University of Passau
Faculty of Computer Science and Mathematics

Chair of Computer Engineering
Prof. Dr. Stefan Katzenbeisser

Master's Thesis
in
Computer Science

A Comprehensive Comparison of Fuzzy Extractor Schemes Employing Different Error Correction Codes

Nico Mexis

Submission date:	2023-08-29
Presentation date:	2023-09-12
Compilation date:	2023-10-26
Supervisors:	Prof. Dr. Stefan Katzenbeisser
	Prof. Dr. Tolga Arul
Advisor:	Dr. Nikolaos Athanasios Anagnostopoulos

ERKLÄRUNG

Ich erkläre, dass ich die vorliegende Arbeit mit dem Titel „A Comprehensive Comparison of Fuzzy Extractor Schemes Employing Different Error Correction Codes“ selbstständig, ohne unzulässige Hilfe und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel verfasst habe und dass alle wörtlich oder sinngemäß übernommenen Stellen als solche kenntlich gemacht sind.

Mit der aktuell geltenden Fassung der Satzung der Universität Passau zur Sicherung guter wissenschaftlicher Praxis und für den Umgang mit wissenschaftlichem Fehlverhalten vom 31. Juli 2008 (vABlUP Seite 283) bin ich vertraut.

Ich erkläre mich mit einer Überprüfung der Arbeit unter Zuhilfenahme von Dienstleistungen Dritter (z. B. Anti-Plagiatssoftware) zur Gewährleistung der einwandfreien Kennzeichnung übernommener Ausführungen ohne Verletzung geistigen Eigentums an einem von anderen geschaffenen urheberrechtlich geschützten Werk oder von anderen stammenden wesentlichen wissenschaftlichen Erkenntnissen, Hypothesen, Lehren oder Forschungsansätzen einverstanden.

.....
(Name, Vorname)

Translation of German text (notice: Only the German text is legally binding)

I hereby confirm that I have composed the present scientific work entitled “A Comprehensive Comparison of Fuzzy Extractor Schemes Employing Different Error Correction Codes” independently without anybody else’s assistance and utilising no sources or resources other than those specified. I certify that any content adopted literally or in substance has been properly identified and attributed.

I have familiarised myself with the University of Passau’s most recent Guidelines for Good Scientific Practice and Scientific Misconduct Ramifications of 31 July 2008 (vABlUP page 283).

I declare my consent to the use of third-party services (*e.g.* anti-plagiarism software) for the examination of my work to verify the absence of impermissible representation of adopted content without adequate attribution, which would violate the intellectual property rights of others by claiming ownership of somebody else’s work, scientific findings, hypotheses, teachings or research approaches.

Supervisor contacts:

Prof. Dr. Stefan Katzenbeisser
Chair of Computer Engineering
University of Passau
Email: stefan.katzenbeisser@uni-passau.de
Web: https://www.fim.uni-passau.de/en/computer-engineering/

Prof. Dr. Tolga Arul
Chair of Computer Engineering
University of Passau
Email: tolga.arul@uni-passau.de
Web: https://www.fim.uni-passau.de/en/computer-engineering/

Acknowledgements

I would like to express my deepest gratitude to the following individuals for their invaluable support and contributions to the completion of my Master's thesis:

Academic Supervisors and Advisors:

- Prof. Dr. Stefan Katzenbeisser
- Prof. Dr. Tolga Arul
- Dr. Nikolaos Athanasios Anagnostopoulos

Your guidance, expertise and unwavering commitment to my academic journey have been instrumental in shaping the quality and direction of this thesis. Your insights and feedback have been invaluable and I am deeply grateful for your mentorship.

Software Expertise:

- Prof. Dr. John Abbott

I am particularly indebted to Prof. Dr. John Abbott for his introduction to the CoCoA-5 software, which played an integral role in shaping the methodology and outcomes of this thesis. His expertise and guidance in utilising this powerful tool were pivotal to the success of my research.

Editor and Proofreader:

- Nikolas Kirschstein

I would like to thank Nikolas Kirschstein for his meticulous editing and proofreading assistance. His attention to detail and constructive suggestions have significantly improved the clarity and coherence of this thesis.

Family, Friends, and Loved Ones:

- My parents, Daniela and Christoph Mexis
- My siblings, Luca and Lina Mexis
- My girlfriend, Sarah Westphal
- My best friend since childhood, Roman Koch
- All the other friends I made on my way

To my beloved family, friends and the wonderful people who have become an integral part of my journey, your unwavering support, encouragement and understanding have been a constant source of strength. Your belief in me has been my motivation and your presence in my life has made every challenge more manageable and every success more meaningful.

* * *

This work would not have been possible without the collective contributions of these remarkable individuals. I am profoundly grateful for their participation in this endeavour.

Thank you all for being an essential part of my academic and personal journey.

Nico Mexis

Abstract

This thesis deals with fuzzy extractors, security primitives often used in conjunction with Physical Unclonable Functions (PUFs). A fuzzy extractor works in two stages: The generation phase and the reproduction phase. In the generation phase, an Error Correction Code (ECC) is used to compute redundant bits for a given PUF response, which are then stored as helper data, and a key is extracted from the response. Then, in the reproduction phase, another (possibly noisy) PUF response can be used in conjunction with this helper data to extract the original key.

It is clear that the performance of the fuzzy extractor is strongly dependent on the underlying ECC. Therefore, a comparison of ECCs in the context of fuzzy extractors is essential in order to make them as suitable as possible for a given situation. It is important to note that due to the plethora of various PUFs with different characteristics, it is very unrealistic to propose a single metric by which the suitability of a given ECC can be measured.

First, we give a brief introduction to the topic, followed by a detailed description of the background of the ECCs and fuzzy extractors studied. Then, we summarise related work and describe an implementation of the ECCs under consideration. Finally, we carry out the actual comparison of the ECCs and the thesis concludes with a summary of the results and suggestions for future work.

Contents

1 Introduction

In an era of ever-expanding interconnected digital ecosystems, the security and integrity of sensitive data have become paramount concerns. As information systems continue to evolve, ensuring the robustness of cryptographic systems has become a critical endeavour. A fundamental challenge is to effectively protect secrets and cryptographic keys within the *Physical Unclonable Functions (PUFs)* integrated with Systems of Systems (SoS). According to Katzenbeisser and Schaller [1], a PUF is a function embedded in a physical object. It should also satisfy three (or preferably four) properties:

- *Robustness*: The same input x should always produce (almost) the same output y on the same object.
- *Unclonability*: The PUF should not be physically reproducible or clonable even by the manufacturer.
- *Unpredictability*: It should be impossible to predict the yet unknown output y from a new input x.
- *Tamper-Evidence*: The PUF changes its output behaviour when invasively attacked.

These properties make PUFs a promising basis for secure key generation and storage. However, realising their potential requires addressing the inherent imperfections and noise that can distort their output.

At the heart of mitigating these imperfections lies the innovative concept of *fuzzy extractors*. They act as a bridge between the inherently noisy PUF outputs and reliable cryptographic keys, ensuring the consistency and security of the generated keys. Today, there are several ways to accomplish the task of removing the noise from PUF responses. However, one of the most popular ways is to use *Error Correction Codes (ECCs)*. An ECC offers two procedures: The *encoding* procedure calculates redundant bits for a message and adds them to it (most often by concatenation). The *decoding* procedure then takes both the received (noisy) message and the (possibly also noisy) redundant bits and is able to reproduce the original message.

In this interplay between PUFs and fuzzy extractors, the choice of the underlying ECC emerges as a critical factor. The present thesis embarks on a comprehensive exploration of ECCs in the context of fuzzy extractor schemes, dissecting their efficacy, trade-offs, and suitability for PUF-based systems of systems.

This research examines the crucial role of fuzzy extractors in maintaining the integrity and reliability of cryptographic keys derived from PUF-based sources. A fuzzy extractor is essentially a mathematical construct that uses error correction techniques to compensate for

the inevitable discrepancies introduced by PUFs. By meticulously analysing and comparing different ECCs, this study aims to provide a deeper understanding of their impact on the overall security and performance of PUF-based systems, such as certain SoS in the era of Internet of Things (IoT), such as the one proposed by Mexis *et al.* [2, 3]. As a result, six classes of ECCs have been implemented and the code has been released as open-source on GitHub: `https://github.com/ThexXTURBOXx/ECC`.

The rest of the thesis is structured as follows: Section 1.1 lists and introduces some important notations for the remainder of the thesis. Then Chapter 2 defines different types of ECCs, enumerates some of their classes and provides further background on these codes and fuzzy extractors. Related work on ECC comparisons both in the context of PUFs and in general is summarised in Chapter 3. Building on this, Chapter 4 describes implementations of the examined codes, which are then evaluated using various metrics in Chapter 5. Finally, we draw conclusions and propose future work in Chapter 6.

1.1 Notation

Although most of the symbols are explained in the List of Symbols and most of the acronyms in the List of Abbreviations, a few notations should be introduced here.

First of all, 0 can be either the usual number 0 or a logical zero, usually typeset as `0`, or even a zero vector or matrix (vector or matrix consisting only of zeros). Similarly, $\mathbb{1}_{m\times n}$ denotes the matrix or vector of ones of size $m \times n$. The size can also be omitted if it is clear from the context. Furthermore, I_n denotes the unit matrix of size n, *i.e.*, a square zero matrix with ones on the main diagonal. Lastly, horizontal concatenation of matrices is either shown through block matrices, *e.g.* $\begin{bmatrix} I_4 & \mathbb{1}_{4\times 4} \end{bmatrix}$, or through a concatenation operator, *e.g.* $\left[I_4 \mid \mathbb{1}_{4\times 4} \right]$.

The *Hamming weight* wt denotes the amount of non-zero symbols within a matrix, vector, array or other list-like structure. Using the Hamming weight, the *Hamming distance* d between some a and b is defined as $d(a, b) = \mathrm{wt}(a - b)$ and intuitively describes the number of symbols which differ between the list-like structures a and b.

Vectors are not specially marked and are considered to be columnar matrices, *i.e.* matrices with a single column. The *scale product* or *dot product* of two vectors v_1, v_2 is denoted as $\langle v_1, v_2 \rangle$. However, it is important to mention that the *linear span* of elements $a_1, \ldots, a_n$ is also denoted as $\langle a_1, \ldots, a_n \rangle$. It should be clear from the context when each definition is being used.

Finally, this thesis assumes the standard definition of *algebraic rings*, without the multiplicative identity 1. If a ring R contains the 1, it is called *unitary*. Furthermore, it is called *commutative* if $a \cdot b = b \cdot a$ for all $a, b \in R$. Also, every element $a \in R$ has a *minimal polynomial* which is the unique monic polynomial of least degree $\mu(x)$ with $\mu(a) = 0$.

A special case of rings are modular rings. They are mostly denoted as $\mathrm{GF}(p)$, $\mathbb{F}_p$, $\mathbb{Z}_p$ or $\mathbb{Z}/p\mathbb{Z}$ and its elements are $\{0, \ldots, p-1\}$. Each of these elements represents all the numbers whose remainder modulo p is that number.

A *n-th root of unity* defines any element ζ which satisfies $\zeta^n = 1$. Furthermore, if $\zeta^k \neq 1$ for every $k < n$, it is a *primitive* n-th root of unity.

There exist many other modular rings which can also be defined as *quotient rings* of some polynomial rings: If $\mathbb{F}_p[\alpha]$ is a polynomial ring, then $f \in \mathbb{F}_p[\alpha]$ induces the quotient ring $P = \mathbb{F}_p[\alpha]/\langle f \rangle$ whose elements can be described through all the polynomials from $\mathbb{F}_p[\alpha]$ whose degree is less than $n = \deg(f)$. In some cases, namely if f is a *primitive polynomial*,

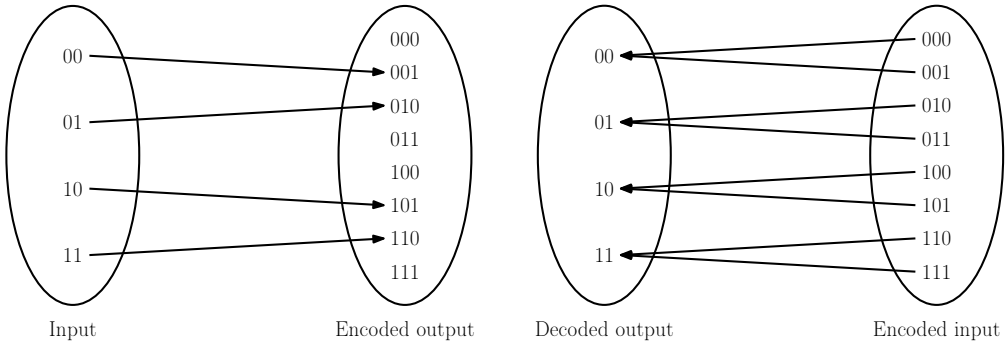
the quotient ring contains a *primitive element* α,[1] whose powers generate the whole ring, *i.e.*, $P = \left\{0, 1, \alpha, \alpha^2, \ldots, \alpha^{p^n-1}\right\}$. Thus, α is simultaneously a p^n-th root of unity in the *splitting field* $\mathbb{F}_p^n \cong \mathbb{F}_{p^n} \cong P$. The *Conway polynomial* is the lexicographically minimal primitive polynomial, which can be used to define a standard representation of $\mathbb{F}_{p^n}$ as a splitting field. There already exist some methods to find them, which are faster than brute force such as the approach presented by Heath and Loehr [4].

[1] The same α as the variable

2 Background

2.1 Error Correction Codes

There are various ways of explaining how ECCs work, some of which will be explained in detail in this and the following sections. From a linear algebra point of view, ECCs can be thought of as pairs of an injective and possibly non-surjective map from one set to another larger set, and a surjective, possibly noninjective "pseudoinverse" map. The first map is called encoding, the second decoding, and they are illustrated in Figure 2.1. It is important to note that in the context of bits, as in the case of fuzzy extractors, it is sufficient to consider only $\mathbb{F}_2$, $\mathbb{F}_{2^n}$, $\mathbb{F}_2^n$ and certain related sets, as these are sufficient to find simple representations for bits.



(a) Encoding procedure

(b) Decoding procedure

Figure 2.1: Encoding and decoding procedures of an exemplary ECC, adapted from Mexis [5].

Following this principle, the simplest codes to create are repetition codes [6].

Definition 1. The *b-repetition code* takes a word $a = [a_1, a_2, \ldots, a_m]$ and encodes it into a codeword by repeating each symbol a_i exactly b times, *i.e.*,

$$a \mapsto [\underbrace{a_1, \ldots, a_1}_{b \text{ times}}, a_2, \ldots, a_2, a_m, \ldots, a_m].$$

The decoding procedure uses majority voting on each of these m blocks to find the most likely original word.

The n-repetition code will now help us derive some general metrics to measure the quality and properties of any ECC. Three obvious metrics can be identified immediately.

Definition 2. Given any ECC, we derive three numbers from it [7]:

- *Block length* n: The amount of symbols an encoded codeword has (b-repetition code: $b \cdot m$)
- *Message length* or *dimension* k: The amount of symbols an input word has (b-repetition code: m)
- *Minimum (Hamming) distance* d: The minimum Hamming distance between two encoded codewords (b-repetition code: b)

Any ECC can be assigned such a triplet $[n, k, d]$.

Another important property can be derived from these.

Definition 3. The *code rate* R of an ECC C is given by $R \coloneqq \frac{\log_q(|C|)}{n}$ where $|C|$ is the amount of codewords of the ECC and q is the number of symbols in the alphabet of C. It is an indicator of how many redundant bits there are in a codeword [8]. Obviously, $R \in [0, 1]$, where 1 would indicate a perfect code and 0 would indicate the worst code. In the case of the b-repetition code, there are obviously q^m codewords, and thus we have $R = \frac{\log_q(q^m)}{b \cdot m} = \frac{m}{m \cdot b} = \frac{1}{b}$, indicating that the code rate gets worse as b increases.

An error correction procedure similar to a repetition code can be found in many modern systems used to transmit bits, such as the FlexRay communications system protocol [9]. This uses majority voting to filter out spike signals by holding the signal to be transmitted for a longer period of time.

In the context of ECCs, a very important distinction should be made. There are two broad classes, namely *convolutional codes* and *block codes*, where the former work by applying discrete convolutions of polynomial functions to encode a data stream, and the latter by encoding finite blocks of data using a single procedure. Although there are some papers that discuss the applicability of convolutional codes to fuzzy extractor schemes [10], the focus of this thesis will be on block codes. They will be referred to simply as "codes."

Again, block codes can be divided into many subcategories in many ways.

Definition 4. A *systematic code* is a code in which the original message can always be found unchanged at fixed coordinates within the corresponding encoded codewords [11]. If the code is not systematic, it is called a *non-systematic code* [12].

According to this definition, the b-repetition code is a systematic code, since if we extract every b-th symbol from a codeword, we obtain the corresponding original message. Obviously, when applying this definition, an error-free message should be assumed. Furthermore, permuting the symbols of a message does not change the systematicness of a code, since the message can still be found within the codeword; only its indices have changed. Lastly, the minimum distance of a code has a huge influence on the practicability of a code.

Lemma 1. *Let C be a $[n, k, d]$-code and $c \in C$ be any codeword.*

1. *If up to $d - 1$ errors are introduced in c, then the code C can correctly detect them.*
2. *If up to $\lfloor \frac{d-1}{2} \rfloor$ errors are introduced in c, then the code C can correctly correct them.*

Proof. We prove both properties separately:

1. Since d denotes the minimum Hamming distance between two codewords within C, any $d - 1$ symbols within a codeword $c \in C$ can be changed and the resulting word will no longer be a codeword.

2. Again, d denotes the minimum Hamming distance between two codewords within C. To correctly identify to which codeword a given erroneous word is nearest, its Hamming distance must obviously at most be $t := \lfloor \frac{d-1}{2} \rfloor$ to any codeword within C. This procedure is called *Minimum Distance Decoding (MDD)* [6] and can be thought of as drawing disjoint spheres, according to the Hamming distance metric, around each codeword within a multidimensional lattice, each with radius t, and determining the codeword in whose sphere the given word lies [13]. □

Many better classes of codes and corresponding decoding procedures are presented in the following subsections, some of which may "overlap." In most practical applications of block codes, general codes play a minor role. Therefore, a very special class of codes, which play the most important role, is defined in the following subsections.

2.1.1 Linear Codes

Before discussing the properties of linear codes in more detail, it is necessary to define them properly to be able to prove their peculiarities.

Definition 5. Let K be a finite field and $C \subseteq K^n$. Then C is said to be a *linear code* if C is a linear subspace of the vector space K^n.

As mentioned earlier, this definition allows us to use many different finite fields as the corresponding alphabet for the code. However, in the context of fuzzy extractor schemes, only $\mathbb{F}_2$ and $\mathbb{F}_{2^m}$ are sufficient to be used for the finite field K. Using this definition, we are able to redefine the properties of Definition 2.

Lemma 2. *Given a linear code $C \subseteq K^n$, we can derive* [14, 15]*:*

- Block length: *The n from K^n.*
- Dimension: $k = \dim(C)$*, which denotes the dimension of the subspace C, that is, the size of any basis of C.*
- Minimum distance: $d = \min_{c \in C \setminus \{0\}} \mathrm{wt}(c)$.
- Code rate: $R = \frac{k}{n}$.

Proof. The proofs by Roth [14] go as follows:

- Obvious since each codeword in C is also in K^n and thus, must be n symbols long.
- According to linear algebra, we have $|C| = q^{\dim(C)} = q^k$ and, thus, $k = \dim(C)$ where q is the number of elements of the finite field K.
- Since C is linear, for each $c_1, c_2 \in C$ we also have $c_1 - c_2 \in C$. According to Definition 2, we have
$$d = \min_{c_1 \neq c_2 \in C} d(c_1, c_2) = \min_{c_1 \neq c_2 \in C} \mathrm{wt}(c_1 - c_2) = \min_{c \in C \setminus \{0\}} \mathrm{wt}(c).$$
- Again, we have $|C| = q^k$ and therefore according to Definition 2, $R = \frac{\log_q(|C|)}{n} = \frac{k}{n}$. □

A very important property of linear codes is that they can be defined using a specific corresponding matrix.

Lemma 3. *Let $G \in K^{k \times n}$ be a matrix with k rows and n columns whose rows are exactly the elements of a basis of a block code C. This matrix is called the* generator matrix of C. *G fulfils the following properties* [16]*:*

1. *The row space of G is the linear code C.*
2. *A given word is a codeword if and only if it is a linear combination of the rows of G.*
3. $\mathrm{rk}(G) = k$.

Proof. 1. and 2. follow easily from Lemma 2, the definition of G and linear algebra. 3. is a direct result of the linear independence of the chosen basis of C. □

There is another complementary matrix that can also be used to uniquely define the same linear code C.

Lemma 4. *Let $H \in K^{(n-k) \times n}$ be a matrix with $n - k$ rows and n columns, which is the corresponding matrix for the homogeneous system of linear equations whose null space is exactly the set of codewords of C. This matrix is called the* parity-check matrix of C. *Then, H fulfils the following properties* [17]*:*

1. *$H \cdot c^T = 0$ if and only if $c \in C$.*
2. *$H \cdot G^T = 0$.*
3. $\mathrm{rk}(H) = n - k$.
4. $\dim(C) = n - \mathrm{rk}(H)$.

Proof. We prove each property separately:

1. This follows directly from the definition of the null space (sometimes also referred to as the *kernel*) of a matrix.
2. According to Lemma 3, each codeword is a linear combination of the rows of G. Additionally, from 1. we know $H \cdot c^T = 0$ and thus $H \cdot G^T = 0$.
3. The rows of H need to be linearly independent by definition, and thus, $rk(H) = n - k$.
4. From 3., we know that $\mathrm{rk}(H) = n - k$ and therefore $\dim(C) = k = n - (n - k) = n - \mathrm{rk}(H)$. □

Both matrices are very useful for the encoding and decoding of linear codes, respectively. The use of the parity-check matrix will be explained in detail later.

Theorem 5 (Encoding linear codes)**.** *When encoding a message w using the linear code C, the corresponding encoded codeword can be calculated by $c = w \cdot G$ where G is the generator matrix of C.*

Proof. Again, using basic linear algebra, G induces a linear map $\varphi\colon K^k \to C$. Because of $k = \dim(K^k) = \dim(C) < \infty$ and φ obviously being surjective, according to the rank–nullity theorem, φ is also bijective [18]. Thus, $C = \varphi(K^k) = \left\{w \cdot G \mid w \in K^k\right\}$. □

As mentioned earlier, there is an important distinction between systematic and non-systematic codes, which can also be made specifically in the context of linear codes.

Lemma 6. *For every linear code C, there is an equivalent[2] code C^* which is systematic. Its generator matrix is of the form $G^* = [I_k \mid P]$ and the corresponding parity-check matrix is of the form $H^* = \left[-P^T \mid I_{n-k}\right]$.*

[2] Equivalence means that it is isomorphic to C.

Proof. The generator matrix can be put into reduced row echelon form, which is exactly what is required [19]. Using the multiplication of block matrices, we then get

$$H^* \cdot G^{*T} = \left[-P^T \mid I_{n-k}\right] \cdot \left[I_k \mid P\right]^T = -P^T + P^T = 0,$$

which proves that G and H are orthogonal matrices and thus a pair of generator matrix and parity-check matrix [5]. □

Remark 7. It is clear that each codeword of a systematic code consists of two distinct areas: Unaltered message bits w_i and redundancy bits p_i:

$$c = [w_1, \ldots, w_k, p_1, \ldots, p_{n-k}].$$

For this reason, the value of $n - k$ is often referred to as the *redundancy* of the code C [20]. It should also be noted that sometimes the fraction $\frac{n}{k}$ is referred to as the redundancy of a code [12].

Now that we are able to encode messages using linear codes in an efficient way, it is a good time to think about decoding methods. MDD was already mentioned in the proof of Lemma 1. However, it is very slow because it has to calculate the Hamming distance for each possible codeword and therefore needs to know each possible codeword in advance. To find a better decoding method specifically for linear codes, some important notation need to be introduced first.

Definition 6. Let $C \subseteq \mathbb{F}_q^n$ be a linear code. The *coset* of C determined by $w \in \mathbb{F}_q^n$ is denoted by $w + C \coloneqq \{w + v \mid v \in C\}$.

Lemma 8. *Let $C \subseteq \mathbb{F}_q^n$ be a linear code and $u, v \in \mathbb{F}_q^n$. Then, the following properties hold:*

1. *$u \in u + C$ and $|u + C| = |C| = q^k$*
2. *$u \in v + C \implies u + C = v + C$*
3. *$u + C = v + C \iff u - v \in C$*
4. *Either $u + C = v + C$ or $(u + c) \cap (v + C) = \emptyset$*
5. *There are q^{n-k} different cosets of C*

Proof. We prove each property separately:

1. Since C is linear, $0 \in C$ and hence $u + 0 \in C$ and $u \in u + C$. Together with Lemma 2 and the fact that subspaces are closed under addition, we must have $|u+C| = |C| = q^k$.
2. We have $u = v + w$ for some $w \in C$ and thus $u + C = (v + w) + C = \{(v + w) + x \mid x \in C\} = \{v + (w + x) \mid x \in C\} = \{v + x \mid x \in C\} = v+C$. The penultimate equality follows again from the fact that subspaces are closed under addition.
3. According to 1., we have $u \in u + C = v + C \iff u = v + w$ for some $w \in C \iff u - v = (v + w) - v = w \in C$ for some $w \in C$.
4. According to MacWilliams and Sloane [11], we have two cases. The case $u + C \cap v + C = \emptyset$ is trivial. Otherwise, there is at least one element w in both cosets. Then $u + a = w = v + b$ with $a, b \in C$. Hence $v = u + a - b = u + a'$ with $a' \in C$ and thus $v + C \subseteq u + C$. Similarly, $u + C \subseteq v + C$ can be proven, and hence $u + C = v + C$.
5. Follows directly from 1. and 4. □

The other important definition inherently follows from these cosets.

Definition 7. The *syndrome* of $u \in \mathbb{F}_q^n$ with respect to a given parity-check matrix H for C is defined as $S_H(u) \coloneqq H \cdot u^T$.

Lemma 9. *Let $C \subseteq \mathbb{F}_q^n$ be a linear code with parity-check matrix H and $u, v \in \mathbb{F}_q^n$. Then, according to MacWilliams and Sloane [11] the following properties hold:*

1. $S_H(u+v) = S_H(u) + S_H(v)$
2. $S_H(u) = 0 \iff u \in C$
3. $S_H(u) = S_H(v) \iff u - v \in C \iff u + C = v + C$

Proof. We prove each property separately:

1. Since the set of all matrices forms an algebraic ring, we have $S_H(u+v) = H \cdot (u+v)^T = H \cdot u^T + H \cdot v^T = S_H(u) + S_H(v)$.
2. Already proven in Lemma 4.
3. According to 2., we have $u - v \in C \iff 0 = S_H(u-v) = H \cdot (u-v)^T = H \cdot u^T - H \cdot v^T = S_H(u) - S_H(v) \iff S_H(u) = S_H(v)$. The last equivalence was already proven in Lemma 8. □

Remark 10. The last property shows that there is a 1-1 correspondence between the syndromes and the cosets with respect to C.

There is one last definition before the decoding algorithm can be properly introduced.

Definition 8. A *Standard Decoding Array (SDA)* for a given linear code C is a table that matches each coset with a describing syndrome, called *coset leader*. It is defined as an element of minimum Hamming weight in that coset.

Theorem 11 (SDA construction)**.** *An SDA can be constructed using Algorithm 1.*

Algorithm 1 SDA Construction.

Require: A linear code $C \neq \emptyset$ over $\mathbb{F}_q$ with parity-check matrix H

Calculate all cosets for C	▷ Simple iteration over all elements of $\mathbb{F}_q$
Choose a coset leader for each coset, say u	▷ Must have minimum weight
$S_H(u) \leftarrow H \cdot u^T$	▷ H fixed parity-check matrix for C
Save all pairs $[S_H(u), u]$	▷ Using a hash table or similar

Proof. We need to prove two properties:

Finiteness: Only a finite amount of elements must be iterated, and the procedure will terminate after a finite amount of time.

Correctness: Follows from Definition 8. □

It is now time to introduce a decoding algorithm that still has a mediocre time complexity but at least exploits the peculiarities of linear codes.

Theorem 12 (Syndrome decoding)**.** *Algorithm 2 allows decoding of general linear codes.*

Algorithm 2 Syndrome Decoding.

Require: A linear code $C \neq \emptyset$ over $\mathbb{F}_q$ with parity-check matrix H, w received word

Construct an SDA for C	▷ Using Algorithm 1
$s \leftarrow S_H(w)$	
Find the coset leader e next to $s = S_H(e)$ in SDA	
Decode w to $v = w - e$	

Proof. We need to prove two properties:

Finiteness: Should be obvious since there is no possibility for an endless loop.

Correctness: From Definition 8, it is known that e must be of minimum weight. Also, since $S_H(w) = S_H(e)$, according to Lemma 9, we have $v = w - e \in C$. Overall, the result of syndrome decoding will be the same as the result of MDD. □

Remark 13. The construction of the SDA could also be done in advance so that it does not need to be calculated every time a new word must be decoded.

Remark 14. If $S_H(w) = 0$, the algorithm could be optimised by just returning w as according to Lemmata 8 and 9, we must already have $w \in C$.

Remark 15. It is important to note that despite being much more efficient than simple MDD as in Lemma 1, syndrome decoding is still very slow and even known to be $\mathcal{NP}$-complete [21]. Even more efficient (but non-general) decoders will be introduced in the following sections.

2.1.1.1 Hamming Codes

The theory of error-detecting codes has been known since at least the early 1940s. Bell Telephone Laboratories developed six relay calculators, named Models I to VI, where Models II to VI used a so-called two-out-of-five code (or a three-out-of-five code). It consists of a data block representing the numbers 0 to 4, and another block that simultaneously represents the number 5 and acts as a parity bit. In summary, every decimal digit can be represented by setting a single bit in the former and latter blocks, each. In the case of the three-out-of-five code, there are always three set bits in the message. If an error occurs, there will be either no or two bits set in one of these blocks. In this way, a single error can be detected [22, 23].

In 1948, Hamming was able to enhance these types of codes in an attempt to create the first ECC, capable of correcting a single error and detecting up to two errors. Nevertheless, Shannon [24] was the first to publish the code, crediting Hamming as its inventor. It was only two years after Shannon's work that Hamming himself published his results. Nowadays, these codes are known as *Hamming codes* [12, 25, 26].

There are several approaches to coding and decoding Hamming codes. However, we will first focus on a more visual approach that has been published by McEliece [27]: In order to be able to reconstruct erroneous data, there should be some mechanism that can detect at which location the error occurred and what the error looks like. For example, in the ternary case, the error could be a digit that changed by 1 or 2. But, as already said, this work is more concerned with binary codes where the only error is a bit flip, that is, a change of exactly 1. Because of this, only the error positions need to be determined. The encoding can be illustrated using Venn diagrams as shown in Figure 2.2 and goes as follows:

1. Insert the message bits into the intersections of the sets, padding if necessary.
2. In each remaining part, write the sum over $\mathbb{F}_2$ (since we are only considering binary Hamming codes) of all the corresponding intersecting entries.

The three calculated numbers are the redundant parity-check bits that are being used to locate and correct errors. Clearly, the four other numbers are the original message bits. The order of the bits does not matter in this case, of course. Now suppose that in the message from Figure 2.2, an error has occurred. The decoding procedure using three different examples is illustrated in Figure 2.3 and goes as follows:

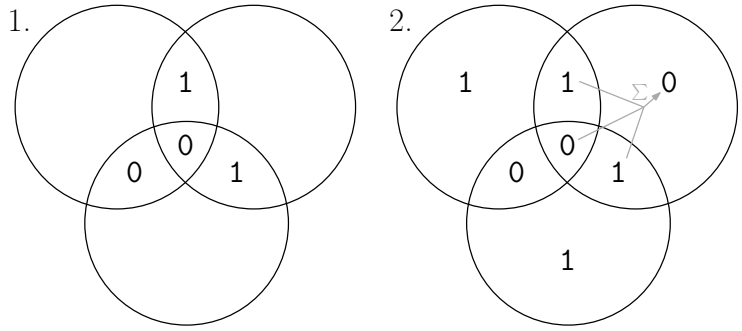


Figure 2.2: Encoding procedure of Hamming codes using Venn diagrams, adapted from McEliece [27].

1. For each set, check if the sum of the intersected parts matches the number in the non-intersected part.
2. If all matches, no error has occurred. If not, find the unique area where all the sets with wrong sum and none of the ones with right sum intersect. This bit needs to be flipped.

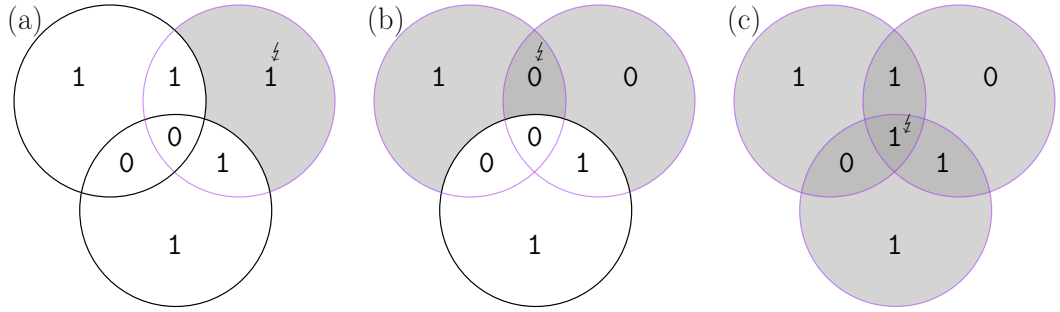


Figure 2.3: Decoding procedure of Hamming codes using Venn diagrams, adapted from McEliece [27].

This approach can also be applied to cases where the Venn diagram consists of four or more overlapping areas. However, due to the limitations of planar graphs, and therefore Venn diagrams, they must be drawn using other shapes, such as ellipses [28].

However, in order to provide an efficient way to encode and decode messages, an algorithmic approach is required. To proceed, the following fact is essential.

Remark 16. A Venn diagram for the n sets is divided into 2^n different areas. This follows from the fact that the power set $\mathcal{P}(\{1, \ldots, n\})$ has exactly 2^n elements. Here, the "outer face" (*i.e.*, the area around the Venn diagram) is also considered an area.

Using this fact, it is obvious that the Venn diagram approach can be converted into a table with $2^n - 1$ entries. In Table 2.1, each bit position is converted to binary, from which the parity coverage can be derived. The bit positions with only one parity coverage entry set are, of course, the parity bits themselves. The other ones with p parity coverage entries set correspond to the areas with p intersections in the corresponding Venn diagram. For example, in Figures 2.2 and 2.3 the area in the centre corresponds to w_7.

Table 2.1 helps to devise a "school" and a "computer" approach to coding and decoding Hamming codes in general. The "computer" approach is explained in Section 4.2.1.

Table 2.1: Hamming code construction using the table approach, adapted from Viswanathan [29].

Decimal bit position	1	2	3	4	5	6	7
Binary bit position	001	010	011	100	101	110	111
Bit name	p_1	p_2	w_3	p_4	w_5	w_6	w_7
Parity coverage (p_1)	✓		✓		✓		✓
Parity coverage (p_2)		✓	✓			✓	✓
Parity coverage (p_4)				✓	✓	✓	✓

Theorem 17 (Hamming code – the school approach)**.** *Algorithm 3 allows encoding and Algorithm 4 allows for decoding of Hamming codes.*

Algorithm 3 Encoding for Hamming codes – school approach.

Require: Hamming code parameter $n \in \mathbb{N}$, m received word

Construct a table similar to Table 2.1, but instead of ✓, write down placeholders
In all the w_i placeholders write down the corresponding message bits m_j
▷ The remaining placeholders are the p_i entries
For each parity coverage row p_i, write the sum of its entries in the remaining placeholder

Algorithm 4 Decoding for Hamming codes – school approach.

Require: Hamming code parameter $n \in \mathbb{N}$, m received word

Construct the Hamming code table similar to Algorithm 3, but write down the parity bits from m itself
For each row, calculate its sum
if All rows have a sum of 0 **then**
  return m ▷ It is already a codeword
else
  Determine the unique w_i column whose entries are exactly in the rows which have a sum of 1
  $c \leftarrow m$, but with the bit at that position flipped
  return c
end if

Proof. We need to prove two properties:

Finiteness: Should be obvious since there is no possibility for an endless loop.

Correctness: Follows directly from the apparent 1-1 correspondence between the table approach and the Venn diagram approach. □

This approach is able to handle any binary Hamming code of any length without the need to draw eccentric Venn diagrams, as mentioned above. All that remains is to determine the properties of the Hamming codes. For this, a new nomenclature is introduced.

Definition 9. $\mathrm{Ham}(r, q)$ denotes the q-ary Hamming code with r redundancy bits. For example, Figures 2.2 and 2.3 and Table 2.1 are based on the code $\mathrm{Ham}(3, 2)$. Nonbinary Hamming codes will not be introduced further in this work, since they are rather unimportant in the context of fuzzy extractors and hence most often $q = 2$ in this work.

Lemma 18. *Let C be the Hamming code* $\mathrm{Ham}(r, 2)$ *with $r \geq 2$. Then, we have:*

- $n = 2^r - 1$
- $k = n - r$
- $d = 3$

In conclusion, $\mathrm{Ham}(r, 2)$ *is a* $[2^r - 1, 2^r - r - 1, 3]$*-linear code.*

Proof. We prove each property separately:

- As explained in Remark 16, we must have $n = 2^r - 1$ since the "outer face" is disregarded. Alternatively, it should be noted that in $\mathbb{F}_2^r$, there are exactly $2^r - 1$ distinct 1-dimensional subspaces [11].
- As will be explained later, the parity coverage rows of Table 2.1 directly induce the parity-check matrix of the describing Hamming code. Since the parity bits p_i obviously have a single bit parity coverage, the parity-check matrix H consisting of r rows has full rank, and hence according to Lemma 4, $k = n - \mathrm{rk}(H) = r$.
- Using the table approach, it is obvious that two codewords that have the minimum Hamming distance to each other are codewords where the only bit that differs is one that is covered by exactly two parity bits, *e.g.*, in Table 2.1, w_3, w_5 and w_6. Whenever one of these bits changes, exactly two parity bits also change, so $d = 3$. □

2.1.1.2 Golay Codes

Shortly after Hamming found his class of ECCs, but had not yet published them on his own, Golay's famous letter-to-the-editor [30] surfaced, which contains two generator matrices for two different codes [26]. These special codes are now known as *Golay codes*, and they have very specific properties that will be examined in this section. First, however, it should be mentioned that Golay codes are often considered to be the "first" ECCs [26], which does not seem quite correct, since Shannon's paper on Hamming codes [24] predates Golay's letter-to-the-editor. However, what is undisputed is that the Golay codes are the first multiple ECCs, as we shall see [25].

Definition 10. The Golay codes G_{11} and G_{23} were originally defined by Golay [30] through their respective generator matrix $\left[I_6 \mid A_{11}^T\right]$ and $\left[I_{12} \mid A_{23}^T\right]$. Here, I_n is the identity matrix of size $n \times n$ and the matrices A_n are defined as follows:

$$A_{11} = \begin{bmatrix} 1 & 1 & 1 & 2 & 2 & 0 \\ 1 & 1 & 2 & 1 & 0 & 2 \\ 1 & 2 & 1 & 0 & 1 & 2 \\ 1 & 2 & 0 & 1 & 2 & 1 \\ 1 & 0 & 2 & 2 & 1 & 1 \end{bmatrix} \quad \text{and} \quad A_{23} = \begin{bmatrix} 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Remark 19. There are many different equivalent ways of defining the Golay codes. For example, in Section 4.2.2 a different generator matrix is used instead.

Alas, it is very difficult to find an intuitive way to describe the Golay codes and their encoding and decoding procedures. For the ternary Golay code, there are some algebraic algorithms [31]. Also, as both Golay codes are also Quadratic Residue Codes, they can be decoded using such algorithms [8]. This section will again focus on the binary Golay code since the ternary one is rather unimportant in the context of fuzzy extractors. There are a few attempts to make the encoding and decoding procedures a bit more tangible. For this, first a few basics need to be introduced.

Remark 20. The Golay code is redefined here. The "new" Golay code actually has a length of 24 instead of 23. This is because an "extended" binary Golay code is described instead, which is further explained in Definition 37.

Definition 11. A *dodecahedron* is a three-dimensional shape consisting of 12 flat *faces*. It is most often drawn using congruent *pentagonal* (two-dimensional shape with 5 *vertices* and *edges* each) faces, which also makes it a regular shape and one of the Platonic solids. It is shown in Figure 2.4.

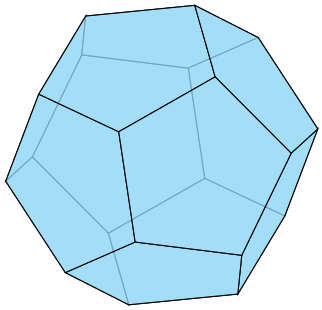


Figure 2.4: A regular dodecahedron, adapted from Apolinar [32].

Definition 12. The *face graph* of a shape is a graph whose vertices correspond to the respective faces of the shape. There is an edge connecting two vertices if and only if the two corresponding faces on the shape share a common edge. For the dodecahedron, the face graph is shown in Figure 2.5. It is equal to the *vertex graph* (construction analogous to the face graph) of the regular icosahedron (three-dimensional shape with 20 faces and 12 vertices).

Definition 13. The *adjacency matrix* of a graph is a matrix whose entry (i, j) is 1 if there is an edge connecting vertex i to j and 0 otherwise. The adjacency matrix of the face graph of the dodecahedron from Figure 2.5 is

$$\begin{bmatrix}
0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\
0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\
1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\
0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\
0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\
0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0
\end{bmatrix}.$$

As expected, every row and column contains exactly five 1s since every face is adjacent to five other faces on the dodecahedron. Also, the matrix is symmetric because adjacency is a

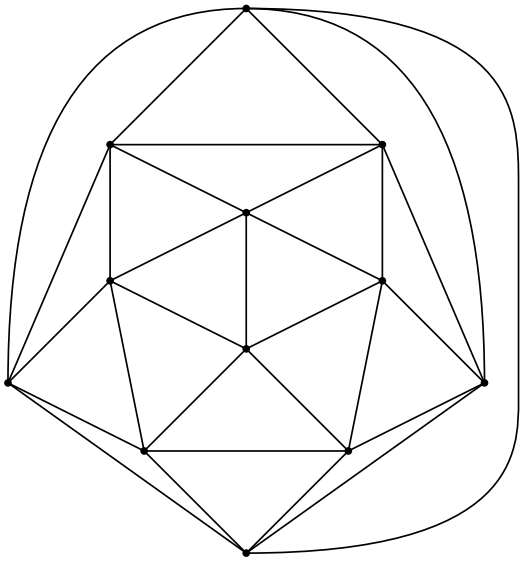


Figure 2.5: Face graph of the dodecahedron, own creation.

symmetric relation between two faces, that is, if face i is a neighbour of face j, then face j is also a neighbour of face i.

Lemma 21. *The binary Golay code consists of 4096 binary words of length 24. All codewords fall into these categories:*

- *The words* $\mathtt{0}\cdots\mathtt{0}$ *(called the* zero word*) and* $\mathtt{1}\cdots\mathtt{1}$ *(called the* ones word*)*
- 759 *words of weight* 8 *(called* octads*)*
- 2576 *words of weight* 12 *(called* dodecads*)*
- 759 *words of weight* 16 *(called* hexadecads*)*

Proof. Using the generator matrix, all codewords can be enumerated. □

Corollary 22. The extended binary Golay code C has a minimum distance of $d(C) = 8$.

Proof. Follows directly from Lemmata 2 and 21. □

Now, based on the work of MacWilliams and Sloane [11] and Curtis [33, 34], Fields [35] suggests using a generator matrix $[I_{12} \mid A]$ for the binary Golay code with

$$A \coloneqq \mathbb{1}_{12\times 12} - F = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \end{bmatrix}$$

and F being the adjacency matrix of the face graph of the dodecahedron from Definition 13. This connection originates from a rather obvious fact: The A part of the generator matrix from Definition 10 contains exactly five `0`s in each row, except the last one, which only calculates a single extension parity bit. Therefore, there is a connection between the dodecahedron and the extended binary Golay code. According to Fields [35], the encoding procedure can now be defined.

Theorem 23 (Encoding of G_{24} – using the dodecahedron)**.** *Algorithm 5 allows encoding of the extended binary Golay code.*

Algorithm 5 Encoding the extended binary Golay code using the regular dodecahedron.

Require: Message to encode and an empty dodecahedron

Write the information bits onto the dodecahedron, one bit per face, using colour A
Place the mask from Figure 2.6 on a face of the dodecahedron
Calculate the parity bit (sum) of the *unmasked* bits in colour A
Write this parity bit onto the face using colour B
Repeat the last three steps for all the remaining faces
The redundant bits are exactly the 12 bits in colour B

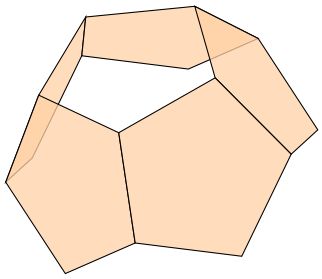


Figure 2.6: Golay code mask for the dodecahedron, adapted from Apolinar [32] and Fields [35].

Proof. Finiteness: The procedure clearly cannot run into an infinite loop.

Correctness: According to Fields [35], any given face on the dodecahedron has one face with distance 0 (itself), five faces with distance 1 (the adjacent faces), another five faces with distance 2 (the "bottom half" when considering the primary face facing up) and one face with distance 3 (the opposite face). Since the extended binary Golay code has a minimum Hamming weight of 8, we must somehow accommodate another seven `1`s when encoding a word of weight 1. The obvious way is to "mask" all the faces with distance 1 and only consider all the *other* seven faces with distances 0, 2 and 3 when calculating the corresponding parity bits. The corresponding mask is shown in Figure 2.6. □

Remark 24. It makes sense to number the different faces of the dodecahedron such that the order of the bits can be preserved. When applying Algorithm 5, one should first extract all the bits in colour A in the given order and then - in the same order - append the bits in colour B such that the resulting codeword matches the systematic structure of the "new" generator matrix.

Now an example might be appropriate. We will now try to encode the word `100000000000` using the approach of Algorithm 5. To better visualise this, Figure 2.7 shows an open-folded dodecahedron with numbered faces. We will now define the colours A and B as black and green, respectively. First, the word to encode is written on the dodecahedron which results

in one black mark as seen in Figure 2.7 (a). Now, the mask is being placed on each face (imagining a closed dodecahedron, of course), and the unmasked black bits are summed up. In this special case, this results in a set bit if and only if the current face is not a neighbour of the face with the black mark. These computed bits are now written to the faces of the dodecahedron. In conclusion, the encoded word can be read as

`100000000000100000111111`.

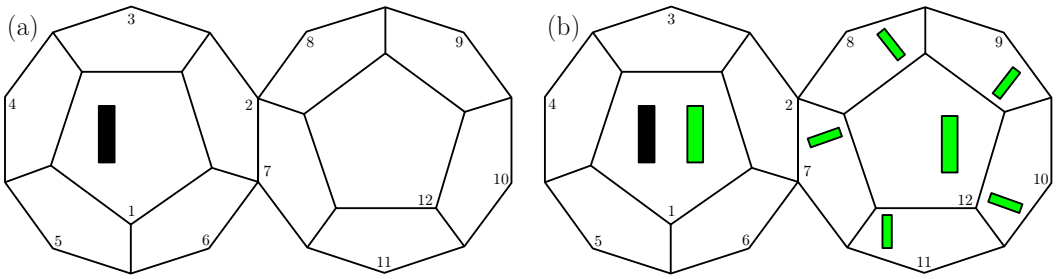


Figure 2.7: Encoding of the word `100000000000` using open folded dodecahedra, adapted from Fields [35].

Now, the only thing left to discuss is the decoding of the extended binary Golay code using dodecahedra. To accomplish this, Lemma 1 and Corollary 22 implies that the extended binary Golay code is capable of correcting up to 3 errors. However, there is a special case: If 4 errors have been introduced, they can be detected but not corrected, since there may be two different, equally likely original codewords, both of which have a Hamming distance of 4 to the received message (recall $d(C) = 8$). Correction is therefore not possible, but detection is.

Theorem 25 (Decoding of G_{24} – using the dodecahedron)**.** *Algorithm 6 allows decoding of the extended binary Golay code.*

Algorithm 6 Decoding the extended binary Golay code using the regular dodecahedron.

Require: Message to decode and an empty dodecahedron

Write the information bits onto the dodecahedron, one bit per face, using colour A
Write the parity bits onto the dodecahedron, one bit per face, using colour B
Place the mask from Figure 2.6 on a face of the dodecahedron
Calculate the parity bit (sum) of the *unmasked* bits in colour A
To this sum, add the parity bit of the current face in colour B
If the sum is odd, mark the current face using colour C, indicating that the check failed
Repeat the last four steps for all the remaining faces
if No face has colour C **then**
    return No error has occurred
else if 3 or less faces have colour C **then**
    Flip the bits in colour B in the faces from the error pattern and stop
end if
Consider pattern of the faces with colour C and process it ▷ See Theorem 26
if Pattern is recognised **then**
    Correct the corresponding incorrect bit(s)
else
    Conclude that 4 errors have been introduced
end if

Proof. Finiteness: Since the dodecahedron only has twenty faces, there is no possibility for an infinite loop.

Correctness: Follows from Theorem 26. □

This algorithm looks quite simple, but there is a small problem: Pattern recognition is quite complex and needs to be able to detect errors within the error pattern. The following example illustrates the problem. However, first, the error patterns must be defined properly. They can be found in Figure 2.8.

In the case of the parachute, it is clear that only one information bit error has occurred since it is exactly the opposite of the mask of Figure 2.6. In the case of the diaper, bent ring, and tropics, two information bit errors have been detected, and the rest of the patterns indicate three information bit errors. Now, the problem mentioned above becomes apparent: If information bit and parity bit errors occur simultaneously, the resulting error pattern is not directly included in the list, but the pattern has some faces removed or added. Fortunately, the number of possibilities for the error pattern is limited, since only 3 errors can be corrected.

Theorem 26 (Decoding of G_{24} – visualising the error patterns)**.** *During decoding of the extended binary Golay code using Algorithm 6, one needs to only consider the following error patterns:*

- *Empty error pattern* $\iff$ *No error*
- *Flawless parachute* $\iff$ 1 *information bit error*
- *Single face* $\iff$ 1 *parity bit error*
- *Flawless diaper, bent ring or tropics* $\iff$ 2 *information bit errors*
- *Parachute with a modified face* $\iff$ 1 *information bit error and* 1 *parity bit error*
- *Two faces* $\iff$ 2 *parity bit errors*
- *Flawless cage, cobra, islands, broken tripod or deep bowl* $\iff$ 3 *information bit errors*
- *Diaper, bent ring or tropics with a modified face* $\iff$ 2 *information bit errors and* 1 *parity bit error*
- *Parachute with two modified faces* $\iff$ 1 *information bit error and* 2 *parity bit errors*
- *Three faces* $\iff$ 3 *parity bit errors*

Proof. Through an enumeration of all the possible error patterns, it can be proven that the list is complete. □

Finally, another example is much needed. Suppose we received the word

`000000000001100000101111`

and we are using black, green, and red as colours A, B and C, respectively, exactly like Fields [35]. To decode it using Algorithm 6, all the information bits and parity bits are written onto the dodecahedron and we arrive at Figure 2.9 (a). Next, the decoding checks produce the error pattern illustrated in Figure 2.9 (b) and (c). Clearly, at least one information bit must be incorrect, as the error pattern consists of 9 faces. The two obvious choices with 9 faces would be the deep bowl and the cage, which do not even allow for modification. However, the observed pattern looks different from both. The same is true for the cobra, islands, and broken tripod. In addition, the diaper and the bent ring would need a modification of at least 3 faces each since they consist of 6 faces. But this would indicate at least 3 parity bit errors, which contradicts our obvious assumption above that there must be at

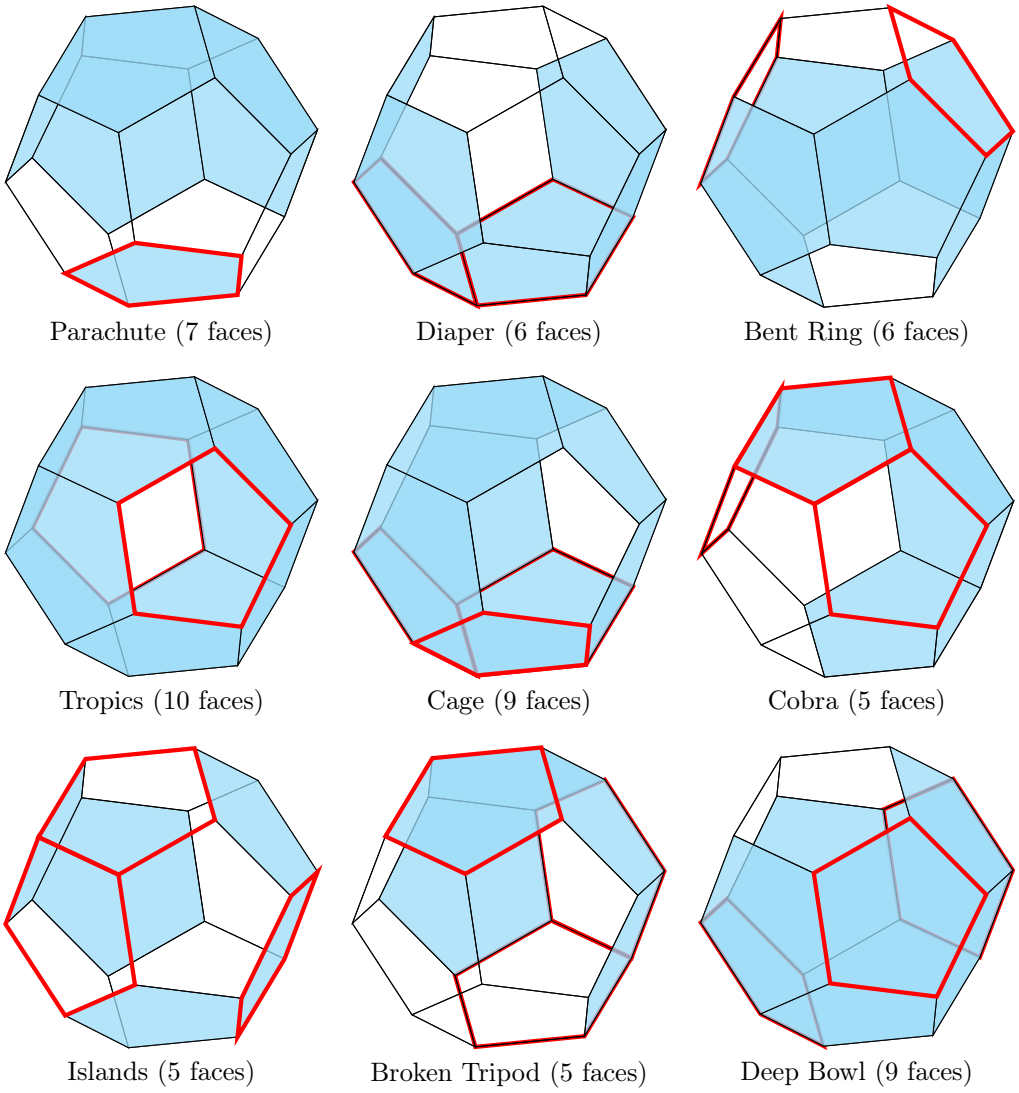


Figure 2.8: The 9 error patterns for the extended binary Golay code, adapted from Apolinar [32] and Fields [35].

least one information bit error (again, we can only correct 3 errors in total). A parachute would need two additional faces in order to fit our observed pattern. However, there cannot be two faces opposite to each other such as faces 1 and 12 in Figure 2.9 (a). Hence, the only solution is the tropics with a parity bit error at position 8. After correcting the error indicated by the tropics and the single parity bit, we arrive at Figure 2.9 (d) which can be translated back into the codeword

$$\texttt{100000000000100000111111}.$$

As is evident, this algorithm relies on pattern recognition and has to take into account many special cases that make it difficult to implement on a computer. Of course, there are many other procedures that are optimised for different use cases and systems.

First, Pless [36] describes algorithms that use the *hexacode*, a code with parameters $[6, 3, 4]$ over the field $\mathbb{F}_4$, and also a *Steiner system*. The name goes back to a combinatorial exercise by Steiner [37] that asks how many elements N can be ordered into triples in such a way

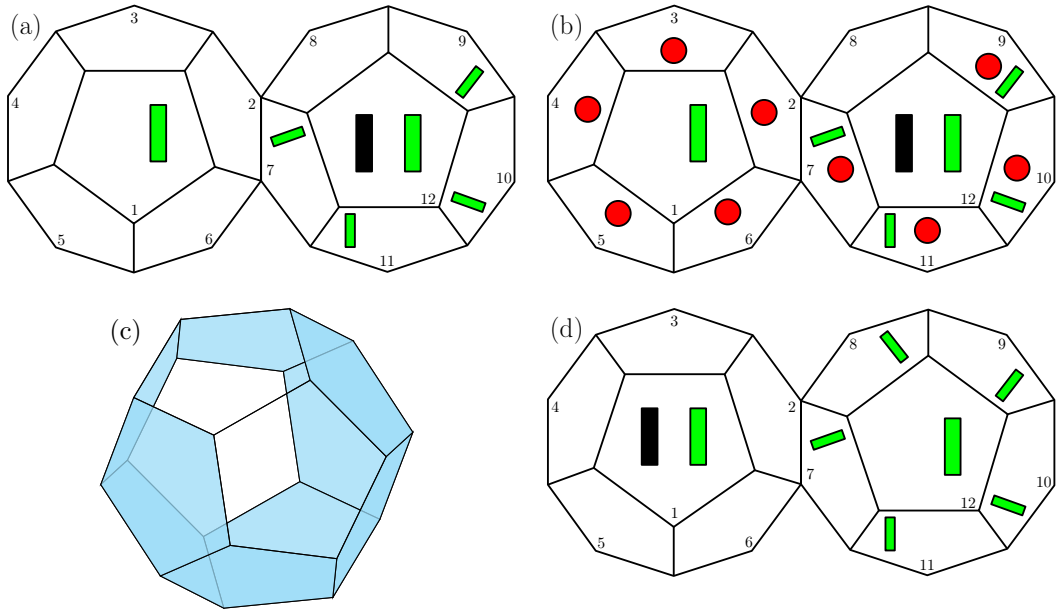


Figure 2.9: Decoding of the word `000000000001100000101111` using open folded and closed dodecahedra, adapted from Apolinar [32] and Fields [35].

that two each occur in one but only in one triple. Nowadays, such problems can be solved by the so-called *block designs*, which divide units into blocks in such a way that each element is selected exactly the same number of times. Steiner systems are then defined as generalised block designs where every subset of t elements appears in exactly one partition. Finally, the *Witt designs* are the only two Steiner systems with $t = 5$ and the number of elements 12 and 24. They have been discovered by Carmichael [38] and Witt [39]. When everything is put together, one can observe that there is a direct connection between the octads of the extended ternary and extended binary Golay codes and the elements of the larger Witt design with 12 and 24 elements, respectively [40].

Additionally, Blaum and Bruck [41] describe an additional algorithm which is able to decode the non-extended binary Golay code using Venn diagrams. By extension, it is thus also able to decode the extended binary Golay code.

Finally, as mentioned above, both Golay codes are also Quadratic Residue Codes and can therefore be encoded and decoded with algorithms specific to them.

Now, the only thing left to discuss about the Golay codes is their parameters.

Lemma 27. *The Golay codes have the properties described in Table 2.2.*

Table 2.2: Properties of the Golay codes, own creation.

Code	n	k	d
Binary Golay code	23	12	7
Extended binary Golay code	24	12	8
Ternary Golay code	11	6	5
Extended ternary Golay code	12	6	6

Proof. The distance of 8 for the extended binary Golay code has already been "proven" in Corollary 22. The rest follows through enumeration. □

Historically, the binary Golay code has been used, for instance, in Voyager missions [40] or the military standard 188 (MIL-STD-188) and federal standard 1045 (FED-STD-1045) [42]. Moreover, it has been proposed for use in modern wireless applications [43] and both extended Golay codes have been used to prove the Kochen-Specker theorem in quantum mechanics [44]. But it should also be mentioned that the binary Golay code is still used in some applications due to its unique properties as being one of the only known binary perfect codes (along with, for example, Hamming codes).

2.1.1.3 Reed-Muller Codes

This class of ECCs has been found by Muller [45] in 1954. The original encoding scheme was based on Boolean algebra, and only detection of errors was possible. However, in the same year, Reed [46] found an iterative algorithm which was additionally able to correct errors in an efficient way, called multi-step majority-logic decoding [47]. Nowadays, these codes are called Reed-Muller (RM) codes, named after these two authors, and they have been used for instance in the Mariner space missions [48, 49]. It should also be mentioned that nowadays nonbinary RM codes can be constructed, but again, they will not be further explained here due to their limited relevance to fuzzy extractors [50].

In this section, the RM codes will be defined using a construction scheme different from that of Muller [45].

Definition 14. Let C_1 and C_2 be two linear codes of length n over $\mathbb{F}_q$. Then, the *code obtained from C_1 and C_2 by the $(u \mid u+v)$-construction* or *by the Plotkin construction C* is defined by $C \coloneqq (C_1 \mid C_2) = \{[u, u+v] \mid u \in C_1, v \in C_2\} \in \mathbb{F}_q^{2n}$.

If a code C is constructed using this Plotkin construction (named after its inventor Plotkin [51]), it satisfies a few properties that make it easy to obtain a generator matrix.

Lemma 28. *Let $C \coloneqq (C_1 \mid C_2)$ be a code over $\mathbb{F}_q$ which has been constructed using the Plotkin construction of linear codes C_1 and C_2 over $\mathbb{F}_q$ with generator matrix G_1 and G_2, respectively. Then, according to Betten [13], C has the generator matrix*

$$G \coloneqq \begin{bmatrix} G_1 & G_1 \\ 0 & G_2 \end{bmatrix}.$$

Proof. Let $[c_1, c_2] \coloneqq c \in C$ be a codeword. According to Definition 14, $c_1 \in C_1$ and $c_2 = c_1 + v$ with $v \in C_2$. Obviously, c_1 has been encoded using only G_1, that is, according to Theorem 5, we have $c_1 = w_1 \cdot G_1$ for some w_1. Similarly, $c_2 = c_1 + v = w_1 \cdot G_1 + w_2 \cdot G_2$ for some w_2. Finally, we observe that $[w_1, w_2] \cdot G = [w_1 \cdot G, w_1 \cdot G_1 + w_2 \cdot G_2] = [c_1, c_2] = c$ using G as defined above, which proves the statement. □

Lemma 29. *Let C_1 be a $[n, k_1, d_1]$-linear code and C_2 be a $[n, k_2, d_2]$-linear code over $\mathbb{F}_q$. Then $C \coloneqq (C_1 \mid C_2)$ constructed using the Plotkin construction is a $[2n, k_1 + k_2, \min\{2d_1, d_2\}]$-linear code, according to Betten [13].*

Proof. We prove each property separately:

- Due to the size of the generator matrix, it is obvious that C has a block length of $2n$.
- Let B_1 and B_2 be a basis of C_1 and C_2, respectively. Then, clearly, $\{[u_1, u_1], \ldots, [u_{k_1}, u_{k_1}], [0, v_1], \ldots, [0, v_{k_2}] \mid u_i \in B_1, v_j \in B_2\}$ is a basis of C.
- Obviously, a nonzero codeword of minimum weight in $C = [c_1, c_1 + c_2]$ must have either $c_1 = 0$ and $c_2 \neq 0$ or $c_1 \neq 0$ and $c_2 = 0$. Hence, $\mathrm{wt}(C) = \min\{2\mathrm{wt}(c_1), \mathrm{wt}(c_2)\}$. In addition to Lemma 2, this proves the statement. □

Now, RM codes can be defined using this Plotkin construction scheme.

Definition 15. The RM code with parameters $m \geq 0$ and $0 \leq r \leq m$, short $\mathrm{RM}(r, m)$, is defined recursively as follows:

- $\mathrm{RM}(0, m) \coloneqq \{0, \mathbb{1}\}$
- $\mathrm{RM}(m, m) \coloneqq \mathbb{F}_2^{2^m}$
- $\mathrm{RM}(r, m) \coloneqq (\mathrm{RM}(r, m-1) \mid \mathrm{RM}(r-1, m-1))$ for $0 < r < m$ using Plotkin construction

Corollary 30. The generator matrix $G(r, m)$ of the code $\mathrm{RM}(r, m)$ can be calculated recursively using

- $G(0, m) = \underbrace{\begin{bmatrix} 1 & \cdots & 1 \end{bmatrix}}_{2^m \text{ times}}$
- $G(m, m) = \begin{bmatrix} G(m-1,m) \\ 0\ 0\ \cdots\ 0\ 1 \end{bmatrix}$
- $G(r, m) = \begin{bmatrix} G(r,m-1) & G(r,m-1) \\ 0 & G(r-1,m-1) \end{bmatrix}$ for $0 < r < m$

Proof. This follows directly from Definition 15 and Lemma 28. □

Again, using the Plotkin construction, the usual properties for the RM codes can be easily calculated.

Lemma 31. *The code* $\mathrm{RM}(r, m)$ *has the following properties:*

- $n = 2^m$
- $k = \sum_{i=0}^{r} \binom{m}{i}$
- $d = 2^{m-r}$

In conclusion, $\mathrm{RM}(r, m)$ *is a binary* $[2^m, \sum_{i=0}^{r} \binom{m}{i}, 2^{m-r}]$*-linear code.*

Proof. We prove each property separately:

- Follows directly from Lemma 29 and Definition 15.
- Obviously, the claim holds for $r = 0 = m$. Using Lemma 29, we can inductively calculate $\dim(\mathrm{RM}(r, m)) = \dim(\mathrm{RM}(r, m-1)) + \dim(\mathrm{RM}(r-1, m-1)) = \sum_{i=0}^{r} \binom{m-1}{i} + \sum_{i=0}^{r-1} \binom{m-1}{i} = \sum_{i=1}^{r} \binom{m-1}{i} + \sum_{i=1}^{r} \binom{m-1}{i-1} + \binom{m-1}{0} = \sum_{i=1}^{r} \left(\binom{m-1}{i} + \binom{m-1}{i-1} \right) + 1 = \sum_{i=1}^{r} \binom{m}{i} + \binom{m}{0} = \sum_{i=0}^{r} \binom{m}{i}$, since $\binom{m}{i} = \binom{m-1}{i} + \binom{m-1}{i-1}$.
- Obviously, the claim holds for $r = 0 = m$. Using Lemma 29, we can inductively calculate $d(\mathrm{RM}(r, m)) = \min\{2d(\mathrm{RM}(r, m-1)), d(\mathrm{RM}(r-1, m-1))\} = \min\{2 \cdot 2^{(m-1)-r}, 2^{(m-1)-(r-1)}\} = \min\{2^{m-r}, 2^{m-r}\} = 2^{m-r}$. □

Alas, the decoding procedure for the RM codes is even more complicated than the procedure for the Golay codes. Furthermore, the set of RM codes describes a rather large class of codes, making it difficult to find a general algorithm that can decode them by hand. However, for certain fixed RM codes, there are some attempts to make them a bit more "tangible," such as for $\mathrm{RM}(2, 5)$ [52, 53]. For this reason, this section will only introduce a decoding algorithm, called *majority logic decoding*, which is based on Boolean algebra, but is still a bit different from the original schemes by Muller [45] and Reed [46]. The entire derivation from now on and the final decoding algorithm is based on Hoffman's work [54].

Definition 16. *Standard ordering* $<_{\text{std}}$ on $\mathbb{F}_2^m$ is defined as follows: Take the two binary vectors v_1 and v_2 to compare and interpret them as binary numbers written in reverse a_1 and a_2, respectively. Now $v_1 <_{\text{std}} v_2$ iff $a_1 < a_2$.

Definition 17. Let $\{u_0, \ldots, u_{2^m-1}\}$ be every element in $\mathbb{F}_2^m$ sorted using the standard ordering as described in Definition 16. Any function $f\colon \mathbb{F}_2^m \to \mathbb{F}_2$ has a unique *vector form* $v = (f(u_0), \ldots, f(u_{2^m-1})) \in \mathbb{F}_2^m$.

Definition 18. In context of the vector space $\mathbb{F}_2^m$, for a subset $I \subseteq \{0, \ldots, m-1\}$, the function $f_I\colon \mathbb{F}_2^m \to \mathbb{F}_2$ is defined as follows:

$$f_I(x_0, \ldots, x_{m-1}) = \prod_{i \in I}(x_i + 1).$$

Note that $f_\emptyset(x_0, \ldots, x_{m-1}) = 1$.

Also, $f_{I,t}(x_0, \ldots, x_{m-1}) = f_I(x_0 + t_0, \ldots, x_{m-1} + t_{m-1})$ where $t = (t_0, \ldots, t_{m-1})$. Therefore, we have $f_{I,0} = f_I$.

Moreover, v_I and $v_{I,t}$ are the vector forms as described in Definition 17 of f_I and $f_{I,t}$, respectively.

Of course, these functions have some important properties that allow them to be used to describe all codewords of RM codes.

Lemma 32. *For any $I \subseteq \{0, \ldots, m-1\}$, we have*

$$f_I(x_0, \ldots, x_{m-1}) = 1 \iff I = \emptyset \text{ or } \forall i \in I\colon x_i = 0.$$

Proof. If $I = \emptyset$, the claim is trivial. Otherwise, we have $f_I(x_0, \ldots, x_{m-1}) = 0$ iff one of the factors in $\prod_{i \in I}(x_i + 1)$ is 0 (iff any $x_i = 1$). By contraposition, the claim follows. □

Definition 19. The *support* of a function $f\colon V \to \mathbb{F}_2$ is defined as the set of all inputs $v \in V$ that map to 1, that is, $\operatorname{supp}(f) = \{v \in V \mid f(s) = 1\}$.

Remark 33. It should be noted that the *support* of a function is often defined differently. For example, as being the set of all inputs $s \in S$ that evaluate to a non-zero element, *i.e.* $\{s \in S \mid f(s) \neq 0\}$. In this case, the definition is equivalent as the only nonzero element in $\mathbb{F}_2$ is 1. However, there are many more definitions that could differ from the one used in this work.

Definition 20. Based on Definitions 18 and 19, the set S_I is now used as a short notation for the support of f_I, that is, $S_I = \{u \in \mathbb{F}_2^m \mid f_I(u) = 1\}$.

Now, the RM codes are redefined in terms of these "Boolean polynomials."

Theorem 34 (RM polynomials)**.** *There is an isomorphism between the codewords of* $\mathrm{RM}(r, m)$ *and the vector space* $\langle v_I \mid I \subseteq \{0, \ldots, m-1\}, |I| \leq r\rangle$.

Proof. According to Hoffman [54], this follows from the fact that any codeword[3] $f \in \mathrm{RM}(r, m)$ can be written uniquely as $f = \sum_{I \subseteq \{0,\ldots,m-1\}, |I| \leq r} m_I \cdot f_I$ for some combination of $m_I \in \mathbb{F}_2$. This in turn is correct since $\{f_I \mid I \subseteq \{0, \ldots, m-1\}\}$ is a basis of $\{f\colon \mathbb{F}_2^m \to \mathbb{F}_2\}$, the set of all Boolean polynomials from $\mathbb{F}_2^m$ to $\mathbb{F}_2$. □

Remark 35. Due to the identity of Theorem 34, the code $\mathrm{RM}(r, m)$ is also often referred to as the *m-variate RM code of degree r*.

[3] if translated from vector form to its corresponding Boolean polynomial

Lemma 36. *Let $c \in \mathrm{RM}(r,m)$ be a codeword and I be a set with $|I| \leq r$. Then, $m_I = c \cdot f_{I^c,t}$ for any $t \in S_I$.*

Proof. Due to the complexity of this proof, we refer to Hoffman's work [54]. □

Lemma 37. *For any $e \in \mathbb{F}_2^{2^m}$, we have $e \cdot f_{I^c,t} = 1$ for at most $\mathrm{wt}(e)$ values of $t \in S_I$ with $I \subseteq \{0,\ldots,m-1\}$.*

Proof. Again, following Hoffman [54], obviously, by Lemma 32, we have $S_I \cap S_{I^c} = \{0\}$. Let $t_1, t_2 \in S_I$ be with $t_1 \neq t_2$. Then, by Lemma 8, we have $(S_{I^c}+t_1) \cap (S_{I^c}+t_2) = \emptyset$ due to each coset of S_{I^c} that contains only one word of S_I at a time, since the sum of two words must be in S_I for them to be in the same coset. In conclusion, v_{I^c,t_1} and v_{I^c,t_2} cannot have any positions with a 1 in common. Hence, every 1 in e affects $e \cdot v_{I^c,t}$ for exactly one $t \in H_I$. □

In order to find a decoding algorithm, we note that any received word w can be written again in terms of the codeword initially sent $c \in \mathrm{RM}(r,m)$ and an error vector e as $c = w + e$. Due to the isomorphism mentioned in Theorem 34, we can write $c = \sum_{I \subseteq \{0,\ldots,m-1\}, |I| \leq r} m_I \cdot f_I$. Now, according to Lemmata 36 and 37, we have for at least $|S_J| - \mathrm{wt}(e)$ values of $t \in S_J$:

$$\begin{aligned} w \cdot f_{I^c,t} &= c \cdot f_{I^c,t} + e \cdot f_{I^c,t} \\ &= c \cdot f_{I^c,t} + 0 \\ &= m_I. \end{aligned}$$

Now, decoding is done by calculating each m_I in order and then iteratively (or recursively) applying the same procedure using $\mathrm{RM}(r-1,m)$.

Theorem 38 (Decoding RM codes)**.** *Algorithm 7 allows decoding of RM codes.*

Algorithm 7 Decoding of RM codes using majority logic [54].

Require: $n = 2^m$, $t = 2^{m-r-1} - 1$, $g\colon \mathbb{F}_2^m \to \mathbb{F}_2$ received word containing up to t errors

```
f(r) ← g
for i ← r downto 0 do
    for each I ⊆ M with |I| = i do
        Σ_I ← [⟨f(i), f_{I^c,u}⟩ | u ∈ S_I]
        if Σ_I contains 0s and 1s ≥ 2^{m-i-1} times each then return Cannot decode!
        else if Σ_I contains ≥ 2^{m-i-1} 0s then m_I ← 0
        else if Σ_I contains ≥ 2^{m-i-1} 1s then m_I ← 1
        else return Cannot decode!
        end if
    end for each
    if i = 0 then
        return Σ_{I⊆M,|I|≤r} m_I · f_I
    else
        f(i-1) ← f(i) − Σ_{I⊆M,|I|=i} m_I · f_I
    end if
    if wt(f(i-1)) ≤ t then
        return Σ_{I⊆M,i-1≤|I|≤r} m_I · f_I
    end if
end for
```

Proof. Due to the complexity of this proof, we again refer to Hoffman's work [54]. However, the basic inner workings can be proven directly through Theorem 34 and Lemmata 36 and 37. □

Now that a working encoding and decoding procedure has been proposed, it is time to propose another approach that is especially useful in the context of Computer Algebra Systems (CASs). These usually provide methods to calculate the Gröbner bases [55, 56]. In 2016, Abrahamsson [57] proposed a way to encode codewords for general linear codes using Gröbner bases while providing a special case study on RM codes. In contrast, in 2018, Andriatahiny *et al.* [58] proposed a way to decode RM codes using Gröbner bases.

It is also worth mentioning that the so-called *augmented Hadamard codes* are closely related to the RM codes and have a similar construction scheme using *Hadamard matrices* [59, 60]. A Hadamard matrix is a square matrix whose rows are pairwise orthogonal. Hence, a Hadamard matrix H_n of size $n \times n$ must satisfy $H_n \cdot H_n^T = n \cdot I_n$ with I_n being the identity matrix of size $n \times n$. Although it is conjectured that there is at least one Hadamard matrix for $n = 1$, 2 and for each multiple of 4 (now known as the *Hadamard conjecture* [61]), for the Hadamard codes, it is only[4] important to consider Hadamard matrices of size $n = 2^m$. These can be easily constructed using *Sylvester's construction* [59] and are also called *Walsh matrices* [62]. This is done recursively as follows:

$$H_1 = \begin{bmatrix} 1 \end{bmatrix} \qquad \text{and} \qquad H_{n+1} = \begin{bmatrix} H_n & H_n \\ H_n & -H_n \end{bmatrix}.$$

As it is evident, the entries of the Hadamard matrices are -1s and 1s. To create a generator matrix from a Hadamard or Walsh matrix H_n of size $n \times n$, all the -1s are converted to 0s. The resulting binary code is then equivalent to the code $\mathrm{RM}(1, \log_2(n))$ and can thus also be decoded using decoding procedures for RM codes [14].

2.1.2 Cyclic Codes

Some linear codes have additional features that make them even superior to traditional linear codes.

Definition 21. A linear code C (see Definition 5) is also a *cyclic code* if for any codeword $[c_0, \ldots, c_{n-2}, c_{n-1}] =: c \in C$ we also have $c' := [c_{n-1}, c_0, \ldots, c_{n-2}] \in C$. In this case, c' is called the *cyclic right shift of* c [13].

Lemma 39. *Let C be a cyclic code and $[c_0, \ldots, c_{n-2}, c_{n-1}] =: c \in C$. Then, any cyclic shift $[c_{i \bmod n}, \ldots, c_{n-2+i \bmod n}, c_{n-1+i \bmod n}]$ with $i \in \mathbb{N}$ of c also satisfies $c' \in C$.*

Proof. Let $c \in C$ be any codeword and c' be its cyclic right shift. Then, we have $c' \in C$ by Definition 21. By iteratively applying Definition 21 to c' and so on, we observe that every codeword shifted must also be in C. □

Remark 40. Definition 21 neither depends on the size nor the direction of the shift. Therefore, cyclic codes can also be defined by specifying that the 42-left shift c' of each codeword $c \in C$ also satisfies $c' \in C$. This is a direct consequence of Lemma 39.

The obvious question now is how cyclic codes can best be described. Since cyclic codes are also linear by Definition 21, there exists a generator matrix for them. However, there are better ways to describe them, some of which will now be introduced.

[4] It is also possible to do the same procedure with other Hadamard matrices, but they might not yield linear codes or they might have worse code rates.

Lemma 41. *Let C be a cyclic code. According to Betten [13], C has a generator matrix of the form*

$$G = \begin{bmatrix} g_0 & g_1 & \cdots & \cdots & g_{n-1} & g_n & 0 & \cdots & 0 \\ 0 & g_0 & g_1 & \cdots & \cdots & g_{n-1} & g_n & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & g_0 & g_1 & \cdots & \cdots & g_{n-1} & g_n \end{bmatrix}.$$

Proof. This statement can be proved by Theorem 42 by showing that the rows of the matrix are linearly independent and generate the codewords [13]. □

Some readers might already have recognised the shift-property from Lemma 39 because it is very similar to polynomial multiplication. Because of this, cyclic codes have a structure very similar to that of certain polynomial rings.

Theorem 42 (Cyclic codes and polynomial rings)**.** *Let $C \subseteq \mathbb{F}_p^n$ be a cyclic code in $\mathbb{F}_p$ and $[c_0, \ldots, c_{n-1}] =: c \in C$ be a codeword in C. Then, according to Hall [6], there is a 1-1 correspondence between each c and its respective polynomial*

$$\tilde{c}(x) = \sum_{k=0}^{n-1} c_k x^k = c_0 + c_1 x + \cdots + c_{n-1} x^{n-1} \in \mathbb{F}_p[x]/\langle x^n - 1 \rangle.$$

Proof. Clearly, each polynomial can only correspond to a unique word $w \in \mathbb{F}_p^n$ and vice versa. Hence, we only need to prove the shift property and the effect of other operations on the codewords.

First of all, a cyclic right shift corresponds to multiplication by x since

$$c_{n-1} x^{n-1} \cdot x \equiv c_{n-1} \pmod{x^n - 1}$$

and hence,

$$\begin{aligned} x \cdot \tilde{c}(x) = x \cdot \sum_{k=0}^{n-1} c_k x^k &= c_0 x + c_1 x^2 + \cdots + c_{n-1} x^n \\ &\equiv c_{n-1} + c_0 x + \cdots + c_{n-2} x^{n-1} \pmod{x^n - 1} \\ &\rightsquigarrow [c_{n-1}, c_0, \ldots, c_{n-2}]. \end{aligned}$$

Clearly, adding two codewords together works the same in polynomial form.

It is also easy to prove by calculation that multiplying $\tilde{c}(x)$ by another polynomial $g(x)$ of degree m is equivalent to multiplying c by a matrix like G from Lemma 41 where g_i is the i-th coefficient from $g(x)$:

$$\begin{aligned} c \cdot G &= \begin{bmatrix} c_0 & \cdots & c_n \end{bmatrix} \cdot \begin{bmatrix} g_0 & g_1 & \cdots & \cdots & g_{m-1} & g_m & 0 & \cdots & 0 \\ 0 & g_0 & g_1 & \cdots & \cdots & g_{m-1} & g_m & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & g_0 & g_1 & \cdots & \cdots & g_{m-1} & g_m \end{bmatrix} \\ &= \begin{bmatrix} c_0 g_0 & c_0 g_1 + c_1 g_0 & \cdots & c_{n-1} g_m + c_n g_{m-1} & c_n g_m \end{bmatrix} \end{aligned}$$

and also

$$\begin{aligned} \tilde{c}(x) \cdot g(x) &= \left(\sum_{k=0}^{n-1} c_k x^k \right) \cdot \left(\sum_{k=0}^{m} g_k x^k \right) = \sum_{k=0}^{n-1+m} \left(x^k \sum_{l=0}^{k} c_l g_{k-l} \right) \\ &= c_0 g_0 + (c_0 g_1 + c_1 g_0) x + \cdots + (c_{n-1} g_m + c_n g_{m-1}) x^{n+m-2} + c_n g_m x^{n+m-1} \\ &\rightsquigarrow \begin{bmatrix} c_0 g_0 & c_0 g_1 + c_1 g_0 & \cdots & c_{n-1} g_m + c_n g_{m-1} & c_n g_m \end{bmatrix} \end{aligned}$$

where c_i and g_i are considered 0 if the index is out of bounds for the respective polynomial. This polynomial multiplication formula follows directly from the *convolution* of the coefficient vectors of the polynomials. □

Remark 43. This 1-1 correspondence between polynomials and codewords in a cyclic code should not be confused with the characterisation of RM codes from Definition 17.

Remark 44. Due to this isomorphism, it is now assumed that the codeword space C consists sometimes of vectors and sometimes of polynomials, depending on which view is more advantageous.

Based on Theorem 42, we can derive a new way to encode messages and check their correctness using a cyclic code.

Corollary 45. Let C be a cyclic code with generator matrix G. Then, there exists a *generator polynomial* $g(x) \in \mathbb{F}_p[x]/\langle x^{n-k} - 1\rangle$ which is able to encode any message $m(x) \in \mathbb{F}_p[x]/\langle x^k - 1\rangle$ (derived from a message in $\mathbb{F}_p^n$ using Theorem 42) to its corresponding codeword by calculating $c(x) = g(x) \cdot m(x)$.

Proof. This is a direct consequence of Theorem 42 and the polynomial is constructed analogously to $g(x)$ in the last part of the proof. □

Corollary 46. Let C be a cyclic code with parity-check matrix H. Then, there exists a *check polynomial* $h(x) \in \mathbb{F}_p[x]/\langle x^n - 1\rangle$ which, when multiplied by any codeword $c(x) \in C$, satisfies $h(x) \cdot c(x) \equiv 0 \pmod{x^n - 1}$.

Proof. Again, this is a direct consequence of Theorem 42. However, it can be constructed from the corresponding generator polynomial $g(x)$ instead. We want to find a polynomial $h(x)$ with

$$0 \equiv x^n - 1 \equiv c(x) \cdot h(x) \equiv (m(x) \cdot g(x)) \cdot h(x) = m(x) \cdot (g(x) \cdot h(x)) \pmod{x^n - 1}$$

for every $m(x) \in \mathbb{F}_p[x]/\langle x^k - 1\rangle$. In conclusion, we have

$$x^n - 1 \equiv g(x) \cdot h(x) \quad \Longleftrightarrow \quad h(x) \equiv \frac{x^n - 1}{g(x)} \pmod{x^n - 1}$$

which is the final formula that calculates the check polynomial. □

Remark 47. It is important to remember that these computations are done using "double" modular arithmetic, *i.e.*, $\bmod p$ and $\bmod x^n - 1$.

Now, we are also able to encode messages using "simple" polynomial multiplication. However, linear codes also offer the possibility of encoding messages in a systematic way, as explained in Definition 4 and Lemma 6. Fortunately, there is also a way to do this using the generator polynomial.

Theorem 48 (Encoding of cyclic codes – the systematic approach)**.** *According to Lütkebohmert [17], any message $m(x)$ can be systematically encoded into a codeword $c(x)$ by first calculating $m'(x) \coloneqq m(x) \cdot x^{n-k}$ and then $c(x) = m'(x) - \bmod(m'(x), g(x))$.*

Proof. According to Schulz [15], we note that $m(x)$ is of degree $\leq k - 1$ since it is an unencoded message and hence $m'(x)$ is of degree $\leq (k-1) + (n-k) = n - 1$. Furthermore, since $g(x)$ is of degree $n - k$ (because of Corollary 45), the polynomial $\bmod(m'(x), g(x))$ is of degree $\leq n - k$ and therefore cannot modify the upper k coefficients of $m'(x)$ when being subtracted from $m'(x)$. Due to this, the coefficients of $m(x)$ have not been modified in the final codeword $c(x)$, and the encoding is really systematic.

Furthermore, by definition $m'(x) = a(x) \cdot g(x) + \mathrm{mod}(m'(x), g(x))$ for some $a(x)$ and hence $c(x) = m'(x) - \mathrm{mod}(m'(x), g(x)) = a(x) \cdot g(x)$, which proves that $c(x) \in C$ is a codeword (more precisely, the non-systematic codeword corresponding to the message $a(x)$). □

Remark 49. It is irrelevant that $m(x)$ is encoded to the codeword $c(x)$ of $a(x)$, since $a(x)$ is systematically encoded to another codeword, *etc.* In terms of obtaining the original codeword, according to Theorem 48, it is enough to extract the vector of coefficients of the upper k terms. Algebraically, this can also be done by calculating

$$\mathrm{mod}(\mathrm{mod}(c(x) \cdot x^k, x^n - 1), x^k).$$

Obviously, Remark 49 does not yet describe how to decode codewords. It just describes a procedure to obtain the original codeword when using systematic encoding. In terms of decoding, a similar procedure to Algorithm 2 can be applied (with proper translations into polynomials). Another well-known procedure for decoding binary cyclic codes is the Meggitt decoder [63] which is implemented mainly in hardware. However, in this section, another decoder is presented which works on all cyclic codes in an algebraic manner and has been proposed for the first time by Cooper [64]. For this, some new terminology needs to be introduced first.

A Short Journey Through Computer Algebra

I hope not to get lost in too many details...

This section is based on the work of Kreuzer and Robbiano [65].

Definition 22. A *term order* $<_\sigma$ is an order relation defined on terms (*i.e.*, 1, x, x^2, $x_1x_2^2x_3$, but not $3 \cdot x^2$ or $x + x^4$ or $x_1x_2 + x_3$) such that for all terms t_1, t_2, t it is

1. total: $t_1 \neq t_2 \implies (t_1 <_\sigma t_2 \text{ or } t_2 <_\sigma t_1)$,
2. compatible with multiplication: $t_1 <_\sigma t_2 \implies t \cdot t_1 <_\sigma t \cdot t_2$ and
3. 1 is the smallest element: $1 \leq_\sigma t$ (equivalent to $(1 <_\sigma t \text{ or } 1 = t)$).

Example 1. The *lexicographic term order* $<_{\texttt{Lex}}$ can be defined on terms by first converting them to exponential vectors, *i.e.*, $x_1x_2^3x_3^2 \mapsto [1, 3, 2]$, and then comparing the vectors lexicographically. Thus, *e.g.*,

$$1 <_{\texttt{Lex}} x_2 <_{\texttt{Lex}} x_2^2 <_{\texttt{Lex}} x_1 <_{\texttt{Lex}} x_1x_2 <_{\texttt{Lex}} x_1x_2^2 <_{\texttt{Lex}} x_1^2.$$

Amongst many others, there is also the *degree-compatible reverse lexicographic term order* $<_{\texttt{DegRevLex}}$ which first compares two terms by degree and applies a reverse lexicographic order if their degrees match:

$$1 <_{\texttt{DegRevLex}} x_2 <_{\texttt{DegRevLex}} x_1 <_{\texttt{DegRevLex}} x_2^2 <_{\texttt{DegRevLex}} x_1x_2 <_{\texttt{DegRevLex}} x_1^2 <_{\texttt{DegRevLex}} x_1x_2^2.$$

Definition 23. Let $<_\sigma$ be a term order and $f(x) = \sum_{k=1}^{m} f_k t_k$ be a polynomial with $t_1 >_\sigma \cdots >_\sigma t_m$.

- *Leading Coefficient* $\mathrm{LC}_\sigma(f) \coloneqq f_1$
- *Leading Term* $\mathrm{LT}_\sigma(f) \coloneqq t_1$
- *Leading Monomial* $\mathrm{LM}_\sigma(f) \coloneqq \mathrm{LC}_\sigma(f) \cdot \mathrm{LT}_\sigma(f) = f_1t_1$

Definition 24. Let $<_\sigma$ be a term order defined on indeterminates $X = \{x_1, \ldots, x_n\}$ and choose $X_{\mathrm{elim}} \subseteq X$ and $X_{\mathrm{keep}} \coloneqq X \setminus X_{\mathrm{elim}}$. Then $<_\sigma$ is an *elimination order* for X_{elim} iff $\forall x_j \in X_{\mathrm{elim}} \colon \forall$terms t over indeterminates from $X_{\mathrm{keep}} \colon x_j >_\sigma t$.

Example 2. $<_{\texttt{Lex}}$ from Example 1 is an elimination order for $X_{\text{elim}} = \{x_1\}$. On the other hand, $<_{\texttt{DegRevLex}}$ is not an elimination order at all.

Definition 25. Let R be a unitary commutative ring. Then, a nonempty subset $I \subseteq R$ is called an *ideal* if

- $\forall f, g \in I \colon f + g \in I$
- $\forall c \in R \colon \forall f \in I \colon c \cdot f \in I$

Lemma 50. *Let $I \subseteq R$ be an ideal in R. If $1 \in I$, we have $I = R$.*

Proof. If $1 \in I$, by Definition 25 we have $1 \cdot r \in I$ for all $r \in R$ and therefore $I = R$. □

Definition 26. Let $G \subseteq R$ be a set of polynomials from the polynomial ring R. Then, the *polynomial ideal* $\langle G \rangle$ is defined as

$$\langle G \rangle \coloneqq \left\{ \sum^{\text{finite}} c_j g_j \mid c_j \in R, g_j \in G \right\}.$$

Definition 27. Let I be an ideal in R and $<_\sigma$ be a term order. Then, the *leading term monoideal* of I is defined as $\mathrm{LT}_\sigma(I) \coloneqq \{\mathrm{LT}_\sigma(f) \mid f \in I \setminus \{0\}\}$

Definition 28. Let $G = \{g_1, \ldots, g_n\} \subsetneq R$ be a set of polynomials, $I = \langle G \rangle$ be a polynomial ideal, and $<_\sigma$ be a term order. Then G is a *σ-Gröbner basis* for I iff $\mathrm{LT}_\sigma(I) = \langle \mathrm{LT}_\sigma(g_1), \ldots, \mathrm{LT}_\sigma(g_n) \rangle$.

Definition 29. The *normal remainder* of f with respect to the non-empty set of polynomials G, also written as $\mathrm{NR}_\sigma(f, G)$ is given by the output `rem` in Algorithm 8.

Algorithm 8 Gröbner Division Algorithm [55, 56].

Require: $f \in P$, tuple $G = [g_1, \ldots, g_s]$ with $g_i \in P \setminus \{0\}$

$R \leftarrow 0 \in P$
$Q \leftarrow \{0, \ldots, 0\} \in P^s$
while $f \neq 0$ **do**
  Let j be minimum index with $\mathrm{LT}_\sigma(g_j) \mid \mathrm{LT}_\sigma(f)$
  if j exists **then**
    $q \leftarrow \frac{\mathrm{LM}_\sigma(f)}{\mathrm{LM}_\sigma(g_j)}$
    $Q_j \leftarrow Q_j + q$
    $f \leftarrow f - q \cdot g_j$
  else
    $R \leftarrow R + \mathrm{LM}_\sigma(f)$
    $f \leftarrow f - \mathrm{LM}_\sigma(f)$
  end if
end while
return $[\mathrm{rem} \leftarrow R, \mathrm{quots} \leftarrow Q]$

Remark 51. If $G = \{g\}$ is a set that contains a single polynomial, we note that G is a Gröbner basis for $I = \langle G \rangle$ with respect to any term order and $\mathrm{NR}(f, G) = \mathrm{mod}(f, g)$.

In general, when using Algorithm 8, the relation between `rem`, $\texttt{quots} = \{q_1, \ldots, q_s\}$, f and a σ-Gröbner basis $G = \{g_1, \ldots, g_s\}$ is the following:

$$f = \texttt{rem} + \sum_{k=1}^{s} q_k g_k.$$

Hence, it calculates a representation of f in terms of G with remainder `rem`.

Theorem 52 (Buchberger algorithm)**.** *Algorithm 9 finds a σ-Gröbner basis G for $I = \langle f_1, \ldots, f_r \rangle$ with respect to the term order $<_\sigma$.*

Algorithm 9 Buchberger's Algorithm [55, 56].

Require: Tuple $[f_1, \ldots, f_r]$ with $f_i \in P \setminus \{0\}$

$G = [g_1, \ldots, g_r] \leftarrow [f_1, \ldots, f_r]$
$B \leftarrow \{[j, k] \mid 1 \leq j < k \leq r\}$
while $B \neq \emptyset$ **do**
  Pick a pair $[j, k]$ from B and remove it from B
  $t \leftarrow \operatorname{lcm}(\mathrm{LT}_\sigma(g_j), \mathrm{LT}_\sigma(g_k))$
  $S_{jk} \leftarrow \frac{t}{\mathrm{LM}_\sigma(g_j)} \cdot g_j - \frac{t}{\mathrm{LM}_\sigma(g_k)} \cdot g_k$
  $R \leftarrow \mathrm{NR}_\sigma(S_{jk}, G)$ ▷ Using Algorithm 8
  if $R \neq 0$ **then**
    $G \leftarrow G \cup \{R\}$
    $r \leftarrow \operatorname{len}(G)$
    $B \leftarrow B \cup \{[j, r] \mid 1 \leq j < r\}$
  end if
end while
return G

Proof. Refer to Kreuzer and Robbiano [65] and Buchberger [55, 56]. □

Remark 53. Intuitively, a Gröbner basis is a set of polynomials that provides a systematic way to solve polynomial equations. It encapsulates the essential information about the solutions of a polynomial system, helping to determine whether a solution exists and to find it efficiently. It is like having a simplified representation of the equations that reveals their underlying structure, making them easier to analyse and solve.

Back to Cyclic Codes...

Now that the basics have been introduced, it is time to propose the decoding algorithm for cyclic codes which is based on Gröbner bases.

Definition 30. Named after its inventor Cooper [64, 66], we introduce the *Cooper system*

$$\mathrm{Cooper}_{q,r,w}(x, z) \coloneqq \begin{cases} \sum_{l=1}^{w} z_l x_l^{i_u} = s_{i_u} & 1 \leq u \leq r \\ x_l^n = 1 & 1 \leq l \leq w \\ z_l^q = z_l & 1 \leq l \leq w \\ x_i x_j p(n, x_i, x_j) & 1 \leq i < j \leq w \end{cases}$$

where $\alpha \in \mathbb{F}_{q^n}$ is the corresponding q^n-th root unity (*i.e.*, a primitive element) of the $[n, k, d]$-cyclic code C, $r = n - k$, $i_u \in \{i \in \{1, \ldots, q^n - 1\} \mid g(\alpha^i) = 0\}$ the defining set of C, generator polynomial $g(x)$ of C, s_{i_u} the syndrome $m(\alpha^{i_u})$ of the received message $m(x)$, and $p(n, x, y) = \frac{x^n - y^n}{x - y}$.

Lemma 54. *In the context of a cyclic code C, the solution to the Cooper system from Definition 30 describes the unique error vector of a given message $m(x)$.*

Proof. A formal proof for this can be found in the works of Cooper [64] and Imran [66]. Here, only a "hand-waving" argument is given.

The variables x_i correspond to the error locators and the variables z_j to the error values. Each line in the system from Definition 30 corresponds to a different property of cyclic codes:

- line 1 contains all the information about the generator polynomial and codeword polynomial in terms of the defining syndromes s_{i_u},
- line 2 ensures that the error vector consists of only n words, so if the error vector is shifted n times, it should not change,
- line 3 ensures that each error value is non-zero and hence invertible in the field $\mathbb{F}_q$ and
- line 4 ensures that all the x_i are pairwise distinct.

Overall, the unique solution to the Cooper system thus describes the transmission error in terms of the variables x_i and z_j under the assumption that w errors have occurred. □

Theorem 55 (Decoding of cyclic codes – using Gröbner bases)**.** *Algorithm 10 allows decoding of cyclic codes using Gröbner bases.*

Algorithm 10 Decoding of cyclic codes using Gröbner bases, adapted from Cooper [64] and Imran [66].

Require: $[n, k, d]$-cyclic code C with primitive element $\alpha \in \mathbb{F}_{q^n}$ and generator polynomial $g(x) \in \mathbb{F}_q[x]$ and received message $m(x)$

$J \leftarrow \left\{i \in \{1, \ldots, q^n - 1\} \mid g(\alpha^i) = 0\right\}$
$S \leftarrow \left\{m(\alpha^j) \mid j \in J\right\}$
if $\forall s \in S \colon s = 0$ **then return** $m(x)$
end if
$w \leftarrow 1$
do
  $I \leftarrow \langle \text{Cooper}_{q,r,w}(x, z) \rangle$ ▷ See Definition 30
  $w \leftarrow w + 1$
while $1 \in I$
$G \leftarrow \text{GBasis}(I)$ ▷ Using Algorithm 9
$p(x_1) \leftarrow$ Unique polynomial from G which only depends on x_1
$roots \leftarrow \{a \mid p(a) = 0\}$
Obtain error polynomial $e(x)$ from $roots$ and the solutions to the unique polynomials which only depend on z_i each
return $m(x) - e(x)$

Proof. Again, a formal proof for this can be found in the works of Cooper [64] and Imran [66]. The intuition behind this algorithm is that it is assumed first that the message $m(x)$ only has a single error. If the Cooper system generates the ideal containing 1 (that is, the entire field), the Cooper system has no solution. After that, it is assumed that the message $m(x)$ has two errors, and so on until the ideal is no longer the entire field. At that point, the solution can be read from a Gröbner basis of that ideal by translating it into the error polynomial. It is important to note that the Gröbner basis must be constructed using an elimination order with $X_{\text{elim}} = \{x_2, \ldots, x_w, z_1, \ldots, z_w\}$ to obtain the unique polynomial $p(x_1)$. □

This algorithm provides a rather simple way of correcting general cyclic codes. There is one possible optimisation in the binary case, though.

Remark 56. Let $e(x)$ be a binary error vector. Then, obviously, the error values of $e(x)$ can only be 1 since they have to be nonzero. For this reason, Definition 30 can be simplified by substituting $z_i \mapsto 1$ for every $i \in \{1, \ldots, w\}$. This change also affects Algorithm 10, where every instruction related to any z_i can be ignored or shortened.

Up until now, we only assumed messages to have errors that occur in an independently uniformly distributed manner, *i.e.*, every codeword position has the same probability of experiencing an error. In practise, this is often not the case, as errors might tend to "condense" in a shorter area. Cyclic codes can be used to handle such cases with special care. However, again, a new notation needs to be introduced first.

Definition 31. Let $e(x)$ be an error polynomial. $e(x)$ is a *b-burst error* if its nonzero coordinates are confined to b consecutive positions [7]. If a code C is able to correct any b-burst error, it is called a *b-Burst Error Correction Code (BECC).*

Remark 57. Other definitions of burst errors are also possible, such as Hankerson *et al.* [19] who propose the following equivalent definition. Let $e(x)$ be an error polynomial. If it can be factored into $x^k e'(x) \pmod{x^n+1}$ where $e'(0) \neq 0$ (*i.e.*, its constant coefficient is nonzero) and $\deg(x^k e'(x))$ is minimum, then $e(x)$ is a $\deg(e'(x))+1$-burst error.

An efficient decoding algorithm which focuses on these burst errors instead of random errors can be derived from the following properties.

Definition 32. Let $m(x) = c(x) + e(x)$ be a received message with $c(x) \in C$ a codeword in the cyclic code C with the generator polynomial $g(x)$. Then, the polynomial $s_m(x) \coloneqq \mathrm{mod}(m(x), g(x))$ is called the *syndrome polynomial of* $m(x)$ [54].

Lemma 58. *Let C be a cyclic code with generator polynomial $g(x)$ and $m(x) = c(x) + e(x)$ be a received message with $c(x) \in C$. Then, the following properties hold:*

- $m(x) \in C \implies s_m(x) = 0$
- *Let $m_i(x)$ be the i-th cyclic right shift of $m(x)$. Then $s_{m_i}(x) = \mathrm{mod}(x^i \cdot s(x), g(x))$. Therefore, the syndromes of the cyclic shifts of $m(x)$ can be calculated iteratively.*

Proof. We prove both properties separately:

- If $m(x) \in C$, we have $e(x) = d(x)$ for some $d(x) \in C$ since otherwise $m(x) \notin C$. Therefore, $m(x) = c(x) + d(x) = g(x) \cdot v(x) + g(x) \cdot w(x) = g(x) \cdot (v(x) + w(x))$ for some $v(x), w(x)$. In conclusion, $s_m(x) = \mathrm{mod}(g(x) \cdot (v(x) + w(x)), g(x)) = 0$.
- We have $s_{m_i}(x) = \mathrm{mod}(m_i(x), g(x)) = \mathrm{mod}(x^i \cdot m(x), g(x)) = \mathrm{mod}(x^i \cdot \mathrm{mod}(m(x), g(x)), g(x)) = \mathrm{mod}(x^i \cdot s(x), g(x))$, according to Hankerson *et al.* [19]. □

Theorem 59 (Decoding of cyclic codes as BECCs)**.** *Algorithm 11 allows decoding of cyclic codes as BECCs.*

Algorithm 11 Decoding of cyclic codes as BECCs, adapted from Hankerson *et al.* [19].

Require: C cyclic b-BECC with generator polynomial $g(x)$ and received message $m(x)$

```
s(x) ← mod(m(x), g(x))
for j ≔ 0 to n − 1 do
    s_j(x) ← mod(x^j · s(x), g(x))
    if deg(s_j(x)) ≤ b then
        e(x) ← mod(x^{n−j} s_j(x), x^n + 1)
        return m(x) − e(x)
    end if
end for
```

Proof. Finiteness: The loop terminates after at most n iterations.

Correctness: The iterative shift approach has already been proven in Lemma 58. For the rest, Hankerson *et al.* [19] note that the first $n-k$ rows of the parity-check matrix of the code

form an identity matrix. Hence, $e(x) \cdot H = s_j(x)$ and finally, shifting $s_j(x)$ appropriately, the error pattern can be obtained. □

Finally, some remarks conclude the general description of cyclic codes.

Remark 60. There are many techniques that can be used to increase the burst error correction capability of any ECC. One of the simplest methods is called *interleaving*. In order to accomplish this, the codewords are not sent in order, but rather "in parallel." Let $c_1, \ldots, c_m$ be codewords and $c_{i,j}$ be the j-th value of c_i. Usually we would transmit

$$c_{1,1}, \ldots, c_{1,n}, c_{2,1}, \ldots, c_{2,n}, \ldots, c_{m,1}, \ldots, c_{m,n}$$

to the receiver. However, when interleaving, we instead send

$$c_{1,1}, \ldots, c_{m,1}, c_{1,2}, \ldots, c_{m,2}, \ldots, c_{1,n}, \ldots, c_{m,n}.$$

If a burst error occurs in this interleaved message, it does not affect only one codeword, but its effect gets divided into the m different codewords. It is easy to prove that (if a b-BECC was used) interleaving improves the burst error correction capability of the code to $b \cdot m$.

Remark 61. Actually, the Golay codes are also cyclic, but are rarely considered as such, which is the reason why they were only introduced as linear codes in this work. As already mentioned in Section 2.1.1.2, the Golay codes are also Quadratic Residue codes which are cyclic codes. According to MacWilliams and Sloane [11], the binary Golay code can be described using either generator polynomial

$$x^{11} + x^{10} + x^6 + x^5 + x^4 + x^2 + 1 \quad \text{or} \quad x^{11} + x^9 + x^7 + x^6 + x^5 + x + 1$$

and the ternary Golay code can be described using either generator polynomial

$$x^5 + x^4 - x^3 + x^2 - 1 \quad \text{or} \quad x^5 - x^3 + x^2 - x - 1.$$

As a result, they can also be decoded using Gröbner bases as in Algorithm 10 or using optimised versions for this specific use case [67].

Remark 62. Furthermore, some Hamming codes are cyclic, such as $\mathrm{Ham}(3, 2)$ which, according to van Lint [49] has the generator polynomial

$$x^3 + x + 1.$$

Moreover, according to MacWilliams and Sloane [11], if r and $q - 1$ are coprime, then $\mathrm{Ham}(r, q)$ is cyclic. In particular, in the binary case, since every $\mathbb{N} \ni r \geq 2$ is coprime to $q - 1 = 1$, every binary Hamming code is cyclic.

Lemma 63. *These properties have already been used before without properly deriving or proving them. Let C be a cyclic code with generator polynomial $g(x)$. Then we have*

- $\deg(g(x)) = n - k$
- $\deg(h(x)) = k$

Proof. This follows from Lemmata 3 and 4 and Theorem 42. □

2.1.2.1 BCH Codes

In 1959 and 1960, Hocquenghem [68] and Bose and Ray-Chaudhuri [69] independently [49] discovered the Bose-Chaudhuri-Hocquenghem (BCH) codes which are defined in this section. They have been used in deep space communications such as *e.g.* the Phobos lander [70].

Definition 33. Let p be a prime, $\alpha \in \mathbb{F}_{p^m}$ be a primitive n-th root of unity, $\delta \in \mathbb{N}$ with $1 \leq \delta \leq n$, $l \in \mathbb{N}$, and finally $\mu_\beta(x) \in \mathbb{F}_p[x]$ denote the respective minimal polynomial of some $\beta \in \mathbb{F}_{p^m}$. Then, the *BCH code with designed distance* δ has the generator polynomial

$$g(x) \coloneqq \operatorname{lcm}(\mu_{\alpha^l}(x), \ldots, \mu_{\alpha^{l+\delta-2}}(x)).$$

The parameter l is most often unnamed and is in this work referred to as the *sense* of the BCH code. If $l = 1$, then the BCH code is said to be *narrow-sense* and if α is a primitive element of $\mathbb{F}_{p^m}$, then the BCH code is said to be *primitive* [49].

The name of the parameter δ already gives us a clue to its purpose.

Theorem 64 (BCH code distances)**.** *Let C be a BCH code with the designed distance δ. Then, $d(C) \geq \delta$.*

Proof. Without going too far into theory, this can be proven through relations to Alternant codes [14], through Fast Fourier Transform (FFT) and Mattson-Solomon polynomials [49] or by calculation of Vandermonde determinants [11, 49, 71]. □

Obviously, BCH codes can be encoded in the same way as general cyclic codes using Corollary 45 and Theorem 48. However, there are many other implementations such as hardware encoders [72] and even parallelised Field Programmable Gate Array (FPGA) procedures [73].

In terms of decoding, Algorithms 10 and 11 are applicable. On the other hand, it is beneficial to use other specialised decoders such as the Berlekamp-Massey algorithm [74, 75, 76] which finds a Linear-Feedback Shift Register (LFSR) of shortest length generating a certain sequence. This is equivalent to finding the best error locator polynomial due to the syndrome calculation based on Newton's identities [77]. Other decoders include the Extended Euclidean Algorithm (EEA) [78] and the Peterson-Gorenstein-Zieler algorithm [79, 80, 16] which is the focus of this section.

According to Moon [77], algebraic decoding using the Peterson-Gorenstein-Zierler algorithm actually consists of four different procedures.

1. *Syndrome calculation*: Using Definition 32 or polynomial evaluation.
2. *Peterson-Gorenstein-Zierler algorithm* [79, 80, 16]: Finding the error locator polynomial (polynomial whose roots are the error positions of the received message). This step could also be accomplished using the other aforementioned procedures [74, 75, 76, 77, 78].
3. *Chien search* [81]: Finding the roots of the error locator polynomial. Other possible procedures could be simple brute force, Berlekamp's algorithm [82] or the Cantor-Zassenhaus algorithm [83].
4. *Forney's algorithm* [84]: Finding the error values (this can be skipped in the binary case, as the only possible error value is 1). This can also be achieved through the simple solving of a linear system of equations.

For the Peterson-Gorenstein-Zierler algorithm, in total $2t$ syndromes need to be computed, where t is the error correction capability of the BCH code, *i.e.*, $t = \lfloor \frac{d-1}{2} \rfloor$ as described in Lemma 1. For d, it is enough to use δ, the designed distance of the BCH code. However, if the exact minimum distance d of the BCH code is known, more errors could potentially be corrected/detected (see Theorem 64).

Algorithm 12 Syndrome calculation for the Peterson-Gorenstein-Zierler algorithm.

Require: Received message w, originally encoded using the BCH code C with minimum distance $\geq d$ and sense l

$t \leftarrow \lfloor \frac{d-1}{2} \rfloor$
return $[s_l, \ldots, s_{l+2t-1}] = \left[s_i = w(\alpha^i) \mid i \in \{l, \ldots, l+2t-1\}\right]$

Remark 65. If the returned list of syndromes from Algorithm 12 only contains 0s, the input is already a codeword and the decoding procedure is already complete, as explained in Lemma 58.

The input polynomial w is now assumed to be nonzero and the goal is to find the error locator polynomial. For this, a few definitions and lemmata are needed.

Definition 34. Analogously to Jungnickel [8], we denote by $X_j \coloneqq \alpha^{i_j}$ the *error locator* and by $Y_j \coloneqq e_{i_j}$ the *error value* for the error at position i_j.

Lemma 66. *According to Moon [77], let $m(x) = c(x) + e(x)$ be a received message with v errors and $c(x) \in C$ a codeword in a BCH code. Denote the error values (that is, the nonzero coefficients of $e(x)$) with e_{i_j} where $j \in \{1, \ldots, v\}$. Then we have*

$$S_k = \sum_{j=1}^{v} Y_j X_j^k.$$

Proof. We obtain $S_k = m(\alpha^k) = \underbrace{c(\alpha^k)}_{=0} + e(\alpha^k) = e(\alpha^k) = \sum_{j=1}^{v} e_{i_j} \alpha^{k \cdot i_j} = \sum_{j=1}^{v} Y_j X_j^k$. □

Definition 35. Similarly to Moon [77], we define the *error locator polynomial* $\Lambda(x)$ as follows:

$$\Lambda(x) \coloneqq \Lambda_0 + \Lambda_1 x + \cdots + \Lambda_v x^v \coloneqq \prod_{j=1}^{v} (1 - \alpha^{i_j} x) = \prod_{j=1}^{v} (1 - X_j x).$$

Hence, clearly the roots of $\Lambda(x)$ are the α^{-i_j}, *i.e.*, the reciprocals of the error locators X_j^{-1}.

Lemma 67. *According to Jungnickel [8], in the setting of Lemma 66, using all known values of S_k, they form a system of equations where every equation is of the form*

$$S_{k+v} + \Lambda_1 S_{k+v-1} + \cdots + \Lambda_v S_k = 0.$$

Proof. This statement is proven through a generalised Newton's identities approach in the works by Jungnickel [8], Moon [77] and Cooper [64]. □

Remark 68. The system of equations from Lemma 67 can be re-written in matrix from as

$$\begin{bmatrix} S_1 & S_2 & \cdots & S_v \\ S_2 & S_3 & \cdots & S_{v+1} \\ \vdots & \vdots & \ddots & \vdots \\ S_v & S_{v+1} & \cdots & S_{2v-1} \end{bmatrix} \cdot \begin{bmatrix} \Lambda_v \\ \Lambda_{v-1} \\ \vdots \\ \Lambda_1 \end{bmatrix} = \begin{bmatrix} -S_{v+1} \\ -S_{v+2} \\ \vdots \\ -S_{2v} \end{bmatrix}$$

This equation is the key to the Peterson-Gorenstein-Zierler algorithm. However, since we do not know beforehand how many errors the received message has, some additional work is needed. Hence, one last property is needed for it to work.

Lemma 69. *Let M be the leftmost matrix of Remark 68. According to Jungnickel [8], it possesses the following properties:*

1. *M is invertible if v is the amount of errors (*i.e., *$v = \mathrm{wt}(e)$)*
2. *M is singular if v is greater than the amount of errors*

Proof. According to Jungnickel [8], this can be proven using the decomposition of M into a Vandermonde matrix and a diagonal matrix. □

Because of this, the Peterson-Gorenstein-Zierler algorithm starts iterating from t (recall the error correction capability of the code) down to 1, as stated below.

Theorem 70 (Peterson-Gorenstein-Zierler algorithm)**.** *The error locator polynomial of $m(x) = c(x) + e(x)$ with $c(x) \in C$ a codeword in a BCH code C can be calculated using Algorithm 13.*

Algorithm 13 Peterson-Gorenstein-Zierler algorithm.

Require: A BCH code C with error correction capability $\geq t$ and a message $m(x)$, originally encoded using C

$v \leftarrow t$
do
  $M \leftarrow \begin{bmatrix} S_1 & S_2 & \cdots & S_v \\ S_2 & S_3 & \cdots & S_{v+1} \\ \vdots & \vdots & \ddots & \vdots \\ S_v & S_{v+1} & \cdots & S_{2v-1} \end{bmatrix}$ ▷ as in Remark 68
  if $\det(M) = 0$ **and** $v = 1$ **then**
    return *Cannot decode!*
  end if
  $v \leftarrow v - 1$
while $\det(M) = 0$ **and** $v > 0$
$w \leftarrow v + 1$
$\begin{bmatrix} \Lambda_w \\ \Lambda_{w-1} \\ \vdots \\ \Lambda_1 \end{bmatrix} \leftarrow M^{-1} \cdot \begin{bmatrix} -S_{w+1} \\ -S_{w+2} \\ \vdots \\ -S_{2w} \end{bmatrix}$ ▷ or simple solving of the system from Lemma 67
return $\Lambda(x) = 1 + \Lambda_1 x + \cdots + \Lambda_w x^w$ ▷ as in Definition 35

Proof. Finiteness: The algorithm must terminate since v gets decreased in every step and $\det(M) = 0$ in the end, and the algorithm throws an error or $\det(M) \neq 0$ and an error locator polynomial is returned.

Correctness: Follows from Lemmata 67 and 69 and Remark 68. □

After using Algorithm 13, we have the error locator polynomial, from which the roots need to be extracted. In theory, any polynomial factorisation algorithm can do this, however, in practise the Chien search algorithm [81] is used most often. In principle, it is a systematic brute-force algorithm which makes use of a primitive element of the underlying field.

Theorem 71 (Chien Search algorithm)**.** *Finding the roots of a polynomial over some finite field extension can be achieved using Algorithm 14.*

Algorithm 14 Chien search root finding algorithm.

Require: Polynomial $f(x) \in F[x]$ over some finite field extension F with m elements and $\alpha \in F$ a primitive element

```
L ← [f_0, ..., f_n]
R ← {}
for i ≔ 0 to m do
    if Σ_{k=0}^{n} f_k = 0 then
        R ≔ R ∪ {i}
    end if
    for j ≔ 0 to n do
        L[j] ← L[j] · α^j
    end for
end for
return R
```

Proof. Finiteness: Again, the algorithm obviously terminates since there are only a finite number of elements that this "smart brute-force" approach needs to check.

Correctness: We note that in the first iteration, the polynomial is evaluated at $\alpha^0 = 1$, where we have $f(1) = \sum_{k=0}^{n} f_k$ since $1^k = 1$ for each k. Depending on the result, 0 is added to the set R that contains the exponents of α that are a root of $f(x)$. Then, for each $0 \leq j \leq n$, the j-th element from L (that is, the coefficients of f in ascending order), is multiplied by α^j. After this step, $\sum_{l \in L} l = f(\alpha)$. In fact, after each iteration, this power of α at which $f(x)$ is evaluated increases by 1 as

$$f(\alpha^{k+1}) = \sum_{j=0}^{n} f_j \cdot (\alpha^{k+1})^j = \sum_{j=0}^{n} f_j \cdot (\alpha^k)^j \cdot \alpha^j. \qquad \square$$

Remark 72. Depending on the use case, instead of appending each "root power" to R, Algorithm 14 could also be modified to find only a single root by returning the current value of i instead. Although, in the case of the error locator polynomial, all roots are needed.

Now that the Chien search algorithm has been properly introduced, the only part of the BCH decoding procedure that has not yet been explained is the last step: Forney's algorithm [84]. However, as said before, since it is not necessary in the binary case, it is not introduced in this work. In conclusion, the binary BCH decoding procedure goes as follows.

Theorem 73 (Decoding of binary BCH codes)**.** *Decoding binary BCH codes can be accomplished using Algorithm 15.*

Algorithm 15 BCH decoding procedure using the Peterson-Gorenstein-Zierler algorithm.

Require: A BCH code C over some finite field extension F with m elements and a message $m(x)$, originally encoded using C

```
S ← list of needed syndromes                          ▷ using Algorithm 12
Λ(x) ← error locator polynomial                       ▷ using Algorithm 13
R ← set of root powers of Λ(x)                        ▷ using Algorithm 14
e(x) ← Σ_{r∈R} x^{m−r−1}   ▷ since R contains the reciprocals of the error locators
return m(x) − e(x)
```

Proof. Finiteness: The algorithm obviously terminates, since all the subroutines terminate.

Correctness: Follows from Definition 35 and Theorems 70 and 71. $\square$

Remark 74. According to Schulz [15], the binary Golay code G_{23} can also be described as a narrow-sense binary BCH code of length $n = 23$ and designed distance 5 (although the resulting minimum Hamming distance of the code is 7) based on a primitive 11-th root of unity.

2.1.2.2 Reed-Solomon Codes

A large subgroup of BCH codes is the so-called Reed-Solomon (RS) codes, which were invented by Reed and Solomon [85] in 1960. They were originally thought to be a new class of ECCs, where each codeword is constructed by evaluating a polynomial of degree $< k$, which defines the given RS code, at k different points. However, nowadays, it is known that they are just a special case of primitive BCH codes. However, there exist additional, more efficient, encoding and decoding procedures for them, such as the Berlekamp-Massey algorithm [75, 76]. Nowadays, RS codes are still widely used and one of the most popular choices for ECCs. Examples include, but are not limited to Compact Discs (CDs), Digital Versatile/Video Discs (DVDs) [15, 86] and Quick Response (QR) codes [87].

Definition 36. Let $q > 2$ be a prime power and C a primitive BCH code of length $n = q-1$ on $\mathbb{F}_q$. Then, C is a *RS code* [15].

According to this definition, every RS code is also a BCH code, but not vice versa. Therefore, everything from Section 2.1.2.1 still applies to the RS codes, including encoding and decoding procedures.

Example 3. QR codes use 36 different RS codes on $\mathbb{F}_{2^8}$, depending on their size and correction capability needed [87]. The corresponding Galois field $\mathbb{F}_{2^8}$ is defined as the splitting field of the Conway polynomial

$$x^8 + x^4 + x^3 + x^2 + 1.$$

Remark 75. It can be shown that RS codes are *Maximum Distance Separable (MDS) codes*, which means that they have an optimal distance of $d = n - k + 1$, according to the *Singleton bound* [88].

2.1.3 Modification of Codes

Each ECC can also be modified, either to combine the capabilities of two codes into one, or to adjust the parameters of a code slightly. Some of these modifications are presented in this section.

Definition 37. According to van Lint [49], the *extended code* $\widetilde{C}$ of any code C is defined as

$$\widetilde{C} \coloneqq \{[c_1, \ldots, c_n, -\textstyle\sum_{i=1}^{n} c_i] \mid [c_1, \ldots, c_n] \in C\}.$$

Hence, it contains exactly the same codewords as C, but each with an extra parity bit at the end.

Lemma 76. *Let C be a $[n, k, d]$ code. Then, $\widetilde{C}$ is a $[n+1, k, \widetilde{d}]$ code with $\widetilde{d} \in \{d, d+1\}$, according to Huffman and Pless [7].*

Proof. Obviously, the block length increases by 1 due to the extra parity bit, and its dimension remains unchanged because the parity bit can be derived from the other bits by linear combination. The minimum distance can only increase by 1 at most, because either the parity bit is the same or different, while the other bits remain unchanged between two codewords with a minimum distance of C. □

Remark 77. The respective extended Golay code G_{12} or G_{24} can also be defined as $\widetilde{G_{11}}$ or $\widetilde{G_{23}}$.

Definition 38. Let $r \coloneqq n - k$. Then, a list of the most common modifications of codes is given as follows [6]:

- *Extending*: Simultaneously increase n and r while fixing k
- *Puncturing*: Simultaneously decrease n and r while fixing k
- *Augmenting*: Simultaneously increase k and decrease r while fixing n
- *Expurgating*: Simultaneously increase r and decrease k while fixing n
- *Lengthening*: Simultaneously increase n and k while fixing r
- *Shortening*: Simultaneously decrease n and k while fixing r

Extension is done as described in Definition 37. The other modifications are described in the lecture notes by Hall [6].

As well as these simple modifications to individual codes, there are also modifications that combine the capabilities of two codes.

Definition 39. Let C_{in} be a $[n_1, k_1, d_1]$ code over an alphabet A_1 and C_{out} be a $[n_2, k_2, d_2]$ code over an alphabet A_2 of size $|A_2| = |A_1|^{k_1}$. Forney [89] defines a *concatenated code*

$$C = \{C_{\text{out}} \mid C_{\text{in}}\}$$

which has the parameters $[n_1 n_2, k_1 k_2, d]$ where $d \geq d_1 d_2$. Its encoding procedure takes a message $m = [m_1, \ldots, m_{k_2}]$ and encodes it to

$$\left[C_{\text{in}}(m'_1), \ldots, C_{\text{in}}(m'_{n_2})\right] \quad \text{where} \quad \left[m'_1, \ldots, m'_{n_2}\right] = C_{\text{out}}([m_1, \ldots, m_{k_2}]).$$

C_{out} and C_{in} are called the *outer code* and *inner code*, respectively [7].

The code construction from Definition 39 is also illustrated in Figure 2.10.

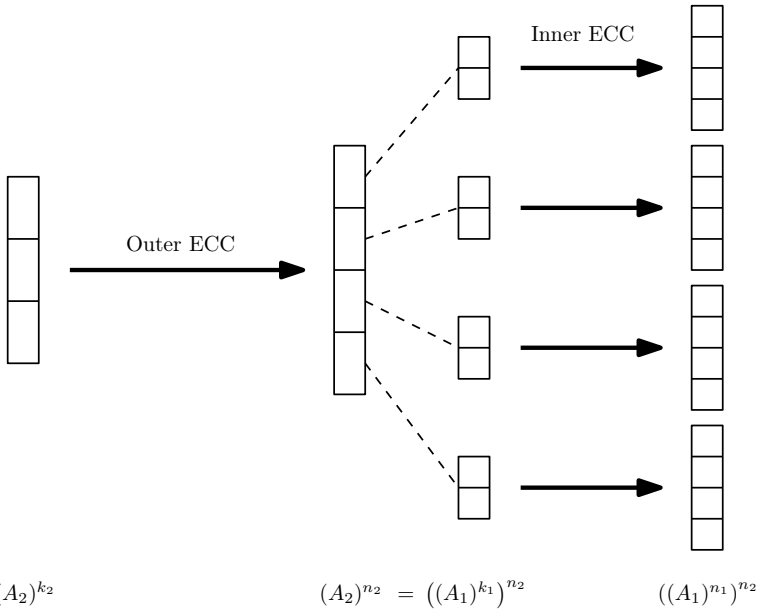


Figure 2.10: Combining two codes using Forney's concatenated codes scheme [89], adapted from Ylloh [90].

In addition, several generalisations of these concatenated codes have been proposed. When *Generalised Concatenated (GC) codes* are referred to in this work, the scheme by Blokh and Zyablov [91] is meant.

2.1.4 The Shannon limit

One of the most famous limits in coding theory is the *Shannon limit* which is derived in this section. The Shannon limit implies a limit on the amount of noise that can be present in a channel through which information is to be transmitted, while still allowing error-free communication. However, first the *Shannon-Hartley Theorem* is described in Theorem 78 and Remark 79 [24].

Theorem 78 (Shannon-Hartley Theorem)**.** *Suppose, one wants to send information with signal power S through a Binary-Input Additive-White-Gaussian-Noise (BI-AWGN) channel with a bandwidth of W in* Hz *and noise power N. Then, the channel capacity C in $\frac{\text{bits}}{\text{s}}$ is given as*

$$C = W \log_2\left(1 + \frac{S}{N}\right).$$

Proof. A proof can be found in Shannon's 1948 work [24]. □

Remark 79. The equality in Theorem 78 can obviously also be described using the inequality

$$R < W \log_2\left(1 + \frac{S}{N}\right)$$

where R is any achievable information bit rate in $\frac{\text{bits}}{\text{s}}$.

There are also new notations to introduce.

Definition 40. The *Signal-to-Noise Ratio (SNR)* of a noisy signal is defined as $\frac{S}{N}$ where S is the signal power and N is the noise power in the same unit as S.

Definition 41. The *spectral efficiency* η is defined as $\eta \coloneqq \frac{R}{W}$ where R is the information rate in $\frac{\text{bits}}{\text{s}}$ and W is the bandwidth of channel in Hz.

Definition 42. The *normalised SNR* $\frac{E_b}{N_0}$ is defined through the *average energy per bit* E_b and *noise spectral density* N_0 which are in turn defined as $E_b \coloneqq \frac{S}{R}$ and $N_0 \coloneqq \frac{N}{W}$. Thus, $\frac{E_b}{N_0} = \frac{S \cdot W}{R \cdot N}$.

From now on, the previous definitions of values such as C and W are consolidated and will not be repeated.

Lemma 80. *We have*

$$\frac{S}{N} = \frac{E_b}{N_0} \cdot \eta.$$

Proof. Using Definition 42, we obtain $\frac{E_b}{N_0} = \frac{S \cdot W}{R \cdot N}$ which is equivalent to $\frac{S}{N} = \frac{E_b}{N_0} \cdot \frac{R}{W}$ and hence, using Definition 41, we obtain $\frac{S}{N} = \frac{E_b}{N_0} \cdot \eta$. □

Now, we are able to derive a "better" formula for the Shannon-Hartley Theorem.

Theorem 81 (Shannon-Hartley using spectral efficiency)**.** *We always have*

$$\frac{E_b}{N_0} > \frac{2^\eta - 1}{\eta}.$$

Proof. Using Remark 79 and Definition 41, we have $\eta = \frac{R}{W} < \log_2\left(1 + \frac{S}{N}\right)$ and hence, $2^\eta - 1 < \frac{S}{N}$. Finally, applying both Definition 41 and Lemma 80, we obtain $2^\eta - 1 < \frac{E_b}{N_0} \cdot \eta \iff \frac{E_b}{N_0} > \frac{2^\eta - 1}{\eta}$. □

Using this formula, the Shannon limit can be easily determined. According to Viswanathan [29], the *ultimate Shannon limit* is reached when the spectral efficiency approaches 0, *i.e.*, the minimum required energy per bit required at the transmitter for reliable communication.

Theorem 82 (Ultimate Shannon Limit)**.** *The ultimate Shannon limit is*

$$\frac{E_b}{N_0} > \ln(2) \approx -1.59\,\text{dB}.$$

Proof. We need to calculate $\lim_{\eta \to 0}\left(\frac{2^\eta - 1}{\eta}\right)$. Since both numerator and denominator approach 0, L'Hôpital's rule can be applied to obtain

$$\lim_{\eta \to 0}\left(\frac{2^\eta - 1}{\eta}\right) = \lim_{\eta \to 0}\left(\ln(2) \cdot e^{\eta \ln(2)}\right) = \ln(2).$$

This value can then be converted to decibel to obtain $10 \log_{10}(\ln(2)) \approx -1.59\,\text{dB}$. □

However, in the binary communication case, a tighter bound can be derived.

Theorem 83 (Binary Shannon Limit)**.** *The binary Shannon limit is*

$$\frac{E_b}{N_0} > 1 = 0\,\text{dB}.$$

Proof. When transmitting binary information, $\eta = 1$ can be fixed and hence, the binary Shannon limit is exactly $\frac{E_b}{N_0} > \frac{2^1 - 1}{1} = 1 = 0\,\text{dB}$ □

2.2 Fuzzy Extractors

In 1999, Juels and Wattenberg [92] introduced the concept of *fuzzy commitment schemes*, which allow to fix errors in biometric data in a secure way by generating a secure key from them. A possible attacker has no practical way to reconstruct the real biometric data. This principle was further developed by Juels and Sudan [93] in the so-called *fuzzy vault* or by Dodis *et al.* [94, 95] as *fuzzy extractor*. All these authors propose a RS code as an underlying error correction procedure. This section is dedicated to explaining the principles behind these new schemes.

According to Mexis [5], the aim of a fuzzy extractor is to provide data with redundant bits and then correct possible errors in data transmission or collection. The former can be caused, for example, by cable breaks, induction, or other hardware-related circumstances, while the latter can already occur during data acquisition due to inaccuracy. With biometric data, such as fingerprints, it is also possible for physical inaccuracies to occur, such as change over time or injury. These small errors can be corrected by a fuzzy extractor. Another example of a physical change can be found in PUFs, which often change over time, *i.e.*, age, or depend on some other factors such as temperature, humidity, or the like [96, 97].

In principle, a fuzzy extractor is based on a fixed-length ECC. The redundant bits are supplemented here with the help of helper data, which must be calculated in advance. A very basic fuzzy extractor can be implemented by simply having the helper data represent the parity bits of the ECC. The helper data generation process is also known as the function

Gen, whose input w is an initial capture of the data and the first part of the output h is the helper data. The other output R forms the key generated or extracted from w. In the most fundamental fuzzy extractor, this function can be specified as follows:

$$\begin{aligned}\texttt{Gen}\colon w &\mapsto (h, R),\\ h &\coloneqq \mathsf{par}(\mathsf{enc}(w)),\\ R &\coloneqq w.\end{aligned}$$

Here enc, respectively dec represent the encoding or decoding process of the ECC and par the extraction of the parity bits of an encoded message. Based on this, the reproduction process can be defined as

$$\begin{aligned}\texttt{Rep}\colon (h, w') &\mapsto R',\\ R' &\coloneqq \mathsf{dec}([w' \mid h]),\end{aligned}$$

which takes a further collection of data w' and the previously generated helper data h as input in order to generate a key R' from it. This key R' now fulfils the central property of a fuzzy extractor

$$R = R' \quad \Longleftrightarrow \quad w \approx w',$$

which means that $R = R'$ holds exactly if the input w' is approximately equal to the input w of the generation process. Here, the tolerance completely depends on the ECC. Therefore, not too many errors should be correctable, as otherwise a lot of incorrect data could be classified as correct. On the other hand, also not only too few errors should be correctable, as otherwise some correct data could be classified as wrong. The tolerance should therefore be adapted to the application, whereby a few test runs can be very helpful to be able to estimate a standard deviation.

The big disadvantage of this implementation is that the output together with the ECC is sufficient to reverse-engineer or even compute the correct input. This is due to the fact that the helper data can be extracted directly from the output. For this reason, a *strong extractor* is typically used. It takes an input x and produces an output y that is distributed as uniformly as possible over all x. Hash functions, which often fulfil this property and additionally are one-way functions, are often suitable for this purpose. This means that the input cannot be calculated back just from the output of the hash function.

Another disadvantage of this implementation is that it requires the helper data to be generated systematically. In order to further protect the parity bits that are in the helper data, randomly generated bit sequences are often used in practice, which are first added bit by bit to the helper data. Thus, the parity bits cannot be read directly from the helper data.

The implementation by Kang *et al.* [98] uses three additional procedures, defined as

$$\begin{aligned}\texttt{SS}\colon (w, K) &\mapsto s = \mathsf{enc}(K) + w,\\ \texttt{Ext}\colon (w, x) &\mapsto R = \mathsf{hash}(w + x),\\ \texttt{Rec}\colon (s, w') &\mapsto w = s + \mathsf{enc}(\mathsf{dec}(s + w')),\end{aligned}$$

in order to define a working fuzzy extractor. Here enc, respectively dec are the encoding or decoding process of the ECC and hash is a hash function, more precisely Secure Hash Algorithm (SHA)-256 [99]. The ECC used in their work is a BCH code, which can be replaced by any other ECC. The abbreviations used above stand for *Secure Sketch*, *(strong) Extractor*, and *Recovery*, respectively. From this, the Generation and Reproduction process can be defined as follows, using two randomly generated values:

$$\begin{aligned}&x \coloneqq \mathsf{rand}(),\ K \coloneqq \mathsf{rand}(),\\ &\texttt{Gen}\colon w \mapsto ((\texttt{SS}(w, K), x), \texttt{Ext}(w, x)),\\ &\texttt{Rep}\colon ((s, x), w') \mapsto \texttt{Ext}(\texttt{Rec}(s, w'), x).\end{aligned} \tag{2.1}$$

The short-term main goal is now to prove that this proposed scheme is a working fuzzy extractor. For this, one lemma is needed.

Lemma 84. *Let e be a error vector and* enc *and* dec *the encoding and decoding procedures of any ECC with error correction capability t. Then, we have*

$$\mathsf{dec}(\mathsf{enc}(K)+e)\begin{cases}=K & \text{if } \mathrm{wt}(e)\leq t\\ \neq K & \text{if } \mathrm{wt}(e)\gg t\end{cases}$$

Proof. The equation follows easily from Lemma 1. Conversely, the inequality follows from the fact that the ECC can only correct at most $t+\varepsilon$ errors. Here, ε defines a certain region of uncertainty, where the ECC could still be able to correct a given word even though its distance to a codeword is beyond the error correction capability t. An example of this phenomenon is the word $\mathtt{00}$ in the context of the non-linear binary code $C \coloneqq \{\mathtt{01},\mathtt{11}\}$. Even though it has a minimum Hamming distance of 1 and hence, according to Lemma 1, has only an error correction capability of 0, it is still clear that the original codeword was most likely $\mathtt{01}$. □

Definition 43. A fuzzy extractor using *Code-Offset (CO) construction* chooses a random codeword c in some underlying ECC and stores the difference between c and the given message w in the helper data. This offset is always added to a new message w' for error correction.

Theorem 85 (Fuzzy extractor)**.** *The procedures* $\mathtt{Gen}$ *and* $\mathtt{Rep}$ *from Equation* (2.1) *define a fully working fuzzy extractor with generation phase* $\mathtt{Gen}$ *and reproduction phase* $\mathtt{Rep}$ *using CO construction.*

Proof. Let it be given that a response R has already been successfully generated from $\mathtt{Gen}(w)$. Also, let $w' \coloneqq w+e$ be new data which should be extracted. First, let $\mathrm{wt}(e)$ be small enough so that the ECC can correct the errors. Then, we have

$$\begin{aligned}
R' &= \mathtt{Rep}((s,x),w')\\
&= \mathtt{Ext}(\mathtt{Rec}(s,w'),x)\\
&= \mathsf{hash}(\mathtt{Rec}(s,w')+x)\\
&= \mathsf{hash}(s+\mathsf{enc}(\mathsf{dec}(s+w'))+x)\\
&= \mathsf{hash}(s+\mathsf{enc}(\mathsf{dec}(s+w+e))+x)\\
&= \mathsf{hash}(s+\mathsf{enc}(\mathsf{dec}(\mathsf{enc}(K)+e))+x) && \triangleright\ s = \mathsf{enc}(\mathsf{K})+\mathsf{w}\\
&= \mathsf{hash}(s+\mathsf{enc}(K)+x) && \triangleright\ \text{Lemma 84}\\
&= \mathsf{hash}(w+x) && \triangleright\ s = \mathsf{enc}(\mathsf{K})+\mathsf{w}\\
&= \mathtt{Ext}(w,x)\\
&= R.
\end{aligned}$$

Thus, if $\mathrm{wt}(e)$ is too large, according to Lemma 84, we instead have

$$\mathsf{hash}(s+\mathsf{enc}(\mathsf{dec}(\mathsf{enc}(K)+e))+x)\neq\mathsf{hash}(s+\mathsf{enc}(K)+x)\quad\Longrightarrow\quad R'\neq R.$$

The described scheme is thus a correct fuzzy extractor. From the definition of $\mathtt{SS}$, it is also obvious that this scheme uses CO construction. □

Remark 86. This fuzzy extractor additionally features a strong extractor, namely the hash function, and the helper data is modified by random numbers. Therefore, the aforementioned problems of the most basic fuzzy extractor do not apply here.

2.2.1 Reverse Fuzzy Extractors

Since the reproduction process in fuzzy extractors involves decoding using the ECC, it is clearly the most computationally intensive step of the fuzzy extractor. In practice, devices that provide a PUF are often very resource-constrained. For this reason, in 2012, van Herrewege *et al.* introduced the so-called *reverse fuzzy extractors* which outsource the reproduction phase to the (typically) more powerful verifier, *i.e.* the other party in the communication process [100].

In Figure 2.11 there is a comparison between the conventional fuzzy extractors and the reverse fuzzy extractors. The *verifier* here is the device that wants to authenticate the PUF device. This means that the PUF device wants to prove, for example, that it really is the real device as the communication partner. This is done by means of biometric data, which in devices can be, for example, a PUF.

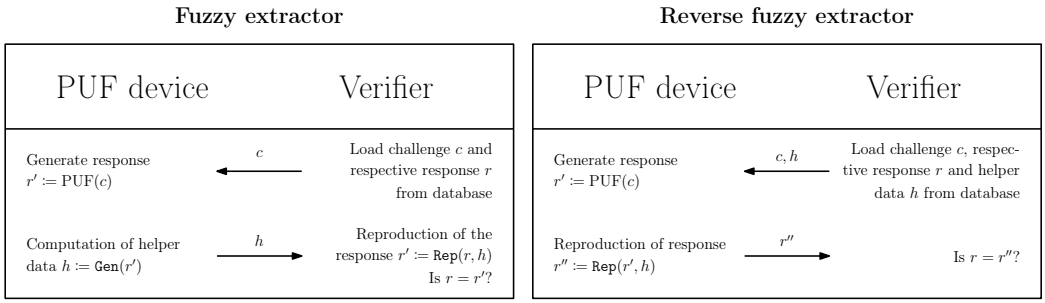


Figure 2.11: Comparison of the two fuzzy extractor types, adapted from Mexis [5] and van Herrewege *et al.* [100].

3 Related Work

3.1 Comparisons

In 2012, Singh and Singh [101] published a comparison of the different main types of error correction and detection. Error detection schemes can be further divided into repetition codes, parity-bit codes, checksum codes, Cyclic Redundancy Check (CRC) codes, and cryptographic hash function codes. On the other hand, error correction schemes consist of Automatic Repeat reQuest (ARQ) codes, forward error correction codes (same as ECCs defined previously), and hybrid-scheme codes.

The authors also mention the differences between systematic and non-systematic codes, as well as between random-error and burst-error-detecting/correcting codes. At the end of their work, they also present the RM error correction algorithm and possible use case scenarios. It is noted that systematic codes are more advantageous than non-systematic codes because the message bits can be immediately separated from the redundancy bits, as these two blocks are simply concatenated together to form the transmitted codeword. Furthermore, a distinction is made between error detection and error correction as two independent categories. In terms of systematic error detection, the encoding procedure could be utilised again to check for errors. Nevertheless, in many cases it is favourable to use a particular decoding algorithm, because it may offer more possibilities or be more efficient.

Next, they note that the type of communication channel being used might exhibit a specific error pattern. As explained earlier, if the channel is known to be susceptible to burst errors or random errors, the code should be able to handle them, respectively. The authors also mention that some codes can be used to correct both of these types of errors simultaneously, such as BCH codes.

Furthermore, error detection and error correction codes could be used as part of a larger scheme, such as ARQ or Hybrid Automatic Repeat reQuest (HARQ) codes. On the one hand, ARQ codes only require an error-detecting code to detect whether an error has occurred in the message and, if so, to request that the message be retransmitted. On the other hand, HARQ codes use a hybrid approach to combine the capabilities of ARQ and an ECC. Two possible approaches are the following:

1. Always transmit messages together with their parity data and request retransmission via ARQ only if decoding using the ECC is unsuccessful.
2. Send messages using ARQ and request parity data only if errors have been detected using an error-detecting code.

Of course, any ECC is also an error-detecting code, but there may be some overhead if the correction capability is used and discarded. The authors then compare the previously mentioned error-detecting and error-correcting procedures:

- Error detection:
  1. *Repetition codes:* (repeat bits n times), extremely simple, not efficient in terms of code rate, susceptible to burst errors.
  2. *Parity-bit codes:* (send a single parity bit), extremely simple, extremely efficient, only odd number of errors detectable.
  3. *Checksum codes:* (modular arithmetic sum, Luhn algorithm [102]), possible specific designs.
  4. *CRC codes:* (single-burst error-detecting cyclic code), easy and efficient specifically on hardware.
  5. *Cryptographic hash functions codes:* (hash function), efficiency depends on the hash function, additional protection against intentional modification by attackers.
- Error correction:
  1. *ARQ codes:* (use ACKnowledgement (`ACK`) and No ACKnowledgement (`NACK`) messages and timeouts), efficient when using a communication channel with varying or unknown capacity, have latency, timers needed.
  2. *Forward error correction codes (ECCs):* (use redundant data), no retransmission request needed, low latency, simple communication.
  3. *Hybrid-scheme codes:* (HARQ as explained above), combines all the advantages of ARQ and ECC.

Lastly, the authors note that RM codes are multiple-errors-correcting codes that can be decoded using majority logic decoding. Due to their characteristics, they are a good choice for long-distance transmissions. Additionally, the authors note that RM codes can also model other codes, such as repetition codes, by choosing a parameter r and using the specific codes $\mathrm{RM}(0, r)$. Parity check codes can be simulated with $\mathrm{RM}(1, r)$ or $\mathrm{RM}(r-1, r)$, and finally extended Hamming codes can be simulated with $\mathrm{RM}(r-2, r)$. For this reason, the same procedures can be used to correct all of these codes.

Then, in 2015, Puchinger *et al.* [103] published comparison results specifically in the context of PUFs by extending the implementation of their previous article [104]. In particular, apart from GC codes, their extension also includes RM and RS codes. All of them use Generalised Minimum Distance (GMD) and Maximum Likelihood (ML) decoding to improve the error correction capability. Furthermore, in 2012, Maes *et al.* [105] published a PUF-based key generator based on the concatenation of a BCH code and a repetition code. Additional results are obtained through practical tests of their scheme.

Puchinger *et al.* first state in the context of PUFs that it is important to distinguish between intra-device distance and inter-device distance. The former describes the Hamming distance between PUF responses of the same PUF instance, which can occur due to ageing, environmental variations, or noise, while the latter describes the Hamming distance between PUF responses of different PUF instances, which most often occur due to manufacturing variations. When working with PUF responses, the intra-device distance is quite small due to the robustness property. On the other hand, the inter-device distance will be close to 50% due to the unpredictability property. Subsequently, the ECC should be able to correct the error amount that lies in the open interval between 0% and at most 50%. Furthermore, the code dimension k should be greater than or equal to the length of the key, and the code should be easy to implement on the hardware. The aforementioned GC codes, RM codes, and RS codes

exhibit these properties. For the GC codes, ML decoding and the implementation explained by Bossert [106] were used, while for the RM codes, GMD decoding was used. Finally, for the RS codes, power decoding is used in order to profit from the code being a MDS code. Their results can be found in Table 3.1. Additionally, in the context of ECCs for PUFs, the authors differentiate between code-offset construction, syndrome construction, index-based syndrome coding, complementary index-based syndrome coding, and differential sequence coding. A fuzzy extractor is then defined using one of these key reproduction algorithms together with a strong extractor, that is, a hash function. However, a comparison of the different key reproduction algorithms is outside the scope of this work.

Table 3.1: Comparison between the codes by Puchinger *et al.*, taken from their paper [103].

Code	Length	P_{err}
BCH	2226	1.00×10^{-9}
GC RM	2048	5.37×10^{-10}
RS	2048	6.79×10^{-37}
RS	1152	1.19×10^{-10}
GC RS	1024	3.47×10^{-10}

In 2016, Wonterghem *et al.* [107] were able to compare four different types of short-length linear codes in terms of their performance, namely RM codes, Polar codes, BCH codes and Low-Density Parity-Check (LDPC) codes. Specifically in the context of fuzzy extractor schemes, LDPC codes have been studied in more detail by Mexis *et al.* [5, 108], and are restricted to codes whose girth is at least 6. Wonterghem *et al.* define short codes as being codes with a length of at most around 256 bits. This allows them to disregard typical phenomena that occur at larger code lengths. Also, the complexity of the investigated codes is not the main focus of this work, but rather the performance when using the same decoding algorithm. As mentioned above, the authors only used linear codes, so they opted for ML decoding and Ordered-Statistics Decoding (OSD), which can be applied to these codes. The former was applied to erroneous messages on a Binary Erasure Channel (BEC), which has many burst errors, and the latter to erroneous messages on a BI-AWGN channel, which has white noise errors. Obviously, the BEC channel can be used to simulate communication channels over long distances, while the BI-AWGN better resembles the behaviour of PUFs, and hence is more relevant for this work.

In addition to testing RM codes, Polar codes, BCH codes, and LDPC codes, the authors used a CRC code to further increase the error detection capabilities of the codes. Since CRC codes are cyclic and therefore linear, the generator matrix corresponding to the chosen CRC code with the generator polynomial

$$g(x) = x^{16} + x^{12} + x^5 + 1,$$

can be multiplied with the generator matrix of the inner code to form a new improved linear code with a higher Hamming distance and thus higher error correction capability. Their analysis shows that the BCH code is outperforming every other code in every measurement, as can be seen in Figure 3.1. In this figure, the y axis shows the Bit Error Rate (BER), which is the number of erroneous bits divided by the total number of bits after decoding with the respective code. It is also worth noting that the usual Belief Propagation (BP) decoding approach for LDPC codes appears to perform worse than the two approaches used mainly in the paper. Finally, the CRC code was able to improve all the codes examined, but now they seem to have almost the same error correction capabilities. This is due to the multiplication of the CRC generator matrix and the inner code generator matrix. The resulting matrix seems to belong to a randomly chosen code, and therefore all codes then seem to have almost the same properties.

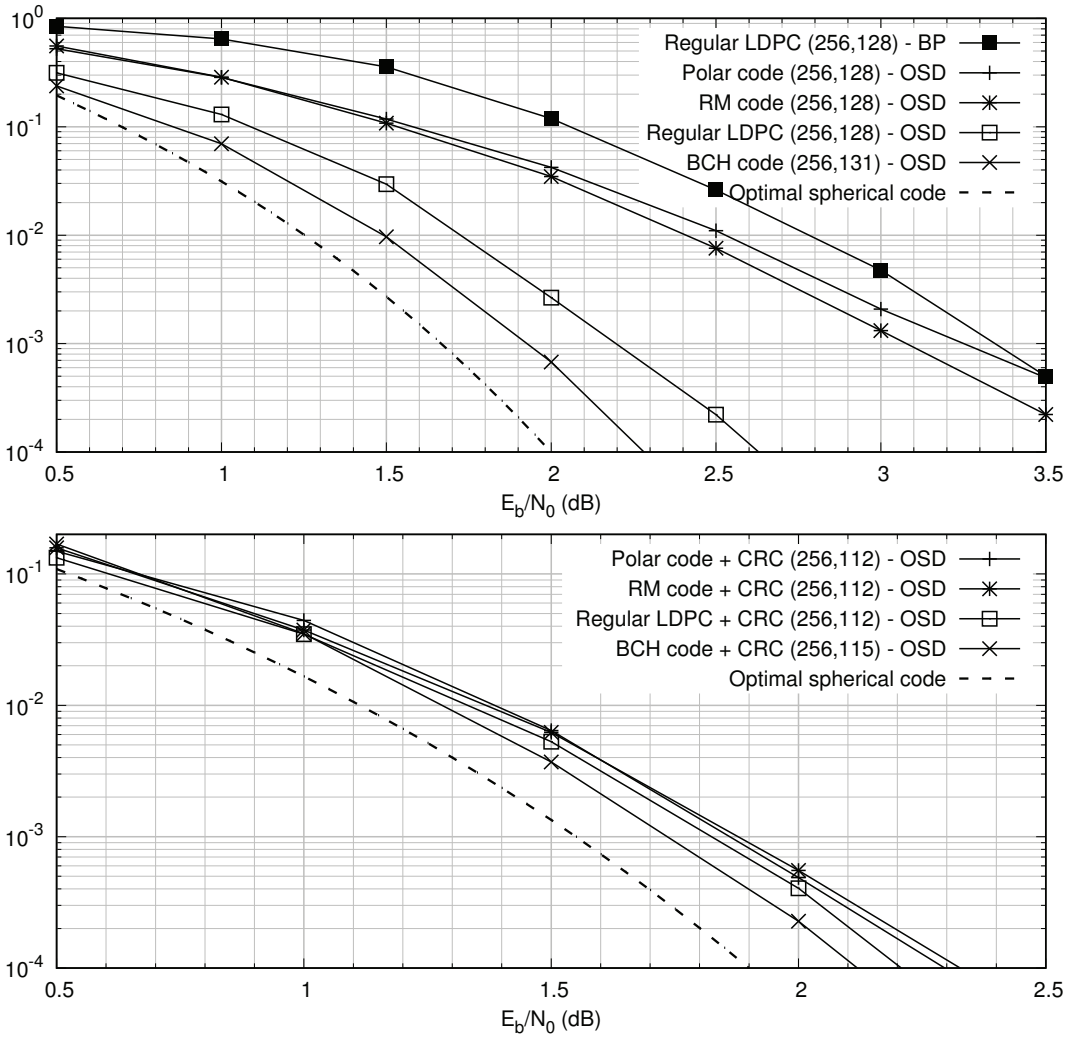


Figure 3.1: BERs on the BI-AWGN channel without (top) and with (bottom) CRC, taken from Wonterghem *et al.* [107].

Furthermore, in 2019, Korenda *et al.* [109] presented a way to extract private keys from Static Random Access Memory (SRAM) PUFs in a lightweight manner. To accomplish this, a fuzzy extractor is utilised in order to remove potential noise from the PUF response. The authors mention two very important factors when choosing the underlying ECC: The false authentication rate and the false rejection rate. Neither should be too high, and a trade-off is needed to achieve the best performance. The authors also only analysed the performance of their examined codes, not their complexity. Furthermore, their fuzzy extractors were implemented with MATLAB which already provides working implementations for some codes. They examined a handful of widely used codes with different key lengths and helper data bits and about the same bit-flip probability. In particular, they used a generalised multiple-concatenated RM code, a BCH repetition code, a GC RM code, a GC RS code, Polar codes using Successive Cancellation (SC) decoding and Hash-Aided Successive Cancellation List (HA-SCL) decoding, and finally, serially concatenated BCH and Polar codes using either SC or BP decoding. Although its generalised multiple-concatenated RM code uses by far the most auxiliary data bits, its error probability, in the range of 10^{-6}, is among the worst. Both GC codes exhibit the best, *i.e.* the lowest failure probability, closely followed by the serially concatenated BCH and Polar codes using BP decoding. Detailed results can be found in their paper accordingly. However, in addition, the authors used many PUF responses orig-

inating from the same PUF that was used to initialise the fuzzy extractor, and also many PUF responses originating from other PUFs of the same type. In this way, they were able to examine the security aspects of the codes and the relevant fuzzy extractors more closely. The results can be seen in Table 3.2. In this case, they only investigated the standard fuzzy extractor as proposed by Dodis *et al.* [95] using a BCH code, as well as the efficient fuzzy extractor by Kang *et al.* [110] using a BCH code, Polar codes using HA-SCL decoding, and serially concatenated BCH and Polar codes. It is evident that Polar codes using the HA-SCL decoding exhibit the best properties compared to those of the other schemes, closely followed by serially concatenated BCH and Polar codes. The other two codes suffer from a very high failure rate when a different PUF instance is being used, that is, unauthenticated PUF responses are being recognised as authenticated ones. This would, of course, open the door to possible attackers who could use random PUF responses to get to the secret protected by the fuzzy extractor.

Table 3.2: Failure probabilities through Intra and Inter PUF distance effects, taken from Korenda *et al.* [109].

Code	% Error when the same PUF is used	% Error when a different PUF is used
BCH	0.2%	47.62%
Efficient BCH	12.5%	48.98%
Polar HA-SCL	0%	19.23%
SC BCH Polar	0%	23.68%

Moreover, a 2019 paper by Gao *et al.* [111] focuses on fuzzy extractors using BCH codes. To compare the BCH codes in practice, a SRAM PUF data set containing measurements at different temperatures was used [112].

The authors refer to a BCH code not by the usual distance parameter d or δ, but rather by t, which is the number of errors which the BCH code can correct, that is, $t = \lfloor (d-1)/2 \rfloor$, according to Lemma 1. The authors examine different BCH codes regarding their Key Failure Rate (KFR) and overhead. The KFR assesses the probability that the resulting extracted key from "correct" (but noisy) PUF responses is correct, while the overhead is just a measure of the computational complexity of the corresponding procedure, *i.e.* the number of clock cycles and memory required.

The KFR is examined for three different types of preprocessors and fuzzy extractors used in conjunction with each other. Table 3.3 shows clearly that their proposed fuzzy extractor Multiple Reference Response (MRR) enrollment procedure is superior to the standard Single Reference Response (SRR) enrollment procedure. The typical SRR method extracts a key from the information provided by a single PUF response, whereas MRR uses the information provided by many different PUF responses from the same PUF instance, perhaps even under different environmental conditions. This makes MRR fuzzy extractors more resistant to environmental influences such as temperature.

However, in addition to the fuzzy extractors, the preprocessors also have a significant impact on the KFR, with the typical single readout method being the worst. Using the result of the majority vote of many PUF responses as input to the fuzzy extractor can give better results. Better still, preselecting stable bits within the PUF response can further reduce the KFR.

As seen in Table 3.4, the number of clock cycles needed to encode a given PUF response increases significantly as the block length n of the underlying BCH code increases, and so does the memory usage for both Ferroelectric Random Access Memory (FRAM) and SRAM. Their reported FRAM usage corresponds to the size of the `.text` segment of the

Table 3.3: KFRs for various BCH code-based fuzzy extractors, taken from Gao *et al.* [111].

		Single readout			Majority voting			Preselection		
(n_1,k_1,t_1)	block num.	SRR	2MRR	3MRR	SRR	2MRR	3MRR	SRR	2MRR	3MRR
(63,18,10)	8	0.6074	0.2821	2.67×10^{-2}	0.4355	0.1446	2.7×10^{-3}	2.26×10^{-2}	1.10×10^{-2}	8.21×10^{-6}
(63,16,11)	8	0.3789	0.1342	8.2×10^{-3}	0.2366	5.9×10^{-2}	6.22×10^{-4}	6.8×10^{-3}	3.0×10^{-3}	9.85×10^{-7}
(127,29,21)	5	0.1712	2.86×10^{-2}	2.49×10^{-4}	7.55×10^{-2}	7.1×10^{-3}	2.82×10^{-6}	1.79×10^{-4}	4.36×10^{-5}	2.97×10^{-11}
(127,22,23)	6	6.62×10^{-2}	7.4×10^{-3}	3.04×10^{-5}	2.39×10^{-2}	1.4×10^{-3}	1.91×10^{-7}	2.08×10^{-5}	4.19×10^{-6}	5.47×10^{-13}
(127,15,27)	9	5.7×10^{-3}	2.95×10^{-4}	2.66×10^{-7}	1.4×10^{-3}	3.51×10^{-5}	5.09×10^{-10}	1.66×10^{-7}	2.67×10^{-8}	$< 10^{-21}$
(255,47,42)	3	2.48×10^{-2}	9.0×10^{-4}	1.25×10^{-7}	5.4×10^{-3}	6.8×10^{-5}	2.47×10^{-11}	6.69×10^{-8}	4.58×10^{-9}	$< 10^{-21}$
(255,29,47)	5	2.8×10^{-3}	3.96×10^{-5}	8.27×10^{-10}	3.83×10^{-4}	1.62×10^{-6}	3.66×10^{-14}	3.92×10^{-10}	1.65×10^{-11}	$< 10^{-21}$
(255,21,55)	7	1.52×10^{-5}	4.72×10^{-8}	4.59×10^{-14}	9.90×10^{-7}	7.17×10^{-10}	$< 10^{-21}$	1.79×10^{-14}	$< 10^{-21}$	$< 10^{-21}$
(255,13,59)	10	7.97×10^{-7}	1.45×10^{-9}	$< 10^{-21}$	3.56×10^{-8}	1.59×10^{-11}	$< 10^{-21}$	$< 10^{-21}$	$< 10^{-21}$	$< 10^{-21}$

corresponding implementation for the BCH code. The SRAM usage is the amount of memory allocated for the internal state of the program, that is, the amount of memory usually referred to as "Random Access Memory (RAM)." It should also be noted that either reducing the dimension k of the BCH code or increasing its error-correcting capability t will result in fewer clock cycles but increased memory usage. The reason why k must get smaller as d increases is most likely the so-called *Singleton bound* $d + k \leq n + 1$ which enforces an upper bound on $d + k$ when n is fixed which is exactly the case in each of the individual blocks of rows in Table 3.4.

Table 3.4: Encoding and decoding overhead, respectively, for various BCH codes, taken from Gao *et al.* [111].

(n_1,k_1,t_1)	clock cycles	FRAM usage	SRAM usage
(63,18,10)	53,944	858 bytes	114 bytes
(63,16,11)	51,003	738 bytes	117 bytes
(63,7,15)	31,577	842 bytes	126 bytes
(127,64,10)	238,671	858 bytes	197 bytes
(127,57,11)	248,433	1,002 bytes	204 bytes
(127,50,13)	235,005	1,034 bytes	211 bytes
(127,43,14)	219,095	1,050 bytes	218 bytes
(127,36,15)	198,758	1,044 bytes	225 bytes
(127,29,21)	181,438	1,058 bytes	232 bytes
(127,22,23)	167,509	1,072 bytes	239 bytes
(127,15,27)	111,335	1,054 bytes	246 bytes
(255,123,19)	930,093	1,370 bytes	394 bytes
(255,63,30)	680,087	1,418 bytes	454 bytes
(255,47,42)	583,024	1,446 bytes	470 bytes
(255,37,45)	476,744	1,492 bytes	480 bytes
(255,29,47)	377,220	1,506 bytes	488 bytes
(255,21,55)	294,783	1,520 bytes	496 bytes
(255,13,59)	201,535	1,418 bytes	504 bytes

(n_1,k_1,t_1)	clock cycles	FRAM usage	SRAM usage
(63,18,10)	393,836	1,882 bytes	1,226 bytes
(63,16,11)	435,027	2,022 bytes	1,168 bytes
(63,7,15)	626,881	2,572 bytes	1,154 bytes
(127,64,10)	670,622	3,676 bytes	1,226 bytes
(127,57,11)	748,933	3,942 bytes	1,228 bytes
(127,50,13)	1,030,444	4,466 bytes	1,228 bytes
(127,43,14)	978,775	4,728 bytes	1,228 bytes
(127,36,15)	1,057,415	5,006 bytes	1,218 bytes
(127,29,21)	1,574,426	6,602 bytes	1,228 bytes
(127,22,23)	1,742,276	7,134 bytes	1,226 bytes
(127,15,27)	2,102,222	8,198 bytes	1,228 bytes
(255,123,19)	2,515,163	11,958 bytes	1,228 bytes
(255,63,30)	4,116,796	17,700 bytes	1,228 bytes
(255,47,42)	6,102,010	23,964 bytes	1,228 bytes
(255,37,45)	6,582,507	25,530 bytes	1,226 bytes
(255,29,47)	6,976,341	26,574 bytes	1,228 bytes
(255,21,55)	8,345,992	30,750 bytes	1,226 bytes
(255,13,59)	8,528,363	34,566 bytes	1,298 bytes

Furthermore, both the decoding complexity and the FRAM usage increase significantly as n increases. This is because larger blocks require more computing power to decode than smaller blocks. However, SRAM usage is almost completely stable across all measurements since the authors use syndrome-based decoding. Moreover, both clock cycles and FRAM usage increase additionally when decreasing k or increasing t. Again, the Singleton bound enforces the upper bound on the sum $d+k$, so increasing t forces k to decrease. Additionally, when increasing t, more errors need to be taken care of.

Finally, the authors discuss a Helper Data Manipulation (HDM) attack [113] which applies to soft-decision decoding algorithms. However, since syndrome decoding is a hard-decision scheme, it is not applicable in this case.

Finally, in 2020, Hiller *et al.* [10] published another in-depth comparison of ECCs in the context of fuzzy extractors in FPGAs. The authors used BCH codes, RM codes, Polar codes, extended Golay codes, repetition codes, and some other procedures that do not depend on ECCs. For this reason, these approaches are beyond the scope of this paper.

However, the authors additionally compared different extractor algorithms after explaining their inner workings. The authors mention three important criteria for fuzzy extractors:

1. *Chip area:* The implementation cost of the PUF (intrinsic versus extrinsic), a trade-off between the number of PUF bits and the algorithmic complexity of post-processing.
2. *Helper data:* The storage cost (depending on their length, the speed of the memory), as well as their integrity.
3. *Run time:* The real-time constraints, potential usage of pipelining.

Furthermore, additional data about the PUF bits could be taken into account, such as their spatial correlation or bias, to select the best-suited ECC for the given use case. It is important to note that the implementation of the BCH code from [10] uses in part a custom decoding algorithm, which is explained in detail in their work. The authors additionally list four fuzzy extractor approaches for use with linear ECCs:

1. *CO fuzzy extractor:* The typical fuzzy extractor as proposed by Dodis *et al.* [95].
2. *Systematic Low-Leakage Coding (SLLC):* Provides additional masking of helper data bits using PUF response bits; see Kang *et al.* [110].
3. *Complementary Index-Based Syndrome coding (C-IBS):* Using a repetition-like code, pointers to specific bits in the PUF response are the helper data; see Yu and Devadas [114].
4. *Maximum-Likelihood Symbol Recovery (MLSR):* Instead of computing pointers to specific bits, they point to different blocks in the PUF response; see Yu *et al.* [115].

Various combinations of these fuzzy extractor approaches and ECCs have been examined. Regarding helper data bits, the BCH code with the CO fuzzy extractor has performed the best, requiring the least amount of bits, closely followed by the extended Golay code using the same fuzzy extractor. The RM code they used needed almost three times the amount of helper data bits when using the C-IBS approach and five times more when using the CO fuzzy extractor. It is also obvious that the amount of helper data bits only increases linearly in the case of BCH codes using the CO construction. However, when the allowed error probability of the fuzzy extractor is reduced, the number of helper data bits of the CO-BCH approach grows quadratically. In this case, Polar codes seem to exhibit the best performance, closely followed by a hybrid approach using RM and RS codes. In terms of the clock cycles required for the decoding algorithm, *i.e.* key reproduction, RM codes are clearly superior to BCH codes. This behaviour also changes when the allowed error probability is decreased. In that case, the BCH codes perform very similar to the RM codes. Detailed results can be found in [10] and Figure 3.2. Finally, the authors give several recommendations for choosing the right ECC, some of which are listed below:

- *Use all available information:* If the PUF is known to have a certain error pattern, the code should be able to error correct it. In addition, if the error rate is known or estimated, BCH codes could be constructed with this information in mind.
- *Decrease error rate beforehand:* Ignoring unreliable bits is crucial in reducing the number of clock cycles needed for the decoding algorithm.
- *Rather more than too little error correction:* To improve reliability, the PUF size may also be increased instead of choosing a "better" ECC.
- *Avoid side-channels:* Iterative approaches such as BP are known to exhibit data-dependent timing behaviour. This is particularly evident in the work of Mexis *et al.* [5, 108], who used BP decoding in the context of LDPC codes.
- *Large blocks improve the error correction performance*: Depending on the processing power of the verifier, the block size can be increased to achieve better code rates.

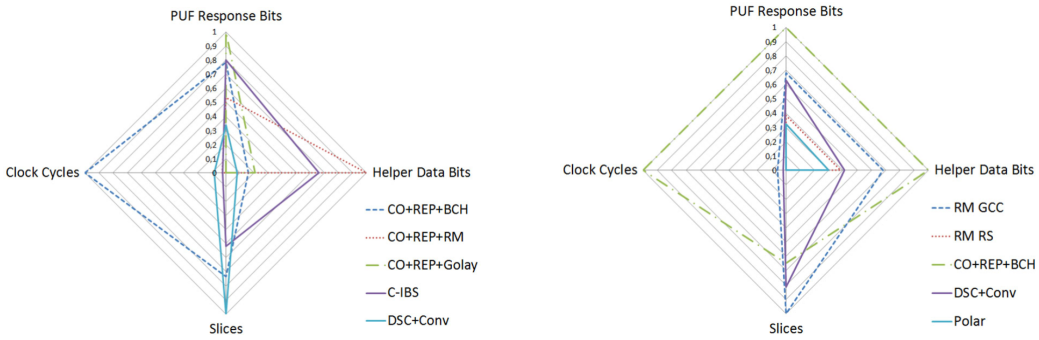


Figure 3.2: Performance of different ECC approaches designed for a key error probability of 10^{-6} and 10^{-9}, respectively, taken from Hiller *et al.* [10].

3.2 CoCoA 5

The accompanying code for this master's thesis was written in C++. However, since many algebraic features such as finite fields, polynomials, matrices, and basic operations on them are required, CoCoALib [116] has been used to ensure maximum efficiency for all procedures. The name CoCoA stands for "Computations in Commutative Algebra," and the CoCoALib library comes with an executable binary called CoCoA-5 [117], which provides a Command-Line Interface (CLI) that allows many of its functions to be used in a script-like manner. Overall, CoCoA is a CAS and was developed by the University of Genoa and is based on two smaller projects by Giovini and Niesi [118], which were merged in 1988 to form the software CoCoA under the coordination of Robbiano [119]. In 1990, the authors released the first version, CoCoA 0.99, which was able to handle multivariate polynomial rings, ideals, and many algebraic algorithms such as the Buchberger Algorithm to calculate Gröbner bases [55, 56]. Since then, many people such as Robbiano, Kreuzer, Abbott, and Bigatti have joined the team and continue to develop the software, releasing CoCoA versions 3, 4 and 5.

For this thesis, all algorithms were first written in the high-level programming language CoCoALanguage, extensively tested, and then ported to C++ to be tested, improved, and optimised again. An example written in this programming language is shown in Listing 3.1.

Listing 3.1: The Buchberger algorithm [55, 56] from Algorithm 9 for finding a Gröbner basis of a set of polynomials in CoCoALanguage.

```
define Buchberger(G)
  // Make all monic for a (sometimes) nicer GB
  G := [monic(g) | g in G];
  r := len(G);
  B := [x in tuples(1..r, 2) | 1 <= x[1] and x[1] < x[2] and x[2] <= r];
  while B <> [] do
    p := B[1];
    remove(ref B, 1);
    gj := G[p[1]];
    gk := G[p[2]];

    // Optimization: Skip if LTs are coprime
    if IsCoprime(LT(gj), LT(gk)) then
      continue;
    endif;
```

```
    t := lcm(LT(gj), LT(gk));
    Sjk := (t / LM(gj)) * gj - (t / LM(gk)) * gk;
    R := NR(Sjk, G);
    if R <> 0 then
      // Make monic for a (sometimes) nicer GB
      append(ref G, monic(R));
      r := len(G);
      B := concat(B, [[i, r] | i in 1..(r-1)]);
    endif;
  endwhile;
  return G;
enddefine; -- Buchberger
```

4 Implementation

As explained in Section 3.2, the accompanying code for this master's thesis was written in C++ using the CoCoALib [116] library. However, to make the compilation process as easy as possible, installation instructions tested on Windows 10 and 11 using Cygwin [120] and Ubuntu (each 64-bit) are included in a markdown formatted `README.md` file. 32-bit Operating Systems (OSs) have not been explicitly tested, but should be compatible. The first thing to do is compile CoCoALib, which is done by installing some packages as described in the `README.md` file, downloading the source to a specific folder, and then running a `make_cocoalib.sh` shell script file. Then CMake [121] is used to first generate `Makefile`s, which are then executed using Make [122] to build the project. A `.editorconfig` file is also included to define general guidelines for all files in the project using EditorConfig [123].

The whole project is available as open-source on GitHub[5] under the Massachusetts Institute of Technology (MIT) licence, which is a permissive licence. This allows the project to be used commercially, modified and redistributed in any way, as long as the copyright notice is still included. In addition, each entity (function, class, its members, etc.) has been documented in code in a format that can be converted to a manual using Doxygen [124]. If Graphviz [125] is installed and its `dot` tool is detected, Unified Modeling Language (UML) diagrams are also generated.

Finally, the architecture of the project is outlined. Being a C++ project, it is standard procedure to split the code into header files and source files. Therefore, the header files containing the function declarations are in the `include` directory, while the `src` directory contains their definitions. Moreover, the root CMake module, called `ECC`, is divided further into the following submodules:

- `util`: General utility functions that are not specific to any use case
- `types`: General linear and cyclic code specific entities (Sections 2.1.1 and 2.1.2)
- `BCH`: BCH and RS code specific entities (Sections 2.1.2.1 and 2.1.2.2)
- `Golay`: Golay code specific entities (Section 2.1.1.2)
- `Ham`: Hamming code specific entities (Section 2.1.1.1)
- `RM`: RM code specific entities (Section 2.1.1.3)
- `Fuzzy`: Fuzzy extractor specific entities (Section 2.2)
- `test`: Executable test cases for all of the above codes and fuzzy extractors

[5] `https://github.com/ThexXTURBOXx/ECC`

4.1 CoCoALib's Built-Ins

As already explained in Section 3.2, CoCoALib offers many built-in functions and data types. These entities are mostly available in the global `namespace CoCoA`. For this reason, the code for this master's thesis is available in the sub-`namespace CoCoA::ECC`, to make everything as accessible as possible without using the `using namespace` directive. In terms of throwing errors, the `CoCoA_THROW_ERROR` macro is used to be consistent with CoCoALib's standard practices. As a result, an `ErrorInfo` object is thrown as an exception, which contains additional information about the crash, such as the line number of the exception and even a detailed error message, which can be either custom or one of the predefined ones from the `CoCoA::ERR` sub-namespace.

The most important built-in data types are listed below:

- `BigInt`: A "normal" integer that can be larger (in theory infinitely long; RAM is one of the limits) than the usual C++ data types
- `ring`: An algebraic ring which can be a `RingZZ` ($\mathbb{Z}$), `RingQQ` ($\mathbb{Q}$), `RingFp` ($\mathbb{F}_p$), `SparsePolyRing` (polynomial ring) *etc.*
- `RingElem`: An element of an algebraic `ring`
- `ConstRefRingElem`: Type alias for `const RingElem &`
- `(Const)MatrixView`: Conversion object that provides access functions to `vectors` and similar data structures and even concatenated or modified `(Const)MatrixViews` as matrices
- `matrix`: A fully converted matrix; its entries are no longer pointers or references as in `(Const)MatrixView`, but are now in their own `matrix` object

Detailed information and examples about these data types and many more can be found in the CoCoALib manual. In addition to these data types, functions are provided in order to aid constructing or manipulating them. For instance, a `ring` can be constructed using either `RingZZ()`, `RingQQ()`, `NewPolyRing(ring)` or `NewZZmod(BigInt)` *etc.* There are also functions such as `power(...)` which allow one to take the power of a *e.g.*, `BigInt` or `RingElem`. Furthermore, operator overloading allows most algebraic data types to be added, multiplied, *etc.* using their respective standard C++ syntax. However, it is important to note that the `operator^` is not used to take the power of any object. This is because it references the bitwise eXclusive OR (XOR) operator by default in C++, and therefore its operator precedence is lower than that of `operator*`. As a result, `3*x^2` would evaluate to the same value as `(3^2)*(x^2)`, which is not expected.

Finally, the accompanying code for this master's thesis contained two functions, `IsPrimitivePoly` and `IsPrimitivePoly_NoArgChecks`, which allowed one to check whether a given polynomial was primitive, using an approach very similar to that of Alanen and Knuth [126]. After careful consideration, it seemed beneficial to include this procedure in the official release of CoCoALib itself. For this reason, Abbott included the functions in CoCoALib 0.99815, but for legacy reasons they still also reside in a multi-line comment in the `util` submodule.

4.2 Error Correction Codes

In this and the following subsections, the implementation of the various ECCs is explained in detail. However, before discussing specific implementations for the ECCs, the common procedures defined in the `types` submodule are explained. It contains two files: `linear.H` and `cyclic.H` which define procedures for linear and cyclic codes, respectively.

Since every linear code is uniquely defined (up to equivalence/isomorphism) through its corresponding generator matrix or parity-check matrix, the two functions `matrix checkMat(const matrix &G)` and `matrix genMat(const matrix &H)` can be used to compute a corresponding generator matrix out of the parity-check matrix of a linear code and vice versa, according to Lemma 6. It should be noted that both functions expect the input matrix to be systematic. Additionally, `linear.H` declares a function `matrix linEncode(const matrix &G, const matrix &w)` which is able to encode a word `w` to its corresponding codeword using the generator matrix `G`. For this, according to Theorem 5, it is sufficient to just `return w * G`.

On the other hand, `cyclic.H` declares common procedures for cyclic codes. This file again contains a procedure to calculate the check polynomial for a given generator polynomial. However, this time, not the check polynomial itself is calculated, but rather the so-called *dual polynomial*, that is, `RingElem dualPoly(ConstRefRingElem g, const long n)` where the additional parameter `n` is the block length of the code. The dual polynomial $\bar{h}$ is defined as $\bar{h} := x^k h(x^{-1})$ where h is the check polynomial of the code. Hence, $\bar{h}$ can be effectively obtained through `reverse(h)` and vice versa. Moreover, the file contains the function `RingElem sysEncodeCyclic(ConstRefRingElem g, ConstRefRingElem p, ConstRefRingElem x, const long n, const long k)` which is able to systematically encode a given word `p` using the generator polynomial `g`, both defined in the indeterminate `x`, for a block code of block length `n` and dimension `k`. Hence, this function is effectively an implementation of Theorem 48. Finally, the function `RingElem decodeCyclicGroebner(ConstRefRingElem g, ConstRefRingElem p, ConstRefRingElem x, ConstRefRingElem a, const long q, const long n, const long qn)` is an implementation of Algorithm 10 where `g`, `p`, `x` and `n` are defined as in `sysEncodeCyclic`, `a` is a primitive element of the underlying field extension, `q` its characteristic and `qn` the amount of elements in it.

4.2.1 Hamming Codes

As explained earlier, Table 2.1 allows us to devise an algorithm to efficiently decode Hamming codes. In order to construct the parity-check matrix for $\mathrm{Ham}(r, 2)$ (again, $q = 2$ is fixed here since we only care about the binary Hamming codes), all the possible $n = 2^r - 1$ non-zero binary vectors need to be enumerated [11]. They form the columns of the parity-check matrix. To create an equivalent systematic Hamming code, the vectors are first sorted in descending order by their Hamming weight and then lexicographically if some vectors have the same weight. In this way, an identity matrix is formed from the last few columns of the resulting matrix. This sorting is done using the standard `std::sort` function and a comparator function `bool sortHam(const vector<long> &a, const vector<long> &b)`. The parity-check matrix itself is generated in the function `matrix hamH(const ring &R, const long r, const long q)`.

A `struct Ham` is also defined as shown in Listing 4.1, which acts as a Data Transfer Object (DTO) holding the parameters of the Hamming code. The variables `r` and `q` are the parameters r and q from the definition of the Hamming code $\mathrm{Ham}(r, q)$. `R` is the underlying ring and `n`, `k` and `d` are the block length, dimension and minimum Hamming distance of the code which are determined during object construction according to Lemma 18. `H` is the parity-check matrix computed by the above function `hamH` and the generator matrix `G` is determined by `genMat` from `types/linear.H`.

Listing 4.1: Declaration of the `struct Ham`.

```
struct Ham {
  const long r;
  const long q;
  const ring R;
  const long n;
  const long k;
  const long d = 3;
  const matrix H;
  const matrix G;

  Ham(const long r, const long q)
      : r(r), q(q), R(NewZZmod(q)), n((SmallPower(q, r) - 1) / (q - 1)),
        k(n - r), H(hamH(R, r, q)), G(genMat(H)) {
  }
};
```

Finally, the functions `matrix encodeHam(const Ham &ham, const matrix &w)` and `matrix decodeHam(const Ham &ham, const matrix &w)` are used to encode a word `w` using the given Hamming code `ham`. It is worth noting that the implementation can also handle non-binary Hamming codes, although they are not used in the fuzzy extractors. The encoding function simply delegates to `linEncode` from `types/linear.H`, while the decoding function is a brute-force algorithm that determines the unique error vector of Hamming weight 1.

4.2.2 Golay Codes

As with Hamming codes, a `struct Golay` is defined, as shown in Listing 4.2, which acts as a DTO for the parameters of the underlying Golay code. Again, `G` is the generator matrix, `R` the underlying ring/field, `q` its characteristic, `n` the block length of the code, `k` its dimension, and `d` its minimum Hamming distance. However, this time, there are additional variables, called `A`, `AExt`, and `GExt`. First of all, `A` corresponds to the part of the generator matrix `G` which is not the identity matrix. Furthermore, `GExt` corresponds to the matrix of the extended Golay code which can be the same as `G` if it is already extended. The definition of `AExt` is then analogous to `A` for `GExt`.

The class has two constructors: One takes just a number `n`, *i.e.*, the block length of the Golay code which can be one of those n listed in Table 2.2, and constructs its own underlying ring. The other one constructs a `Golay` instance over the given ring. If an invalid argument (e.g., 7) is provided, a `CoCoA::ERR:BadArg` error is thrown.

The values for q, n, k and d are determined through formulas and ternary conditional statements which are equivalent to simple `if` or `switch-case` instructions. The other values are generated through the functions `matrix golayMatPart(const ring &R, long n)` or `matrix golayMatPart(long n)` and `matrix golayMat(const ring &R, long n)` or `matrix golayMat(long n)`.

Listing 4.2: Declaration of the `struct Golay`.

```
struct Golay {
  const matrix A;
  const ring R;
  const matrix G;
  const matrix AExt;
  const matrix GExt;
  const long q;
  const long n;
  const long k;
  const long d;

  explicit Golay(const long n)
      : A(golayMatPart(n)), R(RingOf(A)),
        G(NewDenseMat(ConcatHor(IdentityMat(R, NumRows(A)), A))),
        AExt(IsEven(n) ? A : golayMatPart(R, n + 1)),
        GExt(IsEven(n) ? G
          : NewDenseMat(ConcatHor(IdentityMat(R, NumRows(AExt)), AExt))),
        q(n > 20 ? 2:3), n(n), k((n + 1) / 2),
        d(n / (5 - q)) /* magic formula */
  {
  }

  explicit Golay(const ring &R, const long n)
      : A(golayMatPart(R, n)), R(R),
        G(NewDenseMat(ConcatHor(IdentityMat(R, NumRows(A)), A))),
        AExt(IsEven(n) ? A : golayMatPart(R, n + 1)),
        GExt(IsEven(n) ? G
          : NewDenseMat(ConcatHor(IdentityMat(R, NumRows(AExt)), AExt))),
        q(n > 20 ? 2:3), n(n), k((n + 1) / 2),
        d(n / (5 - q)) /* magic formula */
  {
  }
};
```

Again, analogous to Hamming codes, there are two functions `matrix encodeGolay(const Golay &golay, const matrix &w)` and `matrix decodeGolay(const Golay &golay, const matrix &w)` which are able to encode and decode words using the given Golay code instance, respectively. The encoder simply delegates to `linEncode` as well.

For decoding using the binary Golay codes, the algorithm by Hankerson *et al.* [19] is used. It is a very simple syndrome-based decoding algorithm which exploits the fact that any correctable error vector can only have a weight of ≤ 3 (since the minimum Hamming distance of the Golay code can only be 6 or 7). Because of this, any word which had been encoded using the extended binary Golay code, must have at most 1 error in either the first or second half. The algorithm is described in Algorithm 16.

Algorithm 16 Decoding of the extended binary Golay code, adapted from Hankerson *et al.* [19].

Require: Message to decode w, generator matrix G of G_{24}

$S \leftarrow w \cdot G^T$
if $\mathrm{wt}(S) \leq 3$ **then return** $w + [S, 0]$
end if
if $\mathrm{wt}(S + a_i) \leq 2$ for some row a_i of A **then return** $w + [S + a_i, e_i]$
end if
$S_A \leftarrow S \cdot A^T$
if $\mathrm{wt}(S_A) \leq 3$ **then return** $w + [0, S_A]$
end if
if $\mathrm{wt}(S_A + a_i) \leq 2$ for some row a_i of A **then return** $w + [e_i, S_A + a_i]$
end if
return *Cannot decode!*

When decoding words using the non-extended binary Golay code, the last bit is simply calculated and added to the message. Then the above algorithm can also be applied to the word. It should also be noted that the ternary Golay codes have not yet been implemented which is why an `CoCoA::ERR:NYI` error will be thrown when trying to decode any word using a ternary code instance.

4.2.3 Reed-Muller Codes

Again, a `struct RM` acts as a DTO for the parameters of the underlying RM code, as shown in Listing 4.3. Obviously, `R` is the underlying ring, `n`, `k` and `d` are the block length, dimension and minimum Hamming distance of the code, respectively, and `G` is its generator matrix. If the underlying code is $\mathrm{RM}(r, m)$, then these two parameters r and m correspond to `r` and `m`, respectively.

In order to understand the other variables, it is important to mention that the implementation of the RM codes is based on the work by Raaphorst [127]. There, the RM codes are defined similarly to here in Section 2.1.1.3, with the only difference being that Raaphorst also introduced affine geometries. Also, the decoding algorithm given in his work is only a bit different than Algorithm 7, since both procedures utilise simple majority logic decoding.

The definition of the last few variables is as follows. Since the monomials x_i of the code as defined in Section 2.1.1.3 can be translated to vectors in the ring `R`, the function `genXrows` does exactly this translation and the result is held in the variable `xrows`. Moreover, `votingRows` holds exactly all the translated monomial vectors which are not in each row, respectively. Finally, `ribd` (short for "row indices by degree") holds the indices of the rows which correspond to monomials of given degree i, each.

The functions `genK`, `genG`, `genVotingRows` and `genRibd` are the functions which calculate `k`, `G`, `votingRows` and `ribd`, respectively. These "generators" are only needed during the construction of an `RM` object which is why the functions are marked both `private` and `static`. Due to the recursive nature of RM codes, their construction using this procedure involves some recursive computations. There are again two ways to construct an `RM` object. Either by the specification of only `r` and `m` or using an additional argument `R` which is expected to be the ring $\mathbb{F}_2$. In the former case, such a ring is constructed instead.

Listing 4.3: Declaration of the `struct RM`.

```
struct RM {
  const ring R;
  const long r;
  const long m;
  const long n;
  const long k;
  const long d;
  const vector<vector<RingElem>> xrows;
  const matrix G;
  const vector<vector<vector<RingElem>>> votingRows;
  const vector<long> ribd;

  RM(const long r, const long m) : RM(NewZZmod(2), r, m) {
  }

  RM(const ring &R, const long r, const long m)
      : R(R), r(r), m(m), n(SmallPower(2, m)), k(genK(r, m)),
        d(SmallPower(2, m - r)), xrows(genXrows(R, m)),
        G(genG(R, r, m, xrows)), votingRows(genVotingRows(R, r, m)),
        ribd(genRibd(r, m)) {
  }

private:
  static long genK(long r, long m);
  static vector<vector<RingElem>> genXrows(const ring &R, long m);
  static matrix genG(const ring &R, long r, long m,
                     const vector<vector<RingElem>> &xrows);
  static vector<vector<vector<RingElem>>> genVotingRows(const ring &R,
                                                        long r, long m);
  static vector<long> genRibd(long r, long m);
};
```

Since the RM codes are linear, `matrix encodeRM(const RM &rm, const matrix &w)` again only delegates to `linEncode`. The function `matrix decodeRM(const RM &rm, matrix w)` uses the list `ribd` in order to perform majority voting on the relevant rows of the generator matrix `G`.

4.2.4 BCH Codes

The `struct BCH` holds the parameters of the underlying BCH code. The variables `n`, `k` and `d` again stand for the block length, dimension and minimum Hamming distance of the BCH code, respectively. Also, `q` is again the characteristic of the underlying field and `qn` the number of its elements. The underlying polynomial ring is stored in `R`. Alas, the implementation can only handle primitive BCH codes. Since the underlying field is a splitting field extension, there is a primitive polynomial $f(\alpha)$, which is dependent on the indeterminate α and defines the field $\mathbb{F}_p[\alpha]/\langle f(\alpha)\rangle$ which is isomorphic to the underlying field. In the accompanying code this is exactly how the field is expected to be defined. Hence, `qn` is the same value as $\mathtt{q}^{\mathtt{m}}$ where $\mathtt{m} = \deg(f)$. It should also be mentioned that popular choices for this $f(\alpha)$ are Conway polynomials as explained in Section 1.1. The variables `a`, `g` and `x` are then the primitive element of the underlying field and generator polynomial of the BCH code, dependent on the indeterminate `x`.

A big difference between the `struct BCH` and other aforementioned `struct`s is that this time the constructor is "dumb" in a way. It only initialises the internal variables either normally or through `std::move`s. Because of this, `BCH` instances should be constructed using the function `BCH constructBCH(const long q, const long d, const long c, const string &prim, const string &alpha, const string &x)`. It takes as input the desired characteristic of the underlying field `q`, designed distance `d`, sense `c`, primitive polynomial `prim`, dependent on the indeterminate with name `alpha`, and the desired generator polynomial indeterminate name `x`. The function first initialises a polynomial ring $\mathbb{F}_{\texttt{q}}[\texttt{alpha}]/\langle \texttt{f} \rangle[\texttt{x}] \cong \mathbb{F}_{\texttt{q}^{\texttt{m}}}[\texttt{x}]$ and then proceeds with the calculation of the generator polynomial `g`. This is done using Definition 33. Lastly, the parameters of the BCH code can be determined using Lemma 63 and Theorem 64.

Listing 4.4: Declaration of the `struct BCH`.

```
struct BCH {
  const long q;
  const long qn;
  const long n;
  const long k;
  const long d;
  const long c;
  const RingElem a;
  const RingElem g;
  const RingElem x;
  const ring R;

  BCH(const long q, const long qn, const long n, const long k,
      const long d, const long c, RingElem a, RingElem g, RingElem x)
      : q(q), qn(qn), n(n), k(k), d(d), c(c),
        a(std::move(a)), g(std::move(g)), x(std::move(x)), R(owner(x)) {
  }
};
```

Encoding using BCH codes is done using the function `RingElem encodeBCH(const BCH &bch, ConstRefRingElem p)` which delegates to `sysEncodeCyclic` from `types/cyclic.H`. Also, decoding of BCH codes using Gröbner bases is possible using the function `decodeBCHGroebner(const BCH &bch, ConstRefRingElem p)` which delegates to `decodeCyclicGroebner`, also from `types/cyclic.H`. The "normal" `RingElem decodeBCH(const BCH &bch, ConstRefRingElem p)` function uses Algorithms 12 to 15 to decode words using the Peterson-Gorenstein-Zierler and Chien Search algorithms. Additionally, the implementation features an implementation of Forney's algorithm [84] in order to be able to decode non-binary inputs.

Alas, CoCoALib's `LinSolve` function is not yet able to solve linear systems of equations in splitting fields. Because of this, `LinSolve(M, -V)` is left as a comment in the code in favour of `-inverse(M) * V`.

4.2.5 Reed-Solomon Codes

Since, according to Definition 36, RS codes can be simulated using BCH codes, no extra implementation is necessary. The functions of Section 4.2.4 are sufficient to handle RS codes properly.

4.3 Fuzzy Extractor

Since the fuzzy extractor depends on the underlying ECC, it is important to have an abstraction layer that can be used as an interface in the fuzzy extractor methods. The two obvious ways would be the following:

1. Define an interface/superclass from which all ECC classes inherit
2. Pass function references/pointers directly into the fuzzy extractor on creation

Both ways have different pros and cons, some of which are listed below:

Interface/superclass	Function references/pointers
+ Allows inheritance	+ Flexibility
− Code bloat	− Harder to read

Since only very few properties are shared between different ECCs, a superclass may be overkill in this case. Also, the readability of function references or pointers can be greatly improved by specifying a type alias for the signature of the function that is expected to be passed into the fuzzy extractor class. Listing 4.5 defines a type alias `ECCFn` for a function whose only input is a `const matrix &` and which returns a `matrix`. This type alias can be used as-is for any code whose code words are usually described in matrix or vector form or similar. However, since this is not the case for cyclic codes, whose codewords are usually polynomials in a finite ring, conversion functions are needed for `matrix` ↔ `RingElem`. Since these are also useful in other scenarios, they have been placed in the `util` submodule.

Listing 4.5: Type alias for the signature of an encoding or decoding function of an ECC.

```
using ECCFn = std::function<matrix(const matrix &)>;
```

In order to provide the best portability, a `struct FuzzyExtractor` has been declared as shown in Listing 4.6. The type alias from Listing 4.5 is declared inside the `struct` itself. It is worth mentioning that there are two different constructors. The former constructor generates random helper data, whilst the latter takes in an additional `HelperData` object that specifies the helper data that should be used. In order to avoid retrospective changes to the passed `HelperData` object, but simultaneously also avoid copying the object, it is moved instead. The function names `generateHelperData` and `extract` are taken from Mexis *et al.* [5, 108] where the former represents the generation phase `Gen` and the latter represents the reproduction phase `Rep` of the fuzzy extractor as defined in Equation (2.1). There is an additional function that returns a copy of the internal `HelperData` object to avoid modifications. The private function `strongExtract` defines a strong extractor, that is, `Ext` in the context of the fuzzy extractor. The underlying hash function is a MIT licensed SHA-256 implementation in C++ by Lambert [128] that has been forked and included as Git submodule.

Listing 4.6: Declaration of the `struct FuzzyExtractor`.

```
struct FuzzyExtractor {
  using ECCFn = std::function<matrix(const matrix &)>;

  FuzzyExtractor(ECCFn encode, ECCFn decode, long messageBits,
                 long parityBits)
      : encode(std::move(encode)), decode(std::move(decode)),
        messageBits(messageBits), parityBits(parityBits),
        helperDataSet(false), hd(HelperData()) {
  }

  FuzzyExtractor(ECCFn encode, ECCFn decode, long messageBits,
                 long parityBits, HelperData hd)
      : encode(std::move(encode)), decode(std::move(decode)),
```

```
        messageBits(messageBits), parityBits(parityBits),
        helperDataSet(true), hd(std::move(hd)) {
  }

  matrix generateHelperData(const matrix &w);

  matrix extract(const matrix &wd);

  HelperData getHelperData() {
    return {hd}; // Copy to avoid modifying
  }

private:
  ECCFn encode;
  ECCFn decode;
  long messageBits;
  long parityBits;
  bool helperDataSet;
  HelperData hd;

  static matrix strongExtract(const matrix &w);
};
```

The `struct HelperData` features a default constructor that initialises a `HelperData` object with two empty matrices. The default constructor is not really needed in this case, but it removes compiler warnings. In the `struct FuzzyExtractor` it is only used as a placeholder until the real helper data has been set, which is indicated through the `bool helperDataSet`. Apart from that, a `HelperData` object is an effectively immutable DTO that only holds the two helper data parts.

Listing 4.7: Declaration of the `struct HelperData`.

```
struct HelperData {
  matrix s;
  matrix x;

  HelperData() : HelperData(NewDenseMat(RingZZ(), 0, 0),
                            NewDenseMat(RingZZ(), 0, 0)) {
  }

  HelperData(const matrix &s, const matrix &x) : s(s), x(x) {
  }
};
```

From the above assertions, it can be deduced that the function `generateHelperData` should cause a short circuit by throwing an error if the object `HelperData` has already been set. Furthermore, since the used SHA-256 library only accepts `std::string`s and `uint8_t` arrays, and returns only `uint8_t` arrays, two additional conversion functions are needed. The rest can be easily accomplished using CoCoA-5's built-in operators and functions as described in Section 4.1.

5 Comparison

The main contribution of this Master's thesis is the comparison of the above-mentioned ECCs, which is done in this chapter. In order to compare ECCs specifically in the context of fuzzy extractors, different metrics need to be considered. An obvious choice is the block length n, the dimension k and the minimum Hamming distance d, which have been derived for all ECCs in Section 2.1. Building on this, their respective error correction capabilities should be studied in detail.

However, there are other metrics that should also be considered, as they have a direct impact on the overhead required by the respective decoding (or even encoding) algorithms. This includes, but is not limited to, whether the code is linear or even cyclic, its time complexity, and its memory usage. In addition to their theoretical complexity, their practical implementation should also be examined in detail.

5.1 Linearity and Cyclicality

The first and most important thing to determine is whether the ECCs being examined are linear or even cyclic. For this reason, Table 5.1 gives an overview of all the above binary codes and their characteristics in this respect.

Table 5.1: Comparison of the examined ECCs in terms of linearity and cyclicality. ✓ and ✗ mean yes and no, respectively. A (✓) means that the code does have this property, but it is usually not exploited, own creation.

Code	Linear	Cyclic
$\mathrm{Ham}(r, 2)$	✓	(✓)
G_{23}	✓	(✓)
G_{24}	✓	✗
$\mathrm{RM}(r, m)$	✓	✗
BCH	✓	✓
RS	✓	✓

Since each code has an associated generator matrix, they are all trivially linear. In addition, since BCH codes have been described in terms of a generator polynomial and RS codes have been described in terms of BCH codes, these two classes are also cyclic. On the other hand,

RM codes are not cyclic. However, through puncturing as in Definition 38, a cyclic code can be obtained from any RM code [129].

For binary Hamming codes, Remark 62 applies and hence, they are all cyclic. However, as they are rarely considered as such, the check mark in this case is in parentheses. Moreover, according to Remark 61, the binary Golay code G_{23} is also cyclic, but is again rarely considered as such. On the other hand, the extended binary Golay code G_{24} is not cyclic.

Linearity is a very important property that almost all popular codes satisfy. Linearity also allows the construction of a generator matrix, which makes the encoding process as hard as matrix multiplication which can be easily implemented both in software and hardware. Decoding algorithms can also take advantage of this property by relying on syndromes and the like.

Cyclicality is a more specialised property that only a handful of codes satisfy. Since cyclicality implies linearity, both generator matrices and generator polynomials can then be used to encode given messages. Additionally, syndromes can be defined both as matrices/vectors or polynomials when decoding a given word. Moreover, many decoders specific to cyclic codes can be applied to them.

5.2 Parameters

As mentioned earlier, the parameters, namely block length n, dimension k and minimum Hamming distance d of each examined code have been derived already in Section 2.1. However, the block length and dimension alone do not have a direct effect on the quality of the code, so they must be considered together in the form of the code rate R. Because of this, the code rate R is also listed alongside the other metrics in Table 5.2. Since all the codes are linear, according to Table 5.1, the properties from Lemma 2 apply to all of them.

Table 5.2: Block length n, dimension k, minimum Hamming distance d and code rate R of the discussed binary codes, own creation.

Code	n	k	d	R
$\mathrm{Ham}(r,2)$	2^r-1	2^r-r-1	3	$\frac{2^r-r-1}{2^r-1}$
G_{23}	23	12	7	$\frac{12}{23}$
G_{24}	24	12	8	$\frac{1}{2}$
$\mathrm{RM}(r,m)$	2^m	$\sum_{i=0}^{r}\binom{m}{i}$	2^{m-r}	$\frac{\sum_{i=0}^{r}\binom{m}{i}}{2^m}$
BCH	n	$n-\deg(g(x))$	$\geq \delta$	$\frac{n-\deg(g(x))}{n}$
RS	$q-1$	$q-\deg(g(x))-1$	$\deg(g(x))+1$	$\frac{q-\deg(g(x))-1}{q-1}$

One can immediately see that the code rate of the extended binary Golay code G_{24} from Section 2.1.1.2 is slightly worse than the code rate of the "normal" binary Golay code G_{23}. However, its minimum Hamming distance is better and allows the detection of one more error than G_{23}. Comparing the two values, we can see that the code rate is only $\approx 4\%$ worse, while the minimum Hamming distance improves by $\approx 14\%$, which is a worthwhile trade-off.

Alas, to compare the other code rates and distances, their respective formulas need to be evaluated. For this reason, different graphs have been created and are shown in Figure 5.1. It is very obvious that the code rate increases rather quickly for all the codes. However, for instance in the case of the binary Hamming codes, while the code rate surpasses 0.99 already at $r = 10$, its block length n is also already at 1023 whilst only providing a minimum distance of $d = 3$. In summary, only one error can be corrected within each block of 1023

bits which is not sufficient for most practical purposes. The famous $[7, 4]$-Hamming code, denoted as $\mathrm{Ham}(3, 2)$ using the notation introduced in Definition 9, has a code rate of only ≈ 0.57 while being able to correct a single error within each block of only 7 bits.

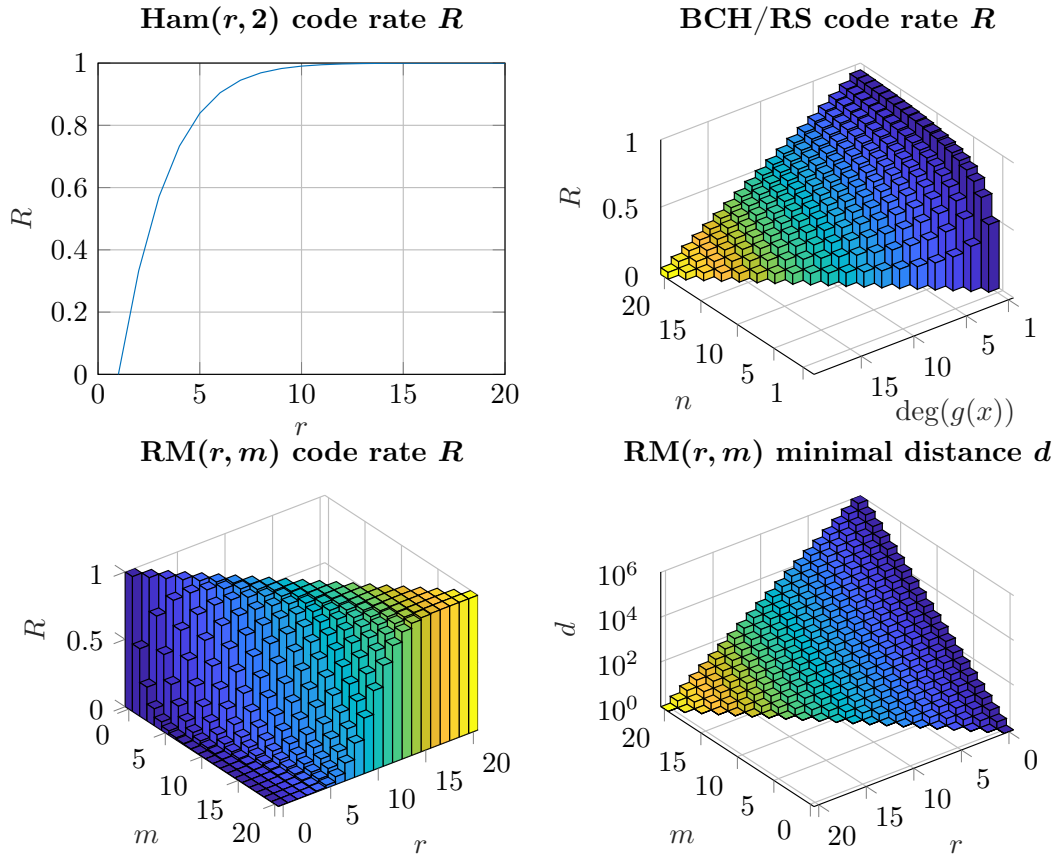


Figure 5.1: Code rate R and the minimum Hamming distance d of some investigated codes, own creation.

In conclusion, there seems to be a "sweet spot" which gives the best minimum Hamming distance d given its code rate R and block length n. The author proposes the following novel metric for a good first indicator:

Definition

The *Multiplicative metric* M is defined as the product

$$M := R \cdot \frac{d}{n} = \frac{k \cdot d}{n^2}.$$

It is maximal when both R and $\frac{d}{n}$ give very good values on their own.

It should also be mentioned that $\frac{d}{n}$ is often referred to as the *error-correction rate* [13] or *relative minimum distance* [130] of the code. For G_{23} and G_{24} this metric evaluates to $M \approx 0.159$ and $M \approx 0.167$ respectively, indicating that G_{24} is a bit better, as expected. For binary Hamming codes and RM codes, their respective values can be extracted from Figure 5.2.

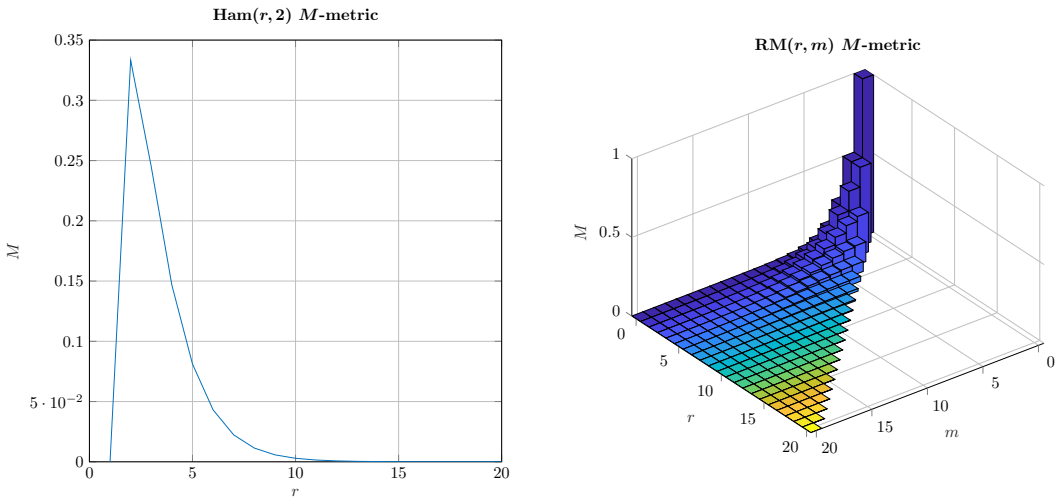


Figure 5.2: Visualisation of the M-metric for binary Hamming codes and RM codes, own creation.

It can be seen that this M-metric indicates that $\mathrm{Ham}(2,2)$ seems to be the best binary Hamming code, even before G_{24}. However, one should keep in mind that this M-metric is a very inconclusive metric which has its limits, especially when n is very low which is the case with $\mathrm{Ham}(2,2)$ where $n = 2^2 - 1 = 3$. In addition, $k = 2^2 - 2 - 1 = 1$, so only a single message bit can be encoded, requiring 2 redundancy bits. For this reason, $\mathrm{Ham}(3,2)$ is most likely still preferred and more metrics have to be considered. Unfortunately, removing the square exponent from the denominator does not solve the problem, because then there is no finite maximum.

Finally, it should be mentioned that in the context of fuzzy extractors, the size of the helper data is exactly $n-k$, *i.e.* the amount of redundant bits in any given codeword. Since $R = \frac{k}{n}$ is correlated with $n - k$, the M-metric is also somewhat practical in this scenario, and the previous rankings and values from Table 5.2 and Figures 5.1 and 5.2 still apply. Thus, the extended binary Golay code seems to give very good results.

5.3 Error Correction Capability

The minimum Hamming distance d parameter of each ECC directly determines how many errors can be detected and corrected by the respective code, according to Lemma 1. In conclusion, for example, both Golay codes can correct up to 3 errors, but detect 6 and 7 errors in the case of G_{23} or G_{24}, respectively. However, since the theoretical error correction capability can be easily determined using Lemma 1 and Table 5.2 together, this section shall focus on the practical capability instead.

For this reason, both the ultimate and binary Shannon limit from Theorems 82 and 83, respectively, are used. They set a limit on error-free communication within a noisy channel, which can also be used to represent the noise in PUF responses. In order to test the different ECCs in this regard, 10^6 bits have been randomly generated which are used to represent either one or many concatenated PUF responses. Using MATLAB [131] and its Communications Toolbox [132], noise using different E_b/N_0 levels has also been randomly generated and converted to discrete bits.

The simulated PUF response bit string is split into blocks of length k bits each (in the case of an ECC of dimension k) and encoded to a codeword. After concatenating all the codewords to a long bit string, the noise vector is applied to it by adding both vectors together. After that, the bit string is split again and each block of n bits is decoded into its original response and all the decoded responses are again concatenated. Since each computation here is done over the field $\mathbb{F}_2$, it suffices to add the original vector to the resulting vector and calculate its Hamming weight in order to determine the number of errors that have occurred. By dividing that amount by the number of bits, the BER is determined and illustrated in Figure 5.3.

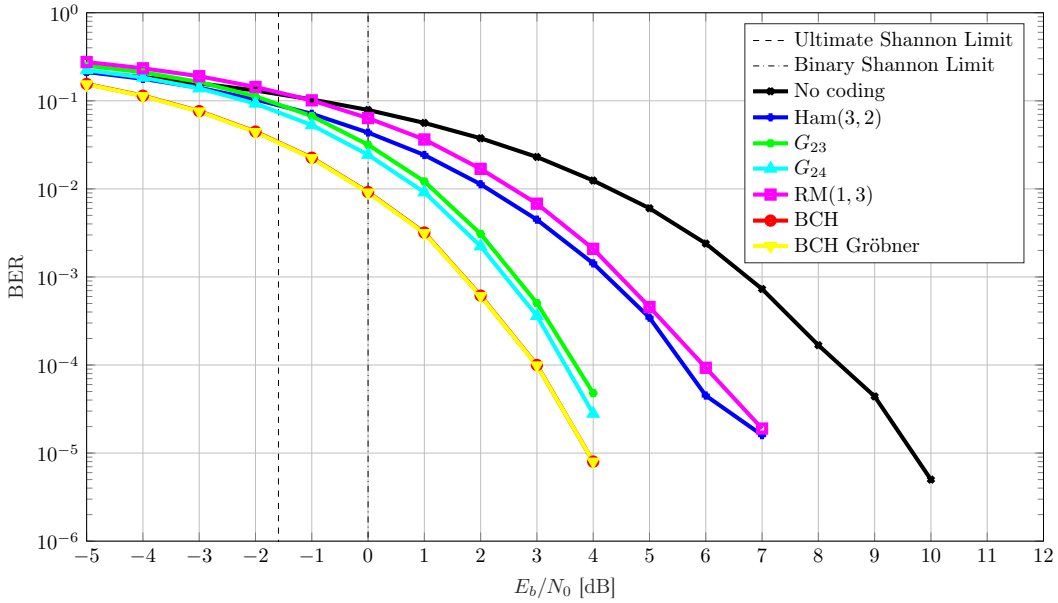


Figure 5.3: Comparison of the BER of some studied ECCs as a function of the noise level, own creation.

First, it is evident that the BCH code approach seems to yield the best error correction capability. Also, the Gröbner basis decoding approach provides only a marginal improvement over the standard Peterson-Gorenstein-Zierler algorithm. The examined BCH code is defined in the splitting field $(\mathbb{F}_2[\alpha]/\langle\alpha^4+\alpha+1\rangle)[x] \cong \mathbb{F}_{2^4}[x]$, has a designed distance of $\delta = 7$ and is narrow-sense. Hence, it has the generator polynomial $x^{10}+x^8+x^5+x^4+x^2+x+1$, according to Definition 33.

Also, G_{24} provides slightly better results than G_{23} due to its ability to detect an additional error which enables it to cancel the decoding process in some cases where G_{23} would introduce more errors. It can also be seen that none of the codes examined are close to the Shannon limit, unlike what some LDPC codes are able to achieve [133, 134]. Furthermore, the examined RM code RM$(1,3)$ and Hamming code Ham$(3,2)$ provide very similar error correction capabilities, with the latter being a bit better.

The burst error correction capability of BCH codes (due to their cyclicality) seems to have a huge effect on their rating which makes them far superior to the other codes examined.

Finally, it should be explained why no coding provides better results than most ECCs when the E_b/N_0 is less than $\approx -3\,\mathrm{dB}$. This is because the decoding algorithms of most ECCs introduce more errors than they correct if the noise level is too high.

5.4 Time Complexity

In this section, the theoretical time complexity of the various procedures is examined. First, the default encoding algorithms depend on a single operation, namely multiplication.

Specifically in the context of linear codes, a matrix-vector product needs to be computed. General matrix-matrix multiplication has a time complexity of $\mathcal{O}(n^3)$ when implemented using a naive algorithm. In the case of matrix-vector multiplication, this gets lowered down to $\mathcal{O}(n^2)$. However, using techniques such as erasure decoding [135] or the "Four Russians" method [135, 136] this run time can be reduced to $\mathcal{O}(n^2/\log n)$.

On the other hand, cyclic codes use polynomial multiplication for encoding. This can be done in $\mathcal{O}(n^2)$ using a trivial implementation. This run time can be lowered to $\mathcal{O}(n \log n)$ using the FFT [137]. However, the systematic encoder works in a different way. First, it does a multiplication of the input polynomial by some power of x which is a simple shift of the entire polynomial. After this, the normal remainder (from Definition 29) of the resulting polynomial with respect to the generator polynomial needs to be calculated. Since it has a time complexity of $\mathcal{O}(n^2)$, it is the computationally intensive step [138]. The last step is simple polynomial subtraction which can be accomplished in linear time. For this reason, the final time complexity for systematic encoding in cyclic codes is also $\mathcal{O}(n^2)$.

In terms of decoding, the time complexities are listed in Table 5.3. For binary Hamming codes, the implementation has a time complexity of $\mathcal{O}(n^2)$ due to the brute-force approach. It is worth mentioning that this time complexity is actually $\mathcal{O}(qn^2)$. However, since $q = 2$ and the vector divisions only need to try non-zero scalars, a single iteration is needed to decode binary Hamming codes. In addition, at the very beginning of the decoding process, a matrix-vector multiplication is performed, which, as already mentioned, also has a time complexity of $\mathcal{O}(n^2)$. Finally, it is worth noting that $n = 2^r$, according to Table 5.2, and hence the time complexity is even exponential in r.

For the binary Golay codes, two matrix-vector multiplications are again performed, which have a time complexity of $\mathcal{O}(n^2)$ each. All other operations have a complexity of either $\mathcal{O}(n)$, *e.g.*, vector addition, or $\mathcal{O}(n^2)$, *e.g.*, matrix transposition. Since $n \in \{23, 24\}$ is constant, one could argue that this time complexity is actually $\mathcal{O}(1)$, but that would be rather short-sighted.

The RM decoding algorithm depends on majority voting which is done for each row of the generator matrix. Using clever preprocessing as explained by Raaphorst [127], the decoding algorithm only needs to loop over r entries and perform $\mathcal{O}(n)$ operations in each iteration. But again, $n = 2^m$ and therefore the time complexity is exponential in m.

For the BCH and RS codes, the Peterson-Gorenstein-Zierler algorithm must compute the determinants of syndrome matrices. For this reason, the `det` function of CoCoALib chooses in the worst case the `DetByBareiss` approach, which is an implementation of the Bareiss algorithm [139] with a time complexity of $\mathcal{O}(n^3)$. However, this must be done several times, starting with a square matrix of size $\lfloor \frac{d-1}{2} \rfloor$ and decreasing the matrix size by 1 each iteration, as shown in Algorithm 13. Hence, the final worst-case time complexity is $\mathcal{O}(dn^3)$. It should be mentioned that the average time complexity is much better than that since `det` can also choose to call `DetOfSmallMat` which has explicit formulas for square matrices of size 2 to 5, using *e.g.*, the Rule of Sarrus.

Finally, the Gröbner basis decoding algorithm has some exponential time complexity which is not evaluated further in this work. This is due to the rather large number of indeterminates introduced to determine the error locations and, in the case of non-binary cyclic codes, even the error values.

Table 5.3: Collection of time complexities of the studied ECCs, own creation.

Code	Time complexity
$\mathrm{Ham}(r, 2)$	$\mathcal{O}(n^2)$
G_{23}	$\mathcal{O}(n^2)$
G_{24}	$\mathcal{O}(n^2)$
$\mathrm{RM}(r, m)$	$\mathcal{O}(nr)$
BCH/RS	$\mathcal{O}(dn^3)$
Cyclic Gröbner	*Exponential*[6]

It is important to mention that these time complexities are only valid for this specific implementation of the decoding procedures. The literature on the subject suggests that better algorithms have already been found that can decode many of the codes mentioned with less time complexity. For instance, determinants could be calculated with a time complexity of only $\mathcal{O}(n^{2.3728639})$, according to Le Gall [140]. Also, SDA decoding can give excellent decoding time complexity at the cost of table construction time complexity as high as $\mathcal{O}(8^n)$, according to Ellero *et al.* [138].

Overall, we can conclude that, considering only time complexity, "simple" codes such as Hamming codes, Golay codes and RM codes are preferable to more "complex" codes like BCH codes and RS codes.

5.5 Practical Run Time

To compare the practical runtime of the implementations, the same codes were chosen as in Section 5.3, and the timing measurements were actually performed simultaneously with the error correction capability measurements. Hence, the times in Figure 5.4 indicate how long it takes to both encode 10^6 bits and decode the resulting codewords.

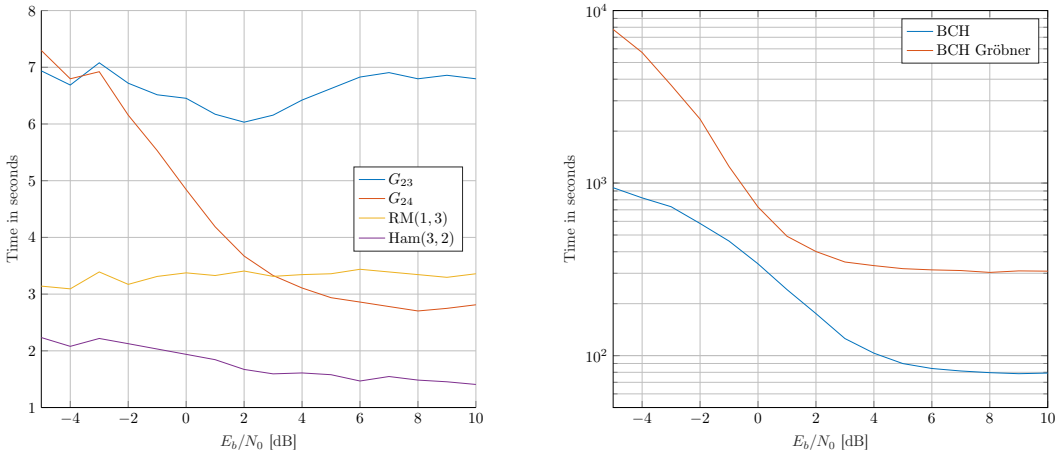


Figure 5.4: Comparison of some examined ECCs in terms of their practical running times, own creation.

[6] Various bounds exist, most of which are either exponential, *i.e.* $\mathcal{O}(\alpha^n)$, or double-exponential, *i.e.* $\mathcal{O}(\alpha^{\beta^n})$. For more details, see Bardet *et al.* [141].

There is a clear difference between two different groups of codes in this case. While the binary Golay, RM and Hamming codes take only a few seconds to complete the task at hand, the BCH code procedures take up to $\approx$ 2 hours. This is a clear indicator of the complexity of algebraic algorithms such as the Peterson-Gorenstein-Zierler algorithm and the Gröbner basis approach. It is worth noting that the latter algorithm takes the longest of all.

Furthermore, the procedures for G_{23} actually take longer than those for G_{24}. Using `callgrind`, which is part of the `valgrind` suite [142], this behaviour can be examined in detail. Then, KCachegrind [143] can be used to create call graphs showing the percentage of time spent on each function. Comparing Figures A.2 and A.3, it can be seen that $\approx 5\%$ of the runtime is spent on just `decodeG23`, which maps any given codeword from G_{23} to the corresponding one in G_{24}, and after decoding it with `decodeG24` maps it back to G_{23}. Alas, all the `callgrind` graphs from Figures A.1 to A.6 do not correspond to the measurements shown in Figure 5.4 since `callgrind` makes the code run ≈ 10 times slower which is infeasible for most of the procedures. Because of this, it is only a good guess to why G_{23} is slower than G_{24}. Overall, the fastest code by far is the Hamming code $\mathrm{Ham}(3, 2)$.

In addition, BCH and RS codes, G_{24} and also $\mathrm{Ham}(3, 2)$, appear to be susceptible to timing side-channel attacks [144] since, as noise levels decrease, so do decoding times. In the context of fuzzy extractors, this means that depending on the time it takes to decode a given response, an attacker can guess whether the given PUF response is similar to the correct one. As mentioned earlier in Section 3.1, Hiller *et al.* [10] also noted that such side-channels should be avoided. This can be done by adding operations to the decoding procedures that wait a certain amount of time so that the final decoding time is almost the same for each noise level. This obviously worsens the overall decoding times, but makes the fuzzy extractor safer in this respect.

5.6 Memory Usage

The `massif` memory profiler, which is part of the `valgrind` suite [142], has been used to assess the memory usage of each implementation. It is important to note that by default `massif` only profiles the heap and not the stack. However, with the `--stacks=yes` parameter, the stack is also profiled. Massif Visualizer [145] was then used to generate the graphs shown in Figures A.7 to A.12.

A quick analysis shows that binary Hamming codes use the least memory, closely followed by binary Golay codes and RM codes. Their respective memory peaks are around 100 KiB, 110 KiB and 125 KiB respectively. However, again the two algebraic procedures for the BCH and RS codes require the most memory, about 900 KiB for the Peterson-Gorenstein-Zierler algorithm and about 1.1 MiB for the Gröbner basis approach.

Clearly, memory usage can be used to determine whether a device is currently encoding or decoding with any ECC, but it is unclear whether this data can be used to determine how "close" a given PUF response is to the desired one. However, this should be investigated in more detail in future work.

5.7 Proposed Metrics

It is non-trivial to propose a single metric that captures the exact degree of applicability of different ECCs. Therefore, different metrics should be used to examine different aspects of the codes. This section provides a brief summary of the above metrics.

Section 5.1 features a brief comparison of linear and cyclic codes. However, as subsequent studies have shown, this had no immediate effect on the applicability of the various codes. It should be noted, however, that cyclic codes have advantages over linear codes that can be exploited in *e.g.* hardware implementations. For example, polynomial multiplication is much easier to implement in hardware, and the limited codeword space also makes decoding with *e.g.*, LFSRs as explained in Section 2.1.2.1, much simpler.

Then in Section 5.2 several metrics were presented. One of them is the code rate $R = \frac{k}{n}$, which gives the ratio of message bits to redundant bits of an ECC. However, as detailed analysis of Hamming codes has shown, the code rate can be arbitrarily close to 1 while the error correction rate $\frac{d}{n}$ deteriorates, which is proportional to the error correction capability, which has been studied in detail in Section 5.3, rendering the code useless. In conclusion, a trade-off between the two values is required, which can be explored using the newly defined M-metric with $M = \frac{k \cdot d}{n^2}$. Unfortunately, the metric is not yet fully applicable to arbitrary codes, since the square in the denominator distorts the results when n is rather small.

The time complexities from Section 5.4 have been shown to affect the performance of the codes in Section 5.5. For example, the exponential Gröbner basis approach performed the worst of all the different decoding schemes. Therefore, both the theoretical time complexity and the practical runtime of a code should be investigated in detail before it is deployed.

Finally, Section 5.6 has shown that the algebraic procedures, although still very low in memory consumption, require much more memory than the non-algebraic schemes. In typical IoT applications, most devices are very resource-constrained and this could be quite a big problem.

Depending on the use case, some of the metrics mentioned in this section may not be too important and can of course be ignored. In any case, however, the list by Hiller *et al.* [10] should be consulted before choosing an ECC for a given PUF.

6 Conclusion

In this thesis, a comprehensive comparison of six different classes of ECCs was made in the context of fuzzy extractors: Binary Hamming codes, both binary Golay codes, RM codes, and both binary BCH and RS codes.

Related work both on ECCs comparisons in general and in the context of fuzzy extractors or PUFs was then summarised. A very valuable work by Hiller *et al.* [10] was discovered and a small list was compiled with many tips from these authors on how to choose the best ECC for a given PUF. Then the CAS CoCoA-5 and its library CoCoALib were presented and some of their functionality was demonstrated.

Several metrics were then proposed and used to evaluate the suitability of each code for fuzzy extractors:

- *Linearity and cyclicality*: It was concluded that these properties do not directly affect the performance of the code, but only its applicability to certain use cases.
- *Code parameters*: Block length n, dimension k, minimum distance d and subsequently the code rate R were analysed. Trade-offs between these parameters determine the overall performance of the code. A new metric, the M metric, has therefore been proposed and briefly analysed.
- *Error correction capability*: Simulations showed the practical error correction performance of each code. BCH and hence also RS codes performed best, probably by exploiting their burst error correction capabilities.
- *Time complexity*: The theoretical complexity of encoding and decoding was discussed. Simpler codes such as Hamming and Golay codes are faster than algebraic codes such as BCH and RS codes.
- *Practical runtime*: Measurements confirmed the slower performance of algebraic procedures such as the Peterson-Gorenstein-Zierler algorithm and Gröbner bases. Binary Hamming codes were the fastest overall.
- *Memory usage*: Again, the algebraic codes used much more memory than the Hamming, Golay and RM codes.

It is also noted that some of these metrics may not be important and some may have a greater impact on the overall applicability of a given ECC in a given scenario. In conclusion, there is no single optimal code for all scenarios. The metrics and analyses provided in this thesis can guide the selection of the most appropriate code for a given fuzzy extractor application and its constraints. Important factors include the noise characteristics of the PUF responses,

available processing resources and security requirements. This work helps to highlight the strengths and weaknesses of each code in the context of fuzzy extractors, and provides a methodology for further analysis of ECCs for fuzzy extractors.

6.1 Future Work

Given the complexity of the issue, there is much room for improvement in many areas. For example, this thesis has only compared six different classes of ECCs, whereas the sheer number of available codes is too vast to comprehend. Thus, the proposed metrics should be applied and tested on other codes such as Polar codes [146], LDPC codes [147] (perhaps using the implementation of Mexis *et al.* [5, 108]) or Quadratic Residue codes [148] and so on.

In addition, a modern way of providing error correction is through machine learning. This has also been applied to PUFs, for example by Suragani *et al.* [149] or Mexis *et al.* [150]. As some of the proposed metrics are not directly applicable to such methods, they should either be omitted or new metrics should be proposed.

Furthermore, side-channel attacks on ECCs should be analysed, as they can pose a significant threat especially in secure environments due to possible key-stealing and other side effects. Careful consideration should be given to whether memory usage can be exploited as a side-channel [144].

Finally, detailed characterisations of PUFs in terms of *e.g.* spatial correlation effects [151] could also help to shed more light on the topic of BECCs in the context of fuzzy extractors or PUFs.

A Appendix

A.1 Callgrind Graphs

In this section, the various graphs produced using the `callgrind` tool of the `valgrind` suite [142] and KCachegrind [143] are presented, as described in Section 5.5. Figures A.1, A.2, A.3, A.4, A.5 and A.6 show the callgrind graphs for encoding and decoding the Hamming code, binary Golay code, extended binary Golay code, RM code and BCH code using the Peterson-Gorenstein-Zierler algorithm and the Gröbner basis decoding algorithm, respectively.

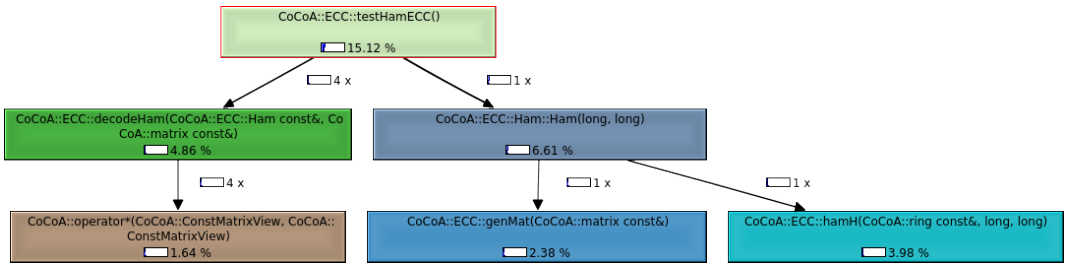


Figure A.1: Callgrind graph for binary Hamming codes, own creation using Valgrind [142] and KCachegrind [143].

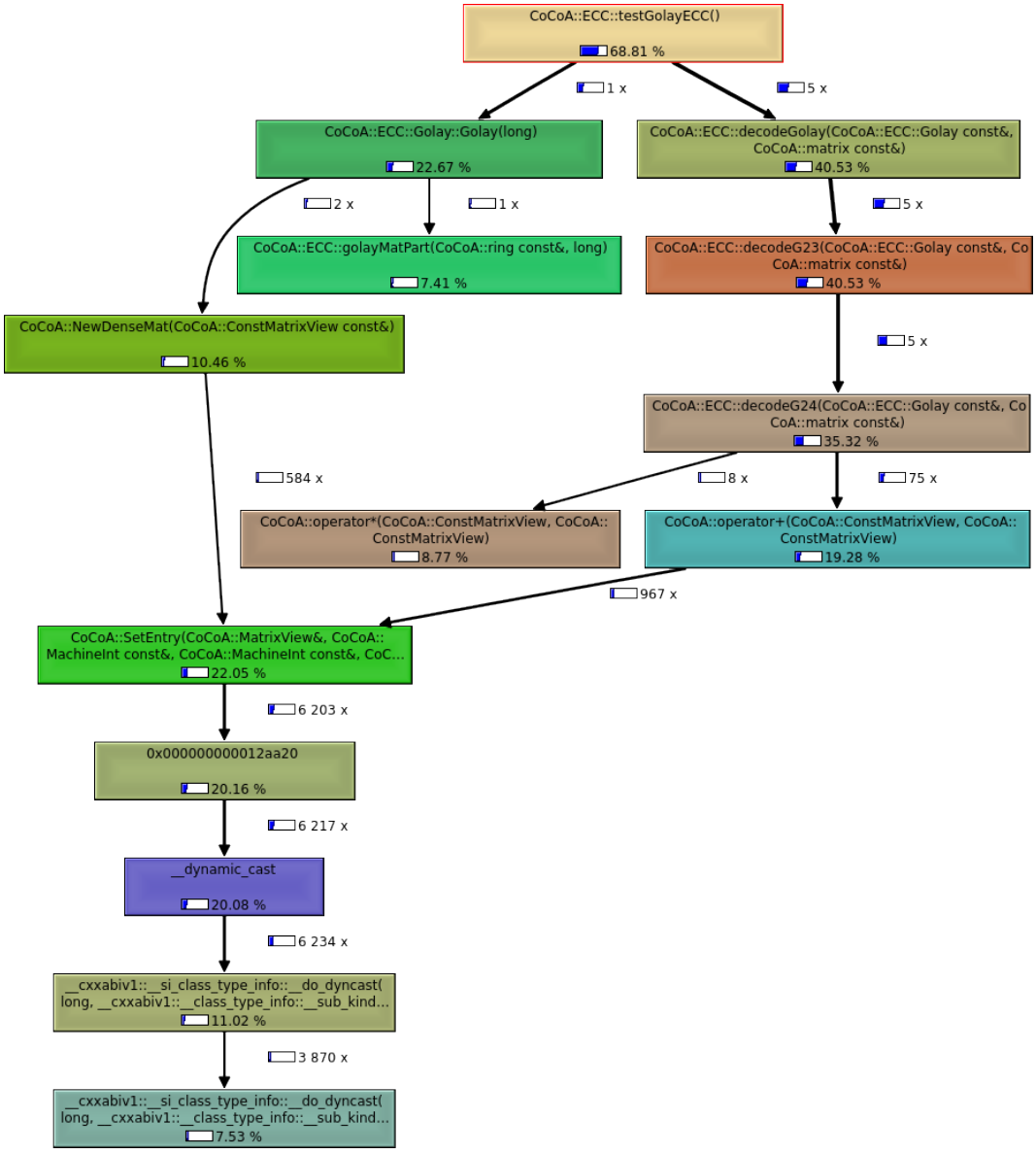


Figure A.2: Callgrind graph for the binary Golay code, own creation using Valgrind [142] and KCachegrind [143].

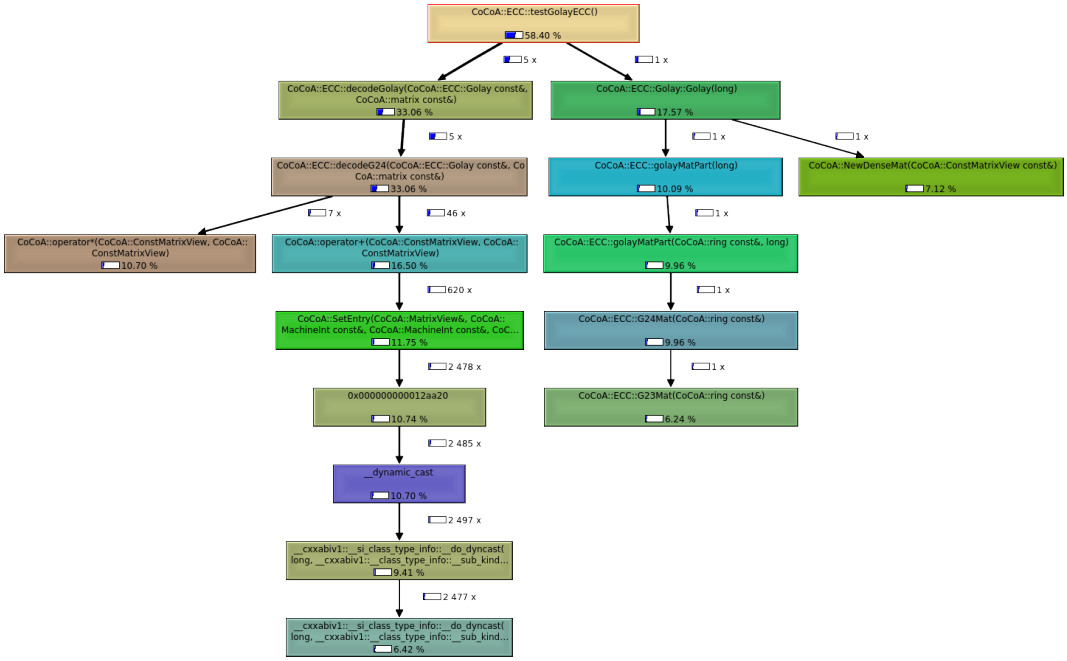


Figure A.3: Callgrind graph for the extended binary Golay code, own creation using Valgrind [142] and KCachegrind [143].

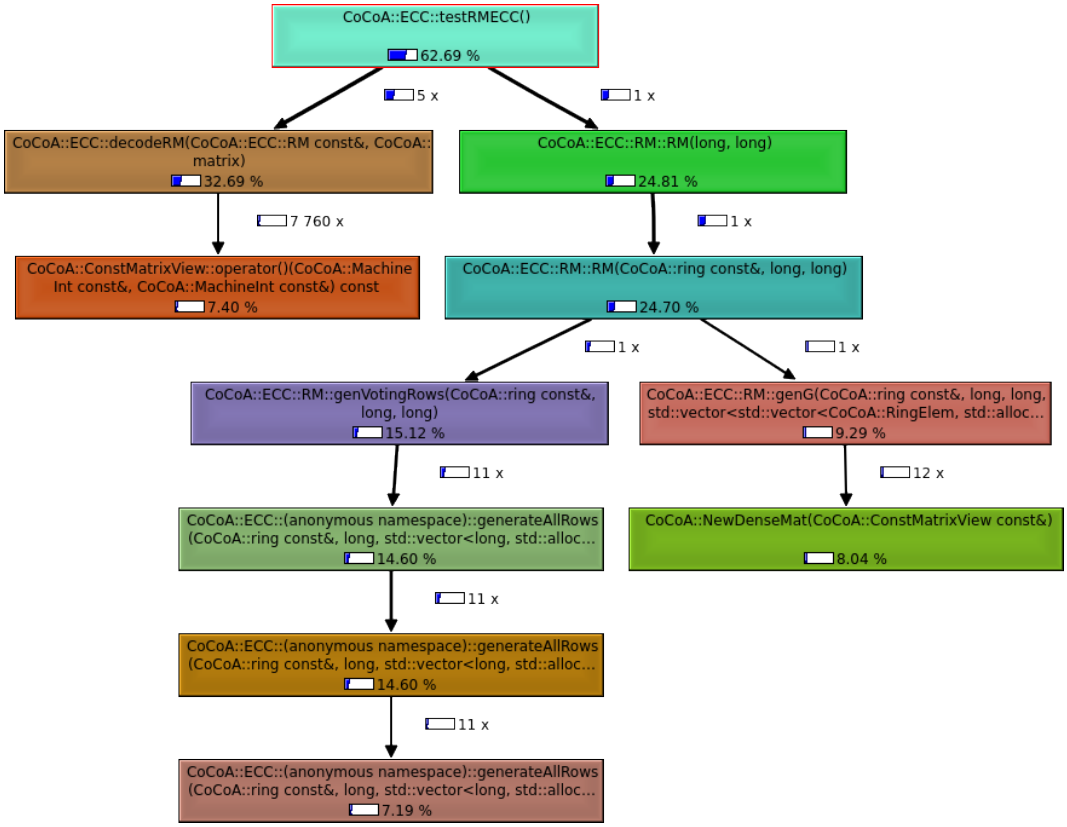


Figure A.4: Callgrind graph for RM codes, own creation using Valgrind [142] and KCachegrind [143].

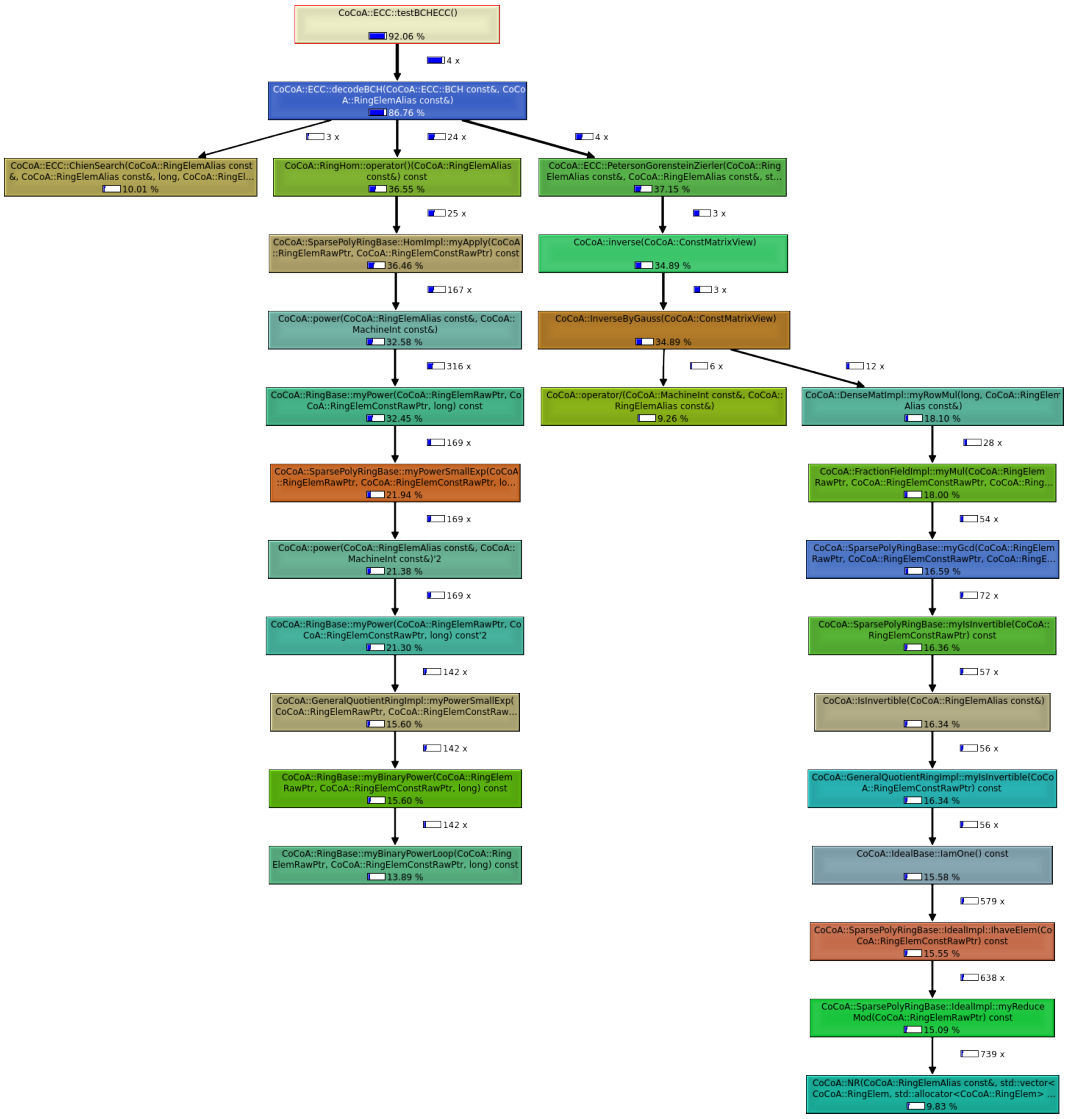


Figure A.5: Callgrind graph for binary BCH and RS codes, own creation using Valgrind [142] and KCachegrind [143].

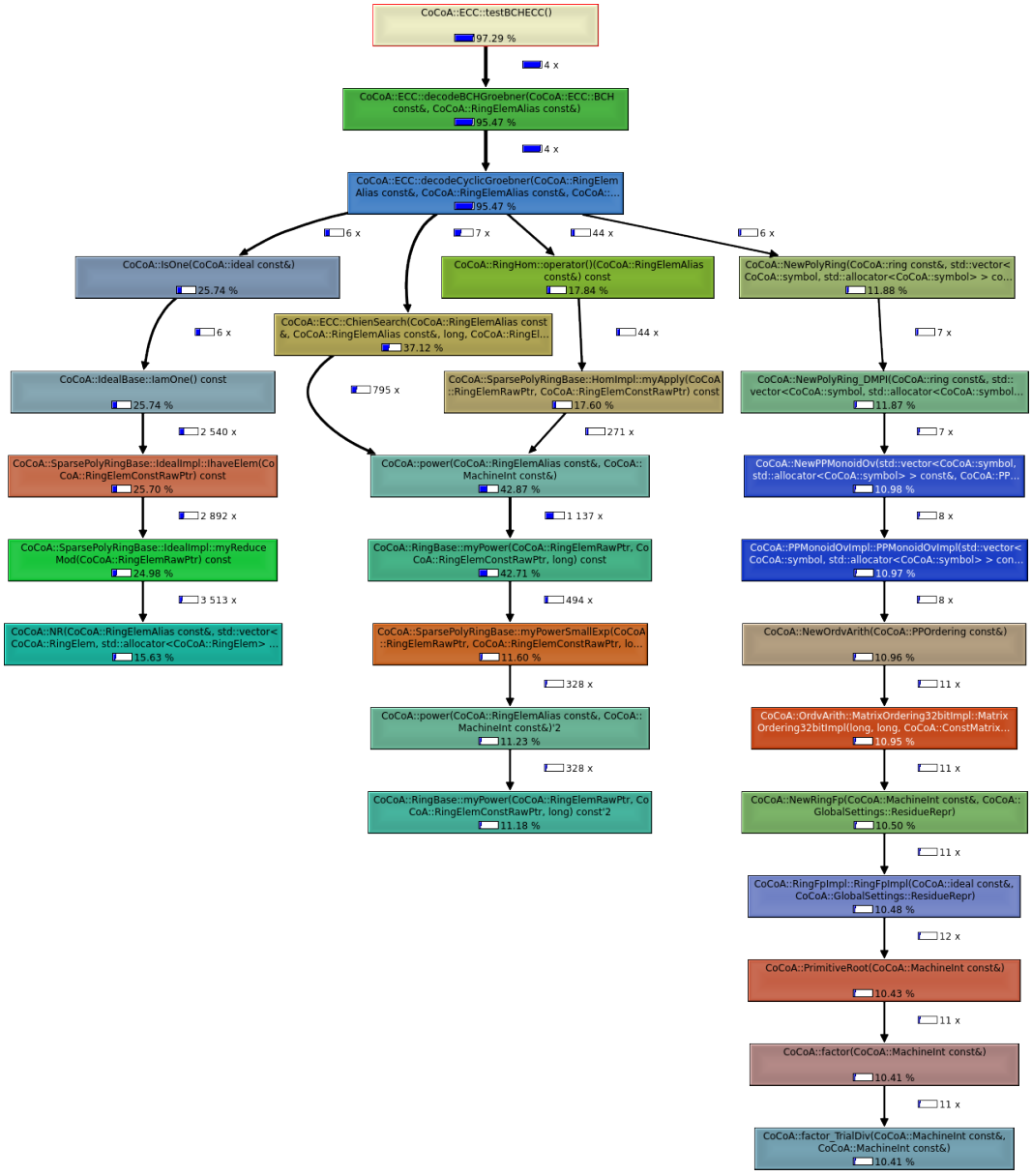


Figure A.6: Callgrind graph for binary BCH and RS codes using Gröbner basis decoding, own creation using Valgrind [142] and KCachegrind [143].

A.2 Massif Graphs

In this section, the various graphs produced using the `massif` tool of the `valgrind` suite [142] and Massif Visualizer [145] are presented, as described in Section 5.6. Figures A.7, A.8, A.9, A.10, A.11 and A.12 show the massif graphs for encoding and decoding the Hamming code, binary Golay code, extended binary Golay code, RM code and BCH code using the Peterson-Gorenstein-Zierler algorithm and the Gröbner basis decoding algorithm, respectively.

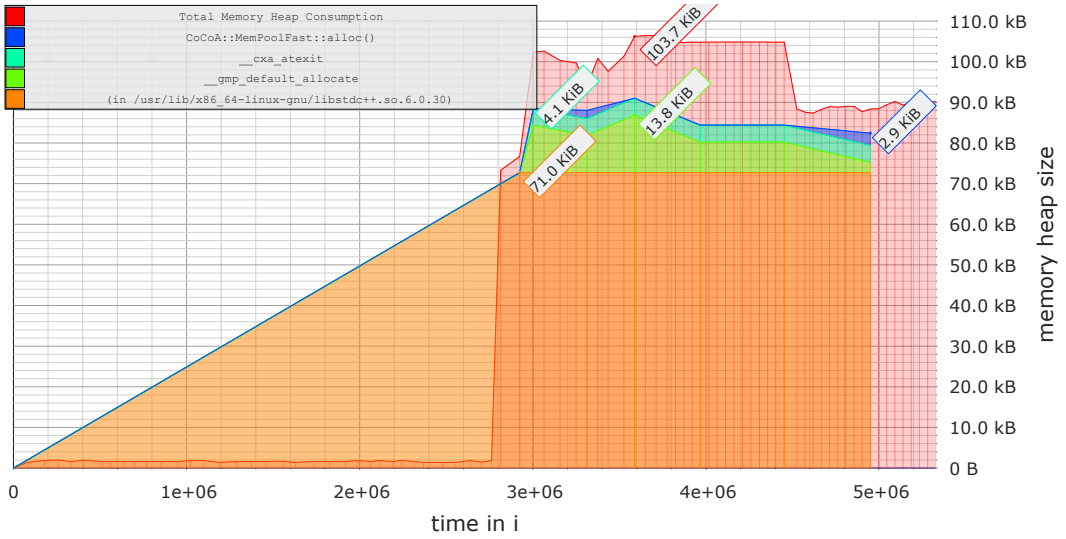


Figure A.7: Massif graph for binary Hamming codes, own creation using Valgrind [142] and Massif Visualizer [145].

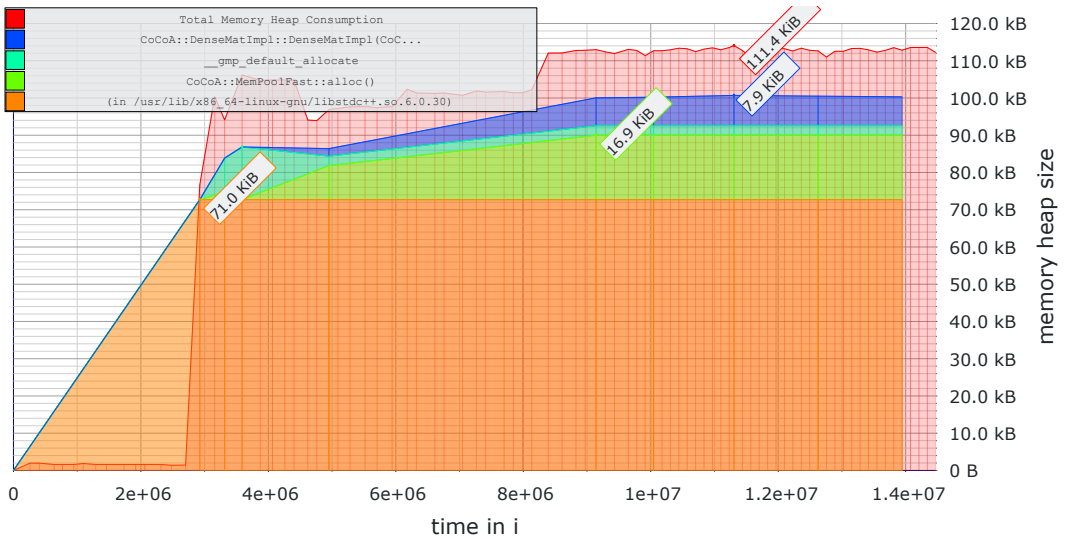


Figure A.8: Massif graph for the binary Golay code, own creation using Valgrind [142] and Massif Visualizer [145].

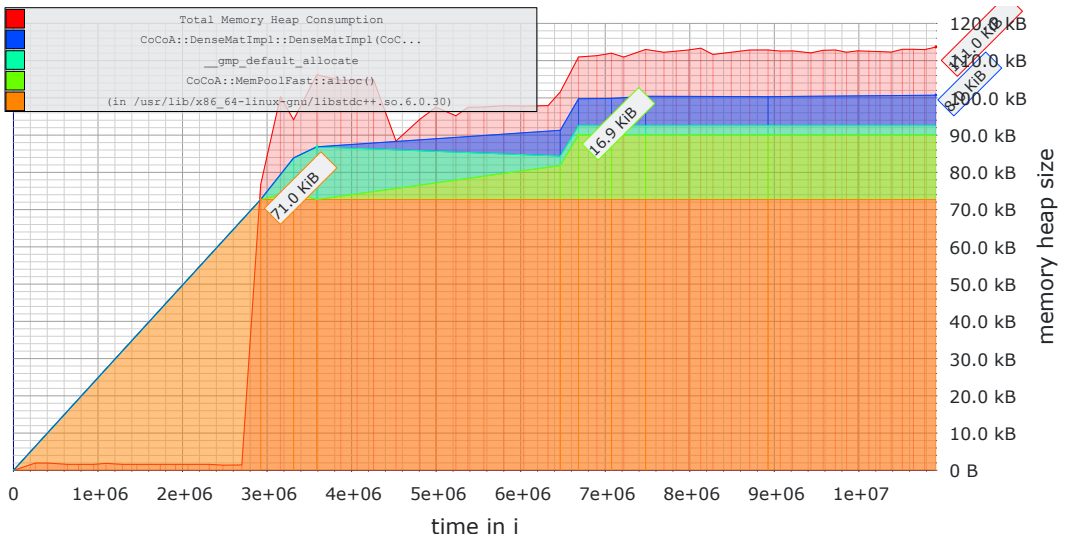


Figure A.9: Massif graph for the extended binary Golay code, own creation using Valgrind [142] and Massif Visualizer [145].

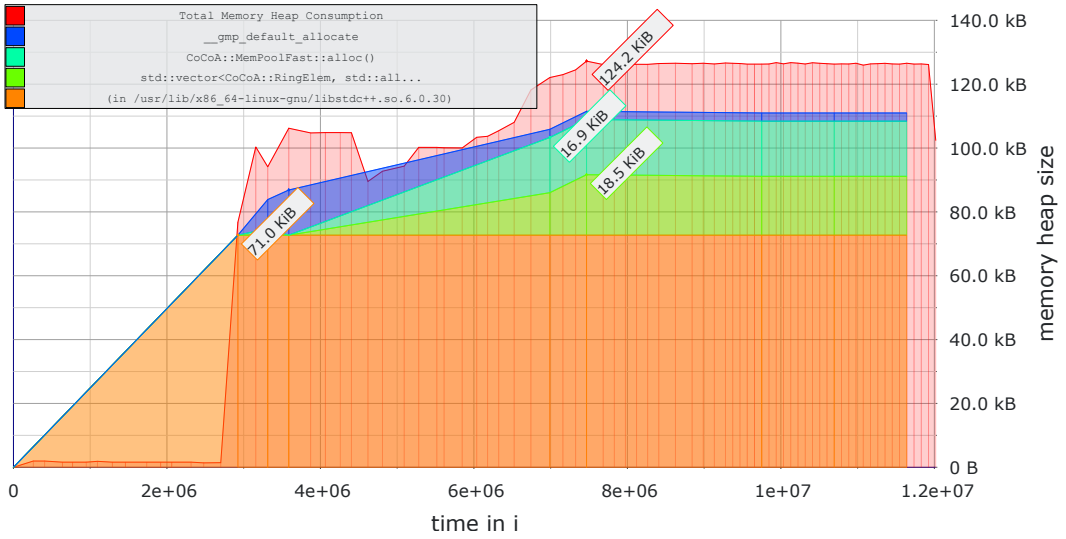


Figure A.10: Massif graph for RM codes, own creation using Valgrind [142] and Massif Visualizer [145].

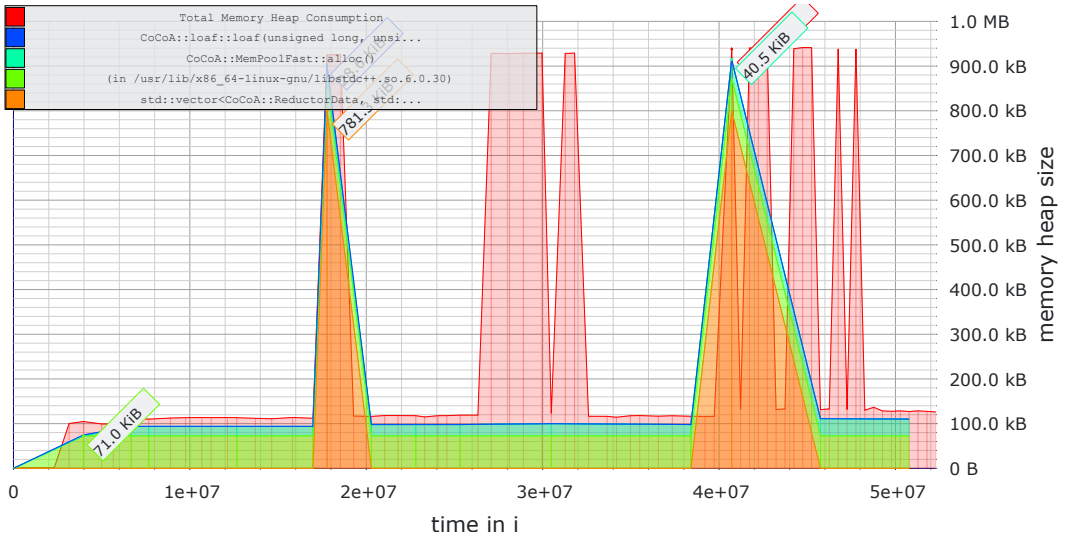


Figure A.11: Massif graph for binary BCH and RS codes, own creation using Valgrind [142] and Massif Visualizer [145].

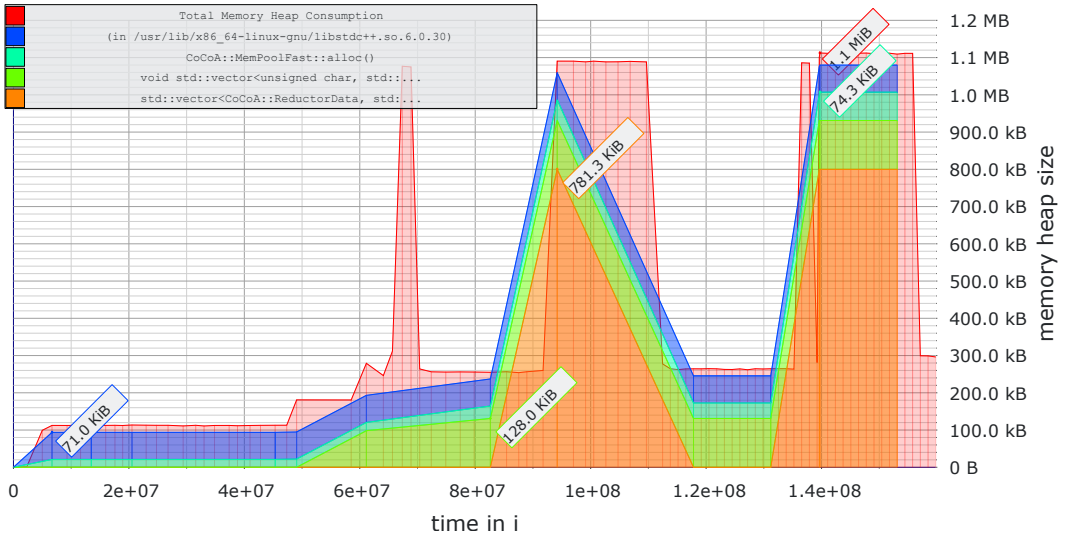


Figure A.12: Massif graph for binary BCH and RS codes using Gröbner basis decoding, own creation using Valgrind [142] and Massif Visualizer [145].

List of Figures

List of Tables

List of Algorithms

List of Symbols

List of Abbreviations

ACK ACKnowledgement 46
ARQ Automatic Repeat reQuest 45, 46

BCH Bose-Chaudhuri-Hocquenghem 33–38, 42, 45–51, 54, 60, 61, 64–66, 68–71, 73, 75, 78–80, 82, 84–86
BEC Binary Erasure Channel 47
BECC Burst Error Correction Code 32, 33, 74, 86
BER Bit Error Rate 47, 48, 68, 83
BI-AWGN Binary-Input Additive-White-Gaussian-Noise 40, 47, 48, 83
BP Belief Propagation 47, 48, 51

C-IBS Complementary Index-Based Syndrome coding 51
CAS Computer Algebra System 25, 52, 73
CD Compact Disc 38
CLI Command-Line Interface 52
CO Code-Offset 43, 51
CRC Cyclic Redundancy Check 45–48, 83

DTO Data Transfer Object 56, 57, 59, 63
DVD Digital Versatile/Video Disc 38

ECC Error Correction Code v, 1, 2, 4, 5, 10, 13, 21, 33, 38, 39, 41–48, 50–52, 55, 62, 64, 67, 68, 70–74, 83, 85
EEA Extended Euclidean Algorithm 34

FFT Fast Fourier Transform 34, 69
FPGA Field Programmable Gate Array 34, 50
FRAM Ferroelectric Random Access Memory 49, 50

GC Generalised Concatenated 40, 46–48
GMD Generalised Minimum Distance 46, 47

HA-SCL Hash-Aided Successive Cancellation List 48, 49
HARQ Hybrid Automatic Repeat reQuest 45, 46
HDM Helper Data Manipulation 50

IoT Internet of Things 2, 72

KFR Key Failure Rate 49, 50, 85

LDPC Low-Density Parity-Check 47, 51, 68, 74
LFSR Linear-Feedback Shift Register 34, 72

MDD Minimum Distance Decoding 6, 8, 10
MDS Maximum Distance Separable 38, 47
MIT Massachusetts Institute of Technology 54, 62
ML Maximum Likelihood 46, 47
MLSR Maximum-Likelihood Symbol Recovery 51
MRR Multiple Reference Response 49

NACK No ACKnowledgement 46

OS Operating System 54
OSD Ordered-Statistics Decoding 47

PUF Physical Unclonable Function v, 1, 2, 41, 44, 46–49, 51, 67, 68, 71–74, 85

QR Quick Response 38

RAM Random Access Memory 50, 55
RM Reed-Muller 21–25, 27, 45–48, 50, 51, 54, 59, 60, 64–71, 73, 75, 77, 80, 81, 83, 84, 86

RS Reed-Solomon 38, 41, 46–48, 51, 54, 61, 64–66, 69–71, 73, 78, 79, 82, 84

SC Successive Cancellation 48, 49
SDA Standard Decoding Array 9, 10, 70, 86
SHA Secure Hash Algorithm 42, 62, 63
SLLC Systematic Low-Leakage Coding 51
SNR Signal-to-Noise Ratio 40
SoS System of Systems 1, 2
SRAM Static Random Access Memory 48–50
SRR Single Reference Response 49

UML Unified Modeling Language 54

XOR eXclusive OR 55

Bibliography

[1] Stefan Katzenbeisser and André Schaller. Physical Unclonable Functions: Sicherheitseigenschaften und Anwendungen. *Datenschutz und Datensicherheit - DuD*, 36(12):881–885, November 2012. ISSN 1614-0702, 1862-2607. doi: 10.1007/s11623-012-0295-z. URL `http://link.springer.com/10.1007/s11623-012-0295-z`.

[2] Nico Mexis, Nikolaos Athanasios Anagnostopoulos, Shuai Chen, Jan Bambach, Tolga Arul, and Stefan Katzenbeisser. A lightweight architecture for hardware-based security in the emerging era of systems of systems. *ACM Journal on Emerging Technologies in Computing Systems*, 17(3):1–25, June 2021. ISSN 1550-4832. doi: 10.1145/3458824. URL `https://doi.org/10.1145/3458824`.

[3] Nico Mexis, Nikolaos Athanasios Anagnostopoulos, Shuai Chen, Jan Bambach, Tolga Arul, and Stefan Katzenbeisser. A design for a secure network of networks using a hardware and software co-engineering architecture. In *Proceedings of the SIGCOMM '21 Poster and Demo Sessions*, SIGCOMM '21, page 65–67, New York, NY, USA, August 2021. Association for Computing Machinery. ISBN 9781450386296. doi: 10.1145/3472716.3472849. URL `https://doi.org/10.1145/3472716.3472849`.

[4] Lenwood S. Heath and Nicholas A. Loehr. New algorithms for generating conway polynomials over finite fields. *Journal of Symbolic Computation*, 38(2):1003–1024, August 2004. ISSN 0747-7171. doi: 10.1016/j.jsc.2004.03.002. URL `https://doi.org/10.1016/j.jsc.2004.03.002`.

[5] Nico Mexis. Fuzzy extractors unter einsatz von low-density parity-check codes. Bachelor's thesis, University of Passau, Passau, July 2021. URL `https://femtopedia.de/theses/bachelor.pdf`.

[6] Jonathan I. Hall. *Notes on Coding Theory*. Michigan State University, January 2003.

[7] William Cary Huffman and Vera Pless. *Fundamentals of error-correcting codes*. Cambridge university press, 2003. ISBN 978-0-521-78280-7.

[8] Dieter Jungnickel. *Codierungstheorie*. Spektrum, Akad. Verl., 1995. ISBN 9783860254325.

[9] *FlexRay Communications System Protocol Specification*. FlexRay Consortium, October 2010. Version 3.0.1.

[10] Matthias Hiller, Ludwig Kürzinger, and Georg Sigl. Review of error correction for PUFs and evaluation on state-of-the-art FPGAs. *Journal of Cryptographic Engineering*, 10(3):229–247, May 2020. doi: 10.1007/s13389-020-00223-w. URL https://doi.org/10.1007/s13389-020-00223-w.

[11] Florence Jessie MacWilliams and Neil James Alexander Sloane. *The theory of error-correcting codes*, volume 16. Elsevier, 1977. ISBN 9780444851932.

[12] Richard Wesley Hamming. Error Detecting and Error Correcting Codes. *Bell System Technical Journal*, 29(2):147–160, April 1950. ISSN 00058580. doi: 10.1002/j.1538-7305.1950.tb00463.x. URL https://ieeexplore.ieee.org/document/6772729.

[13] Anton Betten, Harald Fripertinger, Adalbert Kerber, Alfred Wassermann, and Karl-Heinz Zimmermann. *Codierungstheorie: Konstruktion und Anwendung linearer Codes*. Springer-Verlag, 1998. ISBN 978-3-540-64502-3. doi: 10.1007/978-3-642-58973-7. URL https://doi.org/10.1007/978-3-642-58973-7.

[14] Ron Roth. *Introduction to Coding Theory*. Cambridge University Press, USA, 2006. ISBN 978-0-521-84504-5.

[15] Ralph-Hardo Schulz. *Codierungstheorie*. Vieweg+Teubner Verlag, 2003. doi: 10.1007/978-3-322-80328-3. URL https://doi.org/10.1007/978-3-322-80328-3.

[16] William Wesley Peterson and Edward J Weldon. *Error-correcting codes*. MIT press, second edition, 1972. ISBN 9780262527316.

[17] Werner Lütkebohmert. *Codierungstheorie*. Vieweg+Teubner Verlag, 2003. ISBN 978-3-528-03197-8. doi: 10.1007/978-3-322-80233-0. URL https://doi.org/10.1007/978-3-322-80233-0.

[18] Gerd Fischer. *Lineare Algebra: Eine Einführung für Studienanfänger*. Springer Fachmedien Wiesbaden, 18th edition, 2014. ISBN 9783658039448.

[19] DC Hankerson, Gary Hoffman, Douglas A Leonard, Charles C Lindner, Kevin T Phelps, Chris A Rodger, and James R Wall. *Coding theory and cryptography: the essentials*. CRC Press, 2000. ISBN 9780824704650.

[20] Lekh R Vermani. *Elements of Algebraic Coding Theory*, volume 12. CRC Press, 1996. ISBN 9780412573804.

[21] Elwyn Ralph Berlekamp, Robert James McEliece, and Henk C. A. van Tilborg. On the inherent intractability of certain coding problems (corresp.). *IEEE Transactions on Information Theory*, 24(3):384–386, May 1978. doi: 10.1109/tit.1978.1055873. URL https://doi.org/10.1109/tit.1978.1055873.

[22] E. G. Andrews. Telephone switching and the early bell laboratories computers. *The Bell System Technical Journal*, 42(2):341–353, March 1963. doi: 10.1002/j.1538-7305.1963.tb00503.x. URL https://doi.org/10.1002/j.1538-7305.1963.tb00503.x.

[23] Thomas M. Thompson. *From Error-Correcting Codes Through Sphere Packings to Simple Groups*, volume 21. Mathematical Association of America, 1 edition, December 1983. ISBN 9780883850237.

[24] Claude Elwood Shannon. A Mathematical Theory of Communication. *The Bell System Technical Journal*, 27(3):379–423, July 1948. ISSN 00058580. doi: 10.1002/j.1538-7305.1948.tb01338.x. URL https://ieeexplore.ieee.org/document/6773024.

[25] Jan Broulim, Alexander Ayriyan, Hovik Grigorian, and Vjaceslav Georgiev. OpenCL/CUDA algorithms for parallel decoding of any irregular LDPC code using GPU. *Telfor Journal*, 11(2):90–95, December 2019. doi: 10.5937/telfor1902090b. URL `https://doi.org/10.5937/telfor1902090b`.

[26] Irving Stoy Reed. A brief history of the development of error correcting codes. *Computers & Mathematics with Applications*, 39(11):89–93, 2000.

[27] Robert James McEliece. The reliability of computer memories. *Scientific American*, 252(1):88–95, January 1985. ISSN 00368733, 19467087.

[28] Jean Flower, Andrew Fish, and John Howse. Euler diagram generation. *Journal of Visual Languages & Computing*, 19(6):675–694, December 2008. doi: 10.1016/j.jvlc.2008.01.004. URL `https://doi.org/10.1016/j.jvlc.2008.01.004`.

[29] Mathuranathan Viswanathan. *Wireless Communication Systems in Matlab*. Independently published, 2020. ISBN 979-8648350779.

[30] Marcel Jules Edouard Golay. Notes on digital coding. *Proc. IRE*, 37:657, June 1949. URL `https://www.lama.univ-savoie.fr/pagesmembres/hyvernat/Enseignement/1718/info607/TP-Golay/golay_paper.pdf`.

[31] Vera Pless. A new decoding scheme for the ternary golay code. In *Proc. 20th Ann. Allerton Conf. Communication, Control, and Computing*, 1982.

[32] Efraín Soto Apolinar. Dodecahedron – tikz.net. Available at `https://tikz.net/dodecahedron/`, 2021. Published under CC BY-SA 4.0, `https://creativecommons.org/licenses/by-sa/4.0/`.

[33] Robert Turner Curtis. The regular dodecahedron and the binary golay code. *Ars Combinatoria*, 29:55–64, 1990.

[34] Robert Turner Curtis. Error-correction and the binary golay code. *London Mathematical Society Impact150 Stories*, 1:51–58, May 2016. URL `https://www.lms.ac.uk/sites/default/files/4.%20Curtis%20-%20Error%20Correction%20and%20the%20Binary-Golay%20Code.pdf`.

[35] Joe Fields. Decoding the golay code by hand, October 2000. URL `https://giam.southernct.edu/DecodingGolay/introduction.html`.

[36] Vera Pless. Decoding the golay codes. *IEEE Transactions on Information Theory*, 32 (4):561–567, July 1986.

[37] Jakob Steiner. Combinatorische aufgabe. *Journal für die reine und angewandte Mathematik*, 45:181–182, 1853.

[38] Robert Daniel Carmichael. Tactical configurations of rank two. *American Journal of Mathematics*, 53(1):217, January 1931. doi: 10.2307/2370885. URL `https://doi.org/10.2307/2370885`.

[39] Ernst Witt. Die 5-fach transitiven gruppen von mathieu. *Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg*, 12(1):256–264, December 1937.

[40] David Wilding. What's so special about the golay codes? Unpublished talk, March 2011. URL `https://www.yumpu.com/en/document/view/8442424/whats-so-special-about-the-golay-codes`.

[41] M. Blaum and J. Bruck. Decoding the golay code with venn diagrams. *IEEE Transactions on Information Theory*, 36(4):906–910, July 1990.

[42] Eric Elliot Johnson. *An Efficient Golay Codec for MIL-STD-188-141A and FED-STD-1045*. Department of Electrical and Computer Engineering, New Mexico State University, February 1991.

[43] Aamir Hussain, Muhammad Moaz Rais, and Mohammad Bilal Malik. Golay codes in ranging applications. In *Proceedings of the Eighth IASTED International Conference on Wireless and Optical Communications*, WOC '08, page 184–188, USA, May 2008. ACTA Press. ISBN 9780889867444.

[44] Mordecai Waegell and P. K. Aravind. Golay codes and quantum contextuality. *Phys. Rev. A*, 106:062421, December 2022. doi: 10.1103/PhysRevA.106.062421. URL `https://link.aps.org/doi/10.1103/PhysRevA.106.062421`.

[45] David Eugene Muller. Application of boolean algebra to switching circuit design and to error detection. *Transactions of the I.R.E. Professional Group on Electronic Computers*, EC-3(3):6–12, September 1954.

[46] Irving Stoy Reed. A class of multiple-error-correcting codes and the decoding scheme. *Transactions of the IRE Professional Group on Information Theory*, 4(4):38–49, September 1954.

[47] Marc Fossorier, Ravi Palanki, and Jonathan Yedidia. Iterative decoding of multi-step majority logic decodable codes. In *Proc. 3rd Int. Symp. Turbo Codes and Related Topics*, pages 125–132, Cambridge, Massachusetts, 2003.

[48] Edward C. Posner. Combinatorial structures in planetary reconnaissance. In H. B. Mann, editor, *Error Correcting Codes*, pages 15–46, New York-London-Sydney-Toronto, 1968. Wiley.

[49] J. H. van Lint. *Introduction to Coding Theory*. Springer Berlin Heidelberg, 1999. ISBN 978-3-642-63653-0.

[50] P. Delsarte, J.M. Goethals, and F.J. Mac Williams. On generalized ReedMuller codes and their relatives. *Information and Control*, 16(5):403–442, July 1970.

[51] M. Plotkin. Binary codes with specified minimum distance. *IEEE Transactions on Information Theory*, 6(4):445–450, September 1960.

[52] P. Gaborit, Jon-Lark Kim, and V. Pless. Decoding binary r(2, 5) by hand and by machine. In *Proceedings. 2001 IEEE International Symposium on Information Theory (IEEE Cat. No.01CH37252)*, page 86. IEEE, August 2001. doi: 10.1109/isit.2001.935949. URL `https://doi.org/10.1109/isit.2001.935949`.

[53] Philippe Gaborit, Jon-Lark Kim, and Vera Pless. Decoding binary r(2, 5) by hand. *Discrete Mathematics*, 264(1-3):55–73, March 2003. doi: 10.1016/s0012-365x(02)00550-2. URL `https://doi.org/10.1016/s0012-365x(02)00550-2`.

[54] D.G. Hoffman, D.A. Leonard, C.C. Lindner, K.T. Phelps, C.A. Rodger, and J.R. Wall. *Coding Theory: The Essentials*. Monographs and textbooks in pure and applied mathematics. M. Dekker, 1991. ISBN 9780824786113.

[55] Bruno Buchberger. *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einum nulldimensionalen Polynomideal*. PhD thesis, University of Innsbruck, Innsbruck, 1965.

[56] Bruno Buchberger. History and basic features of the critical-pair/completion procedure. *Journal of Symbolic Computation*, 3(1-2):3–38, 1987.

[57] Olle Abrahamsson. A gröbner basis algorithm for fast encoding of reed-müller codes, November 2016.

[58] Harinaivo Andriatahiny, Jean Jacques Ferdinand Randriamiarampanahy, and Toussaint Joseph Rabeherimanana. Decoding binary reed-muller codes via groebner bases, August 2018. URL `https://arxiv.org/abs/1808.09616`.

[59] James Joeph Sylvester. Lx. thoughts on inverse orthogonal matrices, simultaneous signsuccessions, and tessellated pavements in two or more colours, with applications to newton's rule, ornamental tile-work, and the theory of numbers. *Philosophical Magazine Series 1*, 34(232):461–475, 1867.

[60] Jacques Hadamard. Rèsolution d'une question relative aux dèterminants. *Bulletin des Sciences Mathématiques*, 2(17):240–246, 1893.

[61] R. E. Paley. On orthogonal matrices. *Journal of Mathematics and Physics*, 12:311–320, April 1933.

[62] Joesph L. Walsh. A closed set of normal orthogonal functions. *American Journal of Mathematics*, 45(1):5, January 1923.

[63] J. Meggitt. Error correcting codes and their implementation for data transmission systems. *IRE Transactions on Information Theory*, 7(4):234–244, October 1961.

[64] A Brinton Cooper III. Toward a new method of decoding algebraic codes using groebner bases. Technical report, ARMY RESEARCH LAB ABERDEEN PROVING GROUND MD, October 1993.

[65] Martin Kreuzer and Lorenzo Robbiano. *Computational Commutative Algebra 1*. Springer Berlin Heidelberg, 2000. doi: 10.1007/978-3-540-70628-1. URL `https://doi.org/10.1007/978-3-540-70628-1`.

[66] Muhammad Imran. Gröbner bases for decoding linear codes. Master's thesis, Universiteit Leiden, August 2019.

[67] Mario de Boer and Ruud Pellikaan. *The Golay Codes*, pages 338–347. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999. ISBN 978-3-662-03891-8. doi: 10.1007/978-3-662-03891-8_18. URL `https://doi.org/10.1007/978-3-662-03891-8_18`.

[68] Alexis Hocquenghem. Codes correcteurs d'erreurs. *Chiffres*, 2:147–156, September 1959.

[69] R.C. Bose and D.K. Ray-Chaudhuri. On a class of error correcting binary group codes. *Information and Control*, 3(1):68–79, March 1960. doi: 10.1016/s0019-9958(60)90287-4. URL `https://doi.org/10.1016/s0019-9958(60)90287-4`.

[70] K.-M. Cheung and F. Pollara. Phobos lander coding system: Software and analysis. *The Telecommunications and Data Acquisition Progress Report*, 42-94:274–286, August 1988. URL `https://ipnpr.jpl.nasa.gov/progress_report/42-94/94V.PDF`.

[71] Jiří Adámek. *Foundations of coding: Theory and applications of error-correcting codes with an introduction to cryptography and information theory*. John Wiley & Sons, 1991. ISBN 0-471-62187-0.

[72] Priya Mathew, Lismi Augustine, Sabarinath G., and Tomson Devis. Hardware implementation of (63, 51) BCH encoder and decoder for WBAN using LFSR and BMA. *CoRR*, abs/1408.2908, August 2014. doi: 10.48550/ARXIV.1408.2908. URL `https://arxiv.org/abs/1408.2908`.

[73] Sharad Shakya and Je-Hoon Lee. Implementation of a tree-type systolic array BCH encoder. In *Communications in Computer and Information Science*, pages 406–413. Springer Berlin Heidelberg, 2012. doi: 10.1007/978-3-642-35251-5_57. URL `https://doi.org/10.1007/978-3-642-35251-5_57`.

[74] Elwyn R Berlekamp. Non-binary bch decoding. Technical report, North Carolina State University. Dept. of Statistics, December 1966.

[75] E. Berlekamp. Nonbinary BCH decoding (abstr.). *IEEE Transactions on Information Theory*, 14(2):242–242, March 1968.

[76] J. Massey. Shift-register synthesis and BCH decoding. *IEEE Transactions on Information Theory*, 15(1):122–127, January 1969.

[77] Todd K. Moon. *Error Correction Coding: Mathematical Methods and Algorithms*. John Wiley & Sons, Ltd, 2005. ISBN 9780471648000.

[78] Yasuo Sugiyama, Masao Kasahara, Shigeichi Hirasawa, and Toshihiko Namekawa. A method for solving key equation for decoding goppa codes. *Information and Control*, 27(1):87–99, May 1975. ISSN 0019-9958.

[79] Daniel Gorenstein, W. Wesley Peterson, and Neal Zierler. Two-error correcting Bose-Chaudhuri codes are quasi-perfect. *Information and Control*, 3(3):291–294, September 1960.

[80] Daniel Gorenstein and Neal Zierler. A class of error-correcting codes in p^m symbols. *Journal of the Society for Industrial and Applied Mathematics*, 9(2):207–214, June 1961.

[81] R. Chien. Cyclic decoding procedures for bose- chaudhuri-hocquenghem codes. *IEEE Transactions on Information Theory*, 10(4):357–363, October 1964.

[82] E. R. Berlekamp. Factoring polynomials over finite fields. *Bell System Technical Journal*, 46(8):1853–1859, October 1967.

[83] David G. Cantor and Hans Zassenhaus. A new algorithm for factoring polynomials over finite fields. *Mathematics of Computation*, 36(154):587–592, April 1981.

[84] G. Forney. On decoding BCH codes. *IEEE Transactions on Information Theory*, 11 (4):549–557, October 1965.

[85] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, June 1960.

[86] Markus Hufschmid. Reed-solomon codes. In *Information und Kommunikation*, pages 77–86. Teubner, 2006. ISBN 978-3-8351-0122-7.

[87] ISO/IEC 18004:2015(E). Information technology – Automatic identification and data capture techniques – QR Code bar code symbology specification. Standard, International Organization for Standardization, Geneva, CH, February 2015.

[88] R. Singleton. Maximum distance q-nary codes. *IEEE Transactions on Information Theory*, 10(2):116–118, April 1964.

[89] George David Forney. Concatenated codes. *Research Laboratory of Electronics*, December 1965.

[90] Ylloh. File:concatenation of reed–solomon code with hadamard code.svg – wikimedia.org. Available at `https://commons.wikimedia.org/w/index.php?curid=24587747`, February 2013. Published under CC0 1.0 Universal Public Domain Dedication, `https://creativecommons.org/publicdomain/zero/1.0/`.

[91] È. L. Blokh and V. V. Zyablov. Coding of generalized concatenated codes. *Probl. Peredachi Inf.*, 10(3):45–50, 1974. ISSN 0555-2923.

[92] Ari Juels and Martin Wattenberg. A fuzzy commitment scheme. In *Proceedings of the 6th ACM conference on Computer and communications security - CCS '99*, pages 28–36, Kent Ridge Digital Labs, Singapore, 1999. Association for Computing Machinery. ISBN 9781581131482. doi: 10.1145/319709.319714. URL `http://portal.acm.org/citation.cfm?doid=319709.319714`.

[93] Ari Juels and Madhu Sudan. A fuzzy vault scheme. In *Proceedings IEEE International Symposium on Information Theory,*, page 408, Lausanne, Switzerland, 2002. IEEE. ISBN 9780780375017. doi: 10.1109/ISIT.2002.1023680. URL `http://ieeexplore.ieee.org/document/1023680/`.

[94] Yevgeniy Dodis, Leonid Reyzin, and Adam Smith. Fuzzy extractors: How to generate strong keys from biometrics and other noisy data. In Christian Cachin and Jan L. Camenisch, editors, *Advances in Cryptology - EUROCRYPT 2004*, pages 523–540, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN 978-3-540-24676-3.

[95] Yevgeniy Dodis, Leonid Reyzin, and Adam Smith. Fuzzy Extractors. In Pim Tuyls, Boris Skoric, and Tom Kevenaar, editors, *Security with Noisy Data*, pages 79–99. Springer London, London, 2007. ISBN 9781846289835 9781846289842. doi: 10.1007/978-1-84628-984-2_5. URL `http://link.springer.com/10.1007/978-1-84628-984-2_5`.

[96] André Schaller, Boris Škorić, and Stefan Katzenbeisser. On the Systematic Drift of Physically Unclonable Functions Due to Aging. In *Proceedings of the 5th International Workshop on Trustworthy Embedded Devices*, pages 15–20, Denver Colorado USA, October 2015. ACM. ISBN 9781450338288. doi: 10.1145/2808414.2808417. URL `https://dl.acm.org/doi/10.1145/2808414.2808417`.

[97] Stefan Katzenbeisser, Ünal Kocabaş, Vladimir Rožić, Ahmad-Reza Sadeghi, Ingrid Verbauwhede, and Christian Wachsmann. PUFs: Myth, Fact or Busted? A Security Evaluation of Physically Unclonable Functions (PUFs) Cast in Silicon. In *Cryptographic Hardware and Embedded Systems – CHES 2012*, volume 7428, pages 283–301. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 9783642330261 9783642330278. doi: 10.1007/978-3-642-33027-8_17. URL `http://link.springer.com/10.1007/978-3-642-33027-8_17`.

[98] Hyunho Kang, Y. Hori, T. Katashita, and M. Hagiwara. The Implementation of Fuzzy Extractor is Not Hard to Do: An Approach Using PUF Data. In *Proceedings of the 30th Symposium on Cryptography and Information Security, Kyoto, Japan*, pages 22–25, 2013. URL `https://www.jst.go.jp/crest/dvlsi/list/SCIS2013/pdf/SCIS2013_2E1-5.pdf`.

[99] National Institute of Standards and Technology. FIPS 180-2: Secure Hash Standard (SHS), August 2002.

[100] Anthony Van Herrewege, Stefan Katzenbeisser, Roel Maes, Roel Peeters, Ahmad-Reza Sadeghi, Ingrid Verbauwhede, and Christian Wachsmann. Reverse Fuzzy Extractors: Enabling Lightweight Mutual Authentication for PUF-Enabled RFIDs. In *Financial Cryptography and Data Security*, volume 7397, pages 374–389. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 9783642329456 9783642329463. doi: 10.1007/978-3-642-32946-3_27. URL `http://link.springer.com/10.1007/978-3-642-32946-3_27`.

[101] Jatinder Singh and Jaget Singh. A comparative study of error detection and correction coding techniques. In *2012 Second International Conference on Advanced Computing & Communication Technologies*. IEEE, January 2012. doi: 10.1109/acct.2012.2. URL https://doi.org/10.1109/acct.2012.2.

[102] Hans Peter Luhn. Computer for verifying numbers. Patent US2950048. *International Business Machines Corporation*, August 1960. URL https://patents.google.com/patent/US2950048.

[103] Sven Puchinger, Sven Müelich, Martin Bossert, Matthias Hiller, and Georg Sigl. On error correction for physical unclonable functions, January 2015. URL https://arxiv.org/abs/1501.06698.

[104] Sven Müelich, Sven Puchinger, Martin Bossert, Matthias Hiller, and Georg Sigl. Error correction for physical unclonable functions using generalized concatenated codes, July 2014. URL https://arxiv.org/abs/1407.8034.

[105] Roel Maes, Anthony Van Herrewege, and Ingrid Verbauwhede. PUFKY: A fully functional PUF-based cryptographic key generator. In *Cryptographic Hardware and Embedded Systems – CHES 2012*, pages 302–319. Springer Berlin Heidelberg, 2012. doi: 10.1007/978-3-642-33027-8_18. URL https://doi.org/10.1007/978-3-642-33027-8_18.

[106] Martin Bossert. *Channel Coding for Telecommunications*. Wiley, August 1999. ISBN 978-0-471-98277-7. URL http://eu.wiley.com/WileyCDA/WileyTitle/productCd-0471982776.html.

[107] J. Van Wonterghem, A. Alloum, J. J. Boutros, and M. Moeneclaey. Performance comparison of short-length error-correcting codes, November 2016. URL https://arxiv.org/abs/1609.07907.

[108] Nico Mexis, Nikolaos Athanasios Anagnostopoulos, Tolga Arul, Florian Frank, and Stefan Katzenbeisser. Fuzzy extractors using low-density parity-check codes. *crypto day matters 33*, September 2021. doi: 10.18420/CDM-2021-33-41. URL https://dl.gi.de/handle/20.500.12116/37559.

[109] Ashwija Reddy Korenda, Fatemeh Afghah, Bertrand Cambou, and Christopher Philabaum. A proof of concept SRAM-based physically unclonable function (PUF) key generation mechanism for IoT devices. In *2019 16th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON)*. IEEE, June 2019. doi: 10.1109/sahcn.2019.8824887. URL https://doi.org/10.1109/sahcn.2019.8824887.

[110] Hyunho Kang, Yohei Hori, Toshihiro Katashita, Manabu Hagiwara, and Keiichi Iwamura. Cryptographie key generation from PUF data using efficient fuzzy extractors. In *16th International Conference on Advanced Communication Technology*. Global IT Research Institute (GIRI), February 2014. doi: 10.1109/icact.2014.6778915. URL https://doi.org/10.1109/icact.2014.6778915.

[111] Yansong Gao, Yang Su, Lei Xu, and Damith C. Ranasinghe. Lightweight (reverse) fuzzy extractor with multiple reference PUF responses. *IEEE Transactions on Information Forensics and Security*, 14(7):1887–1901, July 2019. doi: 10.1109/tifs.2018.2886624. URL https://doi.org/10.1109/tifs.2018.2886624.

[112] Yang Su, Yansong Gao, Michael Chesser, Omid Kavehei, Alanson Sample, and Damith C. Ranasinghe. SecuCode: Intrinsic PUF entangled secure wireless code dissemination for computational RFID devices. *IEEE Transactions on Dependable and Secure Computing*, 18(4):1699–1717, July 2021. doi: 10.1109/tdsc.2019.2934438. URL https://doi.org/10.1109/tdsc.2019.2934438.

[113] Jeroen Delvaux and Ingrid Verbauwhede. Key-recovery attacks on various RO PUF constructions via helper data manipulation. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2014*. IEEE Conference Publications, April 2014. doi: 10.7873/date.2014.085. URL `https://doi.org/10.7873/date.2014.085`.

[114] Meng-Day Yu and Srinivas Devadas. Secure and robust error correction for physical unclonable functions. *IEEE Design & Test of Computers*, 27(1):48–65, January 2010. doi: 10.1109/mdt.2010.25. URL `https://doi.org/10.1109/mdt.2010.25`.

[115] Meng-Day Yu, Matthias Hiller, and Srinivas Devadas. Maximum-likelihood decoding of device-specific multi-bit symbols for reliable key generation. In *2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, May 2015. doi: 10.1109/hst.2015.7140233. URL `https://doi.org/10.1109/hst.2015.7140233`.

[116] John Abbott and Anna Maria Bigatti. CoCoALib: a C++ library for doing Computations in Commutative Algebra. Available at `https://cocoa.dima.unige.it/cocoa/cocoalib`, since 2007.

[117] John Abbott, Anna Maria Bigatti, and Lorenzo Robbiano. CoCoA: a system for doing Computations in Commutative Algebra. Available at `http://cocoa.dima.unige.it/cocoa`, since 1988.

[118] Alessandro Giovini and Gianfranco Niesi. CoCoA: A user-friendly system for commutative algebra. In Alfonso Miola, editor, *Design and Implementation of Symbolic Computation Systems*, pages 20–29, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg. ISBN 978-3-540-47014-4. doi: 10.1007/3-540-52531-9_120. URL `https://doi.org/10.1007/3-540-52531-9_120`.

[119] Antonio Capani. *The Design of the CoCoA 3 System*. PhD thesis, Università di Genova, February 2000. URL `http://www.disi.unige.it/dottorato/THESES/2000-02-CapaniA`. Available through the `WayBackMachine`.

[120] Steve Chamberlain and Cygnus Solutions. Cygwin: Get that linux feeling - on windows. Available at `https://cygwin.com/`, since 1995.

[121] Bill Hoffman, Ken Martin, Brad King, Dave Cole, Alexander Neundorf, and Clinton Stimpson. CMake: cross-platform make. Available at `https://cmake.org/`, since 2000.

[122] Stuart Feldman. Make, since 1976.

[123] EditorConfig Team. EditorConfig. Available at `https://editorconfig.org/`, since before 2011.

[124] Dimitri van Heesch. Doxygen. Available at `https://www.doxygen.nl/`, since 1997.

[125] AT&T Labs. Graphviz: Graph visualization tools. Available at `https://graphviz.org/`, since before 1991.

[126] J. D. Alanen and Donald E. Knuth. Tables of finite fields. *Sankhyā: The Indian Journal of Statistics, Series A (1961-2002)*, 26(4):305–328, December 1964.

[127] Sebastian Raaphorst. Reed-muller codes. Carleton University, originally hosted at `http://www.site.uottawa.ca/~raaphors/academic/mat5127paper.pdf`, May 2003. URL `https://github.com/sraaphorst/reed-muller-python/blob/master/reed_muller.pdf`.

[128] Jérémy Lambert. SHA256: A C++ SHA256 implementation. Available at `https://github.com/System-Glitch/SHA256`, since 2019.

[129] Tadao Kasami. Weight distributions of bose-chaudhuri-hocquenghem codes. *Coordinated Science Laboratory Report no. R-317*, August 1966. URL `https://www.ideals.illinois.edu/items/100343`.

[130] John Kerl. An introduction to coding theory for mathematics students. Lecture notes, September 2005. URL `https://johnkerl.org/doc/kerl-ecc-intro.pdf`.

[131] The MathWorks Inc. Matlab version: 9.9.0 (r2020b), 2020. URL `https://www.mathworks.com`.

[132] The MathWorks Inc. Communications toolbox version: 7.4 (r2020b), 2020. URL `https://www.mathworks.com`.

[133] Sae-Young Chung, G.D. Forney, T.J. Richardson, and R. Urbanke. On the design of low-density parity-check codes within 0.0045 dB of the Shannon limit. *IEEE Communications Letters*, 5(2):58–60, 2001. ISSN 1089-7798. doi: 10.1109/4234.905935. URL `http://ieeexplore.ieee.org/document/905935/`.

[134] Fatma A. Newagy, Yasmine A. Fahmy, and Magdi M. S. El-Soudani. Designing near Shannon limit LDPC codes using particle swarm optimization algorithm. In *2007 IEEE International Conference on Telecommunications and Malaysia International Conference on Communications*, pages 119–123. IEEE, May 2007. doi: 10.1109/ICTMICC.2007.4448612. URL `https://ieeexplore.ieee.org/document/4448612/`.

[135] Peter Trifonov. Matrix-vector multiplication via erasure decoding. In *Proceedings of XI International Symposium on Problems of Redundancy in Information and Control Systems*, pages 104–108, July 2007.

[136] V. L. Arlazarov, Y. A. Dinitz, M. A. Kronrod, and I. A. Faradzhev. On economical construction of the transitive closure of an oriented graph. *Doklady Akademii Nauk SSSR*, 194(3):487–488, February 1970.

[137] David Harvey and Joris van der Hoeven. Polynomial multiplication over finite fields in time o(n log n). *Journal of the ACM*, 69(2):1–40, March 2022. doi: 10.1145/3505584. URL `https://doi.org/10.1145/3505584`.

[138] F. Ellero, G. Palese, E. Tomat, and F. Vatta. Computational complexity analysis of hamming codes polynomial co-decoding. In *2021 International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, pages 1–6. IEEE, September 2021. doi: 10.23919/softcom52868.2021.9559071. URL `https://doi.org/10.23919/softcom52868.2021.9559071`.

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

[140] François Le Gall. Algebraic complexity theory and matrix multiplication. In *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation*, ISSAC '14, page 23, New York, NY, USA, July 2014. Association for Computing Machinery. ISBN 9781450325011. doi: 10.1145/2608628.2627493. URL `https://doi.org/10.1145/2608628.2627493`.

[141] Magali Bardet, Jean-Charles Faugère, and Bruno Salvy. On the complexity of the f5 gröbner basis algorithm. *Journal of Symbolic Computation*, 70:49–70, September 2015. ISSN 0747-7171. doi: 10.1016/j.jsc.2014.09.025. URL `https://doi.org/10.1016/j.jsc.2014.09.025`.

[142] Julian Seward and Other Valgrind Developers. Valgrind. Available at `https://valgrind.org/`, since 2000.

[143] Josef Weidendorfer. KCachegrind. Available at `https://kcachegrind.github.io/`, since 2002.

[144] Xinhui Lai, Maksim Jenihhin, Georgios Selimis, Sven Goossens, Roel Maes, and Kolin Paul. Early RTL analysis for SCA vulnerability in fuzzy extractors of memory-based PUF enabled devices. In *2020 IFIP/IEEE 28th International Conference on Very Large Scale Integration (VLSI-SOC)*, pages 16–21. IEEE, October 2020. doi: 10.1109/vlsi-soc46417.2020.9344071. URL `https://doi.org/10.1109/vlsi-soc46417.2020.9344071`.

[145] Milian Wolff. Massif Visualizer. Available at `https://apps.kde.org/de/massif-visualizer/`, since 2010.

[146] Erdal Arikan. Channel polarization: A method for constructing capacity-achieving codes for symmetric binary-input memoryless channels. *IEEE Transactions on Information Theory*, 55(7):3051–3073, July 2009. doi: 10.1109/tit.2009.2021379. URL `https://doi.org/10.1109/tit.2009.2021379`.

[147] Robert G. Gallager. *Low-Density Parity-Check Codes*. The MIT Press, 1963. ISBN 9780262256216. doi: 10.7551/mitpress/4347.001.0001. URL `https://direct.mit.edu/books/book/3867/Low-Density-Parity-Check-Codes`.

[148] E. Prange. Some cyclic error-correcting codes with simple decoding algorithms. Technical Report TN-58-156, Air Force Cambridge Research Center, Cambridge, MA, USA, 1958.

[149] Reshmi Suragani, Emiliia Nazarenko, Nikolaos Athanasios Anagnostopoulos, Nico Mexis, and Elif Bilge Kavun. Identification and classification of corrupted PUF responses via machine learning. In *2022 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 137–140. IEEE, June 2022. doi: 10.1109/host54066.2022.9839919. URL `https://doi.org/10.1109/host54066.2022.9839919`.

[150] Nico Mexis, Nikolaos Athanasios Anagnostopoulos, Tolga Arul, Elif Bilge Kavun, and Stefan Katzenbeisser. An improved machine-learning model for the identification and classification of memory-based puf responses. *crypto day matters 35*, submitted for publication.

[151] Nico Mexis, Tolga Arul, Nikolaos Athanasios Anagnostopoulos, Florian Frank, Simon Böttger, Martin Hartmann, Sascha Hermann, Elif Bilge Kavun, and Stefan Katzenbeisser. Spatial correlation in weak physical unclonable functions: A comprehensive overview. In *2023 26th Euromicro Conference on Digital System Design (DSD)*. IEEE, submitted for publication.