO/IEC JTC 1/SC 27. *Information security, cybersecurity and privacy protection*. URL: https://www.iso.org/committee/45306.html (cit. on p. 1).

[JM96] Heeralal Janwa and Oscar Moreno. "McEliece Public Key Cryptosystems Using Algebraic-Geometric Codes". In: *Designs, Codes and Cryptography* 8.3 (June 1996), pp. 293–307. ISSN: 1573-7586. DOI: 10.1023/A:1027351723034 (cit. on p. 31).

[KL21] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. en. 3rd. CRC Press of Taylor and Francis Group, 2021. ISBN: 978-1-351-13303-6 (cit. on p. 28).

[KM22] Elena Kirshanova and Alexander May. *Decoding McEliece with a Hint - Secret Goppa Key Parts Reveal Everything*. Cryptology ePrint Archive, Paper 2022/525. 2022. URL: https://eprint.iacr.org/2022/525 (cit. on p. 81).

[LB88] P. J. Lee and E. F. Brickell. "An Observation on the Security of McEliece's Public-Key Cryptosystem". In: *Advances in Cryptology — EUROCRYPT '88*. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1988, pp. 275–280. ISBN: 978-3-540-45961-3. DOI: 10.1007/3-540-45961-8_25 (cit. on p. 33).

[LDW94] Yuan Xing Li, R.H. Deng, and Xin Mei Wang. "On the equivalence of McEliece's and Niederreiter's public-key cryptosystems". In: *IEEE Transactions on Information Theory* 40.1 (Jan. 1994). Conference Name: IEEE Transactions on Information Theory, pp. 271–273. ISSN: 1557-9654. DOI: 10.1109/18.272496 (cit. on p. 20).

[LN97] Rudolf Lidl and Harald Niederreiter. *Finite Fields*. 2nd ed. Encyclopedia of Mathematics and its Applications. Cambridge: Cambridge University Press, 1997. ISBN: 0-521-39231-4. DOI: 10.1017/CBO9780511525926 (cit. on pp. 5–8).

[LPZW21] Hedongliang Liu, Sabine Pircher, Alexander Zeh, and Antonia Wachter-Zeh. “Decoding of (Interleaved) Generalized Goppa Codes”. In: *2021 IEEE International Symposium on Information Theory (ISIT)*. 2021, pp. 664–669. DOI: 10.1109/ISIT45174.2021.9517785. URL: http://arxiv.org/abs/2102.02831 (cit. on pp. 37, 48).

[MS83] F.J. MacWilliams and N.J.A. Sloane. *The Theory of Error-Correcting Codes*. 1st. Vol. 16. North-Holland, 1983. ISBN: 978-0-444-85193-2 (cit. on pp. 5, 10, 11).

[Mas69a] J. Massey. “Shift-register synthesis and BCH decoding”. en. In: *IEEE Transactions on Information Theory* 15.1 (Jan. 1969), pp. 122–127. ISSN: 0018-9448. DOI: 10.1109/TIT.1969.1054260. URL: http://ieeexplore.ieee.org/document/1054260/ (cit. on p. 12).

[Mas69b] James L. Massey. “Shift-register synthesis and BCH decoding”. In: *IEEE Transactions on Information Theory* 15.1 (Jan. 1969), pp. 122–127. ISSN: 0018-9448. DOI: 10.1109/TIT.1969.1054260 (cit. on p. 26).

[MMT11] Alexander May, Alexander Meurer, and Enrico Thomae. “Decoding Random Linear Codes in $\tilde{\mathcal{O}}(2^{0.054n})$”. In: *Advances in Cryptology – ASIACRYPT 2011*. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2011, pp. 107–124. ISBN: 978-3-642-25385-0. DOI: 10.1007/978-3-642-25385-0_6 (cit. on p. 34).

[MO15] Alexander May and Ilya Ozerov. “On Computing Nearest Neighbors with Applications to Decoding of Binary Linear Codes”. In: *Advances in Cryptology – EUROCRYPT 2015*. Vol. 9056. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 203–228. DOI: 10.1007/978-3-662-46800-5_9 (cit. on p. 34).

[McE78] R. J. McEliece. “A Public-Key Cryptosystem Based On Algebraic Coding Theory”. In: *Deep Space Network Progress Report* 44 (Jan. 1978), pp. 114–116 (cit. on pp. 3, 19, 20, 32).

[MAB+21] Carlos Aguilar Melchor, Nicolas Aragon, Slim Bettaieb, Loic Bidoux, Jurjen Bos, Jean-Christophe Deneuville, Arnaud Dion, Philippe Gaborit, Jerome Lacan, Edoardo Persichetti, Jean-Marc Robert, Pascal Veron, and Gilles Zemor. “Hamming Quasi-Cyclic (HQC) Third round version Updated version 06/06/2021”. en. In: (2021), p. 48. URL: https://pqc-hqc.org/doc/hqc-specification_2021-06-06.pdf (cit. on p. 3).

[MS07] Lorenz Minder and Amin Shokrollahi. “Cryptanalysis of the Sidelnikov Cryptosystem”. In: *Advances in Cryptology - EUROCRYPT 2007*. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2007, pp. 347–360. ISBN: 978-3-540-72540-4. DOI: 10.1007/978-3-540-72540-4_20 (cit. on p. 31).

[MIN20] MINRES Technologies GmbH. *DBT-RISE-RISCV*. 2020. URL: https://github.com/Minres/DBT-RISE-RISCV (visited on 08/04/2023) (cit. on p. 66).

[MB09] Rafael Misoczki and Paulo S. L. M. Barreto. "Compact McEliece Keys from Goppa Codes". In: *Selected Areas in Cryptography*. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2009, pp. 376–392. ISBN: 978-3-642-05445-7. DOI: 10.1007/978-3-642-05445-7_24 (cit. on p. 31).

[MKJR16] Kathleen Moriarty, Burt Kaliski, Jakob Jonsson, and Aneas Rusch. *PKCS #1: RSA Cryptography Specifications Version 2.2*. Request for Comments RFC 8017. ISSN: 2070-1721. Internet Engineering Task Force (IETF)), Nov. 2016, p. 78. URL: https://doi.org/10.17487/RFC8017 (cit. on p. 2).

[MDG+17] Daniel Mueller-Gritschneder, Martin Dittrich, Marc Greim, Keerthikumara Devarajegowda, Wolfgang Ecker, and Ulf Schlichtmann. "The Extendable Translating Instruction Set Simulator (ETISS) Interlinked with an MDA Framework for Fast RISC Prototyping". In: *2017 International Symposium on Rapid System Prototyping (RSP)*. IEEE, Oct. 2017, pp. 79–84. ISBN: 978-1-4503-5418-9. URL: https://ieeexplore.ieee.org/document/8547814 (cit. on pp. 54, 66).

[Nat16] National Institute for Standards and Technology. *Submission Requirements and Evaluation Criteria for the Post-Quantum Cryptography Standardization Process*. Tech. rep. Dec. 2016. URL: https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf (visited on 02/05/2023) (cit. on pp. 19, 27, 28).

[Nat17] National Institute for Standards and Technology. *Post-Quantum Cryptography Standardization*. Jan. 2017. URL: https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization (visited on 09/19/2022) (cit. on p. 2).

[Nat15] National Institute of Standards. *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*. en. Tech. rep. Federal Information Processing Standard (FIPS) 202. U.S. Department of Commerce, Aug. 2015. DOI: 10.6028/NIST.FIPS.202. URL: https://csrc.nist.gov/publications/detail/fips/202/final (visited on 09/16/2022) (cit. on p. 23).

[Nie86] Harald Niederreiter. "Knapsack-type Cryptosystems and Algebraic Coding Theory". In: *Problems of Control and Information Theory* 15(2) (1986), pp. 159–166 (cit. on pp. 3, 19, 20, 22, 31).

[NB20a] I. K. Noskov and S. V. Bezzateev. "One Realization of the Generalized (L, G)-Codes". In: *2020 Wave Electronics and its Application in Information and Telecommunication Systems (WECONF)*. 2020, pp. 1–5. DOI: 10.1109/WECONF48837.2020.9131441 (cit. on p. 39).

[NB20b] I.K. Noskov and Sergey Bezzateev. "Effective implementation of modern McEliece cryptosystem on generalized (L,G)-codes (Russian Only)". ru. In: *Scientific and Technical Journal of Information Technologies, Mechanics and Optics* 20 (Aug. 2020), pp. 539–544. DOI: 10.17586/2226-1494-2020-20-4-539-544 (cit. on pp. 40, 42).

[Opea] OpenHW Group. *CV32E40P - GitHub*. URL: https://github.com/openhwgroup/cv32e40p (visited on 08/25/2022) (cit. on pp. 85, 86).

[Opeb] OpenHW Group. *CV32E40S - GitHub*. URL: https://github.com/openhwgroup/cv32e40s (visited on 08/25/2022) (cit. on pp. 85, 86).

[OTD10] Ayoub Otmani, Jean-Pierre Tillich, and Léonard Dallot. "Cryptanalysis of Two McEliece Cryptosystems Based on Quasi-Cyclic Codes". In: *Mathematics in Computer Science* 3.2 (Apr. 2010), pp. 129–140. ISSN: 1661-8289. DOI: 10.1007/s11786-009-0015-8 (cit. on p. 31).

[Ove08] R. Overbeck. "Structural Attacks for Public Key Cryptosystems based on Gabidulin Codes". In: *Journal of Cryptology* 21.2 (Apr. 2008), pp. 280–301. ISSN: 1432-1378. DOI: 10.1007/s00145-007-9003-9. URL: https://doi.org/10.1007/s00145-007-9003-9 (cit. on p. 31).

[Pat75] N. Patterson. "The algebraic decoding of Goppa codes". In: *IEEE Transactions on Information Theory* 21.2 (Mar. 1975), pp. 203–207. ISSN: 1557-9654. DOI: 10.1109/TIT.1975.1055350 (cit. on p. 12).

[Pet60] W. Peterson. "Encoding and error-correction procedures for the Bose-Chaudhuri codes". In: *IRE Transactions on Information Theory* 6.4 (Sept. 1960). Conference Name: IRE Transactions on Information Theory, pp. 459–470. ISSN: 2168-2712. DOI: 10.1109/TIT.1960.1057586 (cit. on p. 12).

[PGZM21] S. Pircher, J. Geier, A. Zeh, and D. Mueller-Gritschneder. "Exploring the RISC-V Vector Extension for the Classic McEliece Post-Quantum Cryptosystem". In: *2021 22nd International Symposium on Quality Electronic Design (ISQED)*. ISSN: 1948-3287. Apr. 2021, pp. 401–407. DOI: 10.1109/ISQED51717.2021.9424273 (cit. on pp. 54, 67).

[PGD+23] Sabine Pircher, Johannes Geier, Julian Danner, Daniel Mueller-Gritschneder, and Antonia Wachter-Zeh. "Key-Recovery Fault Injection Attack on the Classic McEliece KEM". In: *Code-Based Cryptography*. Lecture Notes in Computer Science. Springer Nature Switzerland, 2023, pp. 37–61. ISBN: 978-3-031-29689-5. DOI: 10.1007/978-3-031-29689-5_3 (cit. on pp. 70, 81).

[PGMW22] Sabine Pircher, Johannes Geier, Daniel Mueller-Gritschneder, and Antonia Wachter-Zeh. *Key-Recovery Fault Injection Attack on the Classic McEliece KEM.* Conference. Trondheim, Norway, May 2022. URL: https://www.cb-crypto.org/previous-editions/cbcrypto2022/program (visited on 09/02/2023) (cit. on p. 70).

[PQC20] PQC Forum. *ROUND 3 OFFICIAL COMMENT: Classic McEliece.* Nov. 2020. URL: https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/EiwxGnfQgec (visited on 02/14/2023) (cit. on p. 28).

[PQC21] PQC Forum. *Security strength categories for Code Based Crypto (And trying out Crypto Stack Exchange).* July 2021. URL: https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/6XbG66gI7v0 (visited on 02/14/2023) (cit. on p. 28).

[Pra62] E. Prange. "The use of information sets in decoding cyclic codes". In: *IRE Transactions on Information Theory* 8.5 (Sept. 1962), pp. 5–9. ISSN: 2168-2712. DOI: 10.1109/TIT.1962.1057777 (cit. on pp. 32, 33, 48, 92).

[PMIB17] Sven Puchinger, Sven Müelich, Karim Ishak, and Martin Bossert. "Code-Based Cryptosystems Using Generalized Concatenated Codes". In: *Applications of Computer Algebra.* Ed. by Ilias S. Kotsireas and Edgar Martínez-Moro. Springer Proceedings in Mathematics & Statistics. Cham: Springer International Publishing, 2017, pp. 397–423. ISBN: 978-3-319-56932-1. DOI: 10.1007/978-3-319-56932-1_26 (cit. on p. 31).

[RN87] T. R. N. Rao and Kil-Hyun Nam. "Private-Key Algebraic-Coded Cryptosystems". en. In: *Advances in Cryptology — CRYPTO' 86.* Ed. by Andrew M. Odlyzko. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1987, pp. 35–48. DOI: 10.1007/3-540-47721-7_3 (cit. on p. 32).

[Rot06] Ron M. Roth. *Introduction to coding theory.* en. OCLC: ocm61757112. Cambridge, UK ; New York: Cambridge University Press, 2006. ISBN: 978-0-521-84504-5. DOI: https://doi.org/10.1017/CBO9780511808968 (cit. on pp. 5, 7, 13–15, 39).

[Sag22] Sage Developers. *SageMath, the Sage Mathematics Software System (Version 9.5).* 2022. URL: https://www.sagemath.org (cit. on p. 82).

[SHS16] Bodo Selmke, Johann Heyszl, and Georg Sigl. "Attack on a DFA Protected AES by Simultaneous Laser Fault Injections". In: *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC).* Aug. 2016, pp. 36–46. DOI: 10.1109/FDTC.2016.16 (cit. on pp. 69, 71).

[Sen00] N. Sendrier. "Finding the permutation between equivalent linear codes: the support splitting algorithm". In: *IEEE Transactions on Information Theory* 46.4 (July 2000), pp. 1193–1203. ISSN: 1557-9654. DOI: 10.1109/18.850662 (cit. on p. 32).

[Sen95] Nicolas Sendrier. *On the Structure of Randomly Permuted Concatenated Code*. Tech. rep. RR-2460. INRIA, 1995. URL: https://hal.inria.fr/inria-00074216v1 (cit. on p. 31).

[Sen98] Nicolas Sendrier. "On the Concatenated Structure of a Linear Code". In: *Applicable Algebra in Engineering, Communication and Computing* 9.3 (Nov. 1998), pp. 221–242. ISSN: 1432-0622. DOI: 10.1007/s002000050104 (cit. on p. 31).

[Sen11] Nicolas Sendrier. "Decoding One Out of Many". In: *Post-Quantum Cryptography*. Vol. 7071. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2011, pp. 51–67. ISBN: 978-3-642-25405-5. DOI: 10.1007/978-3-642-25405-5_4 (cit. on p. 35).

[SM81] N. A. Shekhunova and E. T. Mironchikov. "Cyclic (L,G)-Codes". ru. In: *Problemy Peredachi Informatsii* 17 (1981), pp. 3–9 (cit. on p. 37).

[Sho94] P.W. Shor. "Algorithms for Quantum Computation: Discrete Logarithms and Factoring". In: *Proceedings 35th Annual Symposium on Foundations of Computer Science*. Nov. 1994, pp. 124–134. DOI: 10.1109/SFCS.1994.365700 (cit. on p. 2).

[Sid94] V. M. Sidelnikov. "A public-key cryptosystem based on binary Reed-Muller codes". In: *Discrete Mathematics and Applications* 4.3 (1994), pp. 191–207. ISSN: 1569-3929. DOI: 10.1515/dma.1994.4.3.191 (cit. on pp. 20, 31).

[SS92] V. M. Sidelnikov and S. O. Shestakov. "On insecurity of cryptosystems based on generalized Reed-Solomon codes". en. In: 2.4 (Jan. 1992). Publisher: De Gruyter Section: Discrete Mathematics and Applications, pp. 439–444. ISSN: 1569-3929. DOI: 10.1515/dma.1992.2.4.439. (Visited on 02/05/2023) (cit. on pp. 20, 31).

[Sny] Wilson Snyder. *Verilator*. URL: https://www.veripool.org/verilator/ (visited on 08/25/2022) (cit. on p. 86).

[Ste22] Stephan Ehlen. *Request for feedback on Classic McEliece*. Dec. 2022. URL: https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/jgevyeKehcM (visited on 02/14/2023) (cit. on p. 3).

[Ste89] Jacques Stern. "A method for finding codewords of small weight". In: *Coding Theory and Applications*. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1989, pp. 106–113. ISBN: 978-3-540-46726-7. DOI: 10.1007/BFb0019850 (cit. on p. 33).

[Str69] Volker Strassen. "Gaussian elimination is not optimal". In: *Numerische Mathematik* 13.4 (Aug. 1969), pp. 354–356. ISSN: 0945-3245. DOI: 10.1007/BF02165411. URL: https://doi.org/10.1007/BF02165411 (visited on 01/26/2023) (cit. on p. 33).

[Stu05] Bernd Sturmfels. "What is a Gröbner Basis?" In: *Americal Mathematical Society (AMS)* 52.10 (Nov. 2005), p. 2. URL: https://math.berkeley.edu/~bernd/what-is.pdf (cit. on p. 99).

[SKST75] Yasuo Sugiyama, Masao Kashara, Shigeichi Hirasawa, and Toshihiko Namekawa. "A Method for Solving Key Equation for Decoding Goppa Codes". In: *Information and Control* 27.1 (1975), pp. 87–99. DOI: 10.1016/S0019-9958(75)90090-X (cit. on p. 12).

[Til88] Johan van Tilburg. "On the McEliece Public-Key Cryptosystem". In: *Advances in Cryptology — CRYPTO' 88*. Ed. by Shafi Goldwasser. Lecture Notes in Computer Science. New York, NY: Springer, 1988, pp. 119–131. ISBN: 978-0-387-34799-8. DOI: 10.1007/0-387-34799-2_10 (cit. on p. 33).

[Wei] Josef Weidendorfer. *KCachegrind*. URL: https://kcachegrind.github.io/html/Home.html (visited on 03/07/2023) (cit. on p. 57).

[Wie10] Christian Wieschebrink. "Cryptanalysis of the Niederreiter Public Key Scheme Based on GRS Subcodes". In: *Post-Quantum Cryptography*. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2010, pp. 61–72. ISBN: 978-3-642-12929-2. DOI: 10.1007/978-3-642-12929-2_5 (cit. on p. 31).

[XIU+21] Keita Xagawa, Akira Ito, Rei Ueno, Junko Takahashi, and Naofumi Homma. "Fault-Injection Attacks Against NIST's Post-Quantum Cryptography Round 3 KEM Candidates". In: *Advances in Cryptology – ASIACRYPT 2021*. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2021, pp. 33–61. ISBN: 978-3-030-92075-3. DOI: 10.1007/978-3-030-92075-3_2 (cit. on pp. 69, 70).

[ZB19] F. Zaruba and L. Benini. "The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27.11 (Nov. 2019), pp. 2629–2640. ISSN: 1557-9999. DOI: 10.1109/TVLSI.2019.2926114 (cit. on pp. 66, 67).