Error-correction coding in the MagiCtwin Diode

Lamers, Isa

Award date:
2022

Eindhoven University of Technology
Department of Mathematics and Computer Science

Error-correction coding in the MagiCtwin Diode

Bachelor final project Applied Mathematics

Isa Lamers

Supervisors:
Benne de Weger - Eindhoven University of Technology
Altan Kiliç - Eindhoven University of Technology
Alex Pellegrini - Eindhoven University of Technology
Berry Busser - Compumatica Secure Networks
Ries van Son - Compumatica Secure Networks

Thursday 29th December, 2022

Contents

Chapter 1

Introduction

This thesis is the final submission for the bachelor's program Applied Mathematics at Eindhoven University of Technology. It is supervised by Benne de Weger from research group Coding theory & Cryptology at Eindhoven University of Technology. The assessment committee was formed by Benne de Weger and Altan Kiliç. In addition, Alex Pellegrini has been of help in mathematical support. This thesis focuses on coding theory and the implementation of coding theory in a Dutch company called Compumatica. The project was overseen by Berry Busser, the direct supervisor at Compumatica, and Ries van Son, Chief Technology Officer and Compumatica's primary supervisor of this thesis.

In this thesis, we will elaborate on how Compumatica currently implements error-correcting coding in one of their products: the MagiCtwin Diode. The company had some questions as to how efficient the implemented error-correcting code in the MagiCtwin Diode is, and whether there were any paths towards improvement. We will therefore evaluate the efficiency of their implementation of error-correcting coding, and discuss various ways to make the coding more efficient. We will conclude with recommendations for improvements, possible substitutes and suggestions for further research.

In order to introduce error-correcting coding in the MagiCtwin Diode and make a meaningful assessment of the code's efficiency and its possible substitutes later in this thesis, we will kick off by introducing information theory. Information theory formalizes communication in a scientific manner. In this thesis we interpret communication as the transmission of information via a certain channel. After that, we will continue with a mathematical explanation of error-correcting coding, the detection and restoration of erasures in particular. Also, we will briefly touch the topic of interleaving. Subsequently, more details about Compumatica and the MagiCtwin Diode will be provided. This is directly followed by a thorough explanation of Compumatica's implementation of a Cauchy Reed-Solomon error-correcting code. Thereafter, we will assess the code's efficiency, suggest improvements and propose possible substitutes for the Cauchy Reed-Solomon code. We will conclude with a recommendation on how Compumatica can improve the efficiency of error-correcting coding in the MagiCtwin Diode.

Note: This report assumes the reader has a mathematical background that is comparable to that of a bachelor's student Applied Mathematics. This includes a working knowledge of linear algebra, algorithmic algebra and finite field arithmetic.

Chapter 2

Preliminaries

In this thesis we make use of finite fields of order p^m, where p is prime and $m \in \mathbb{N}_{>0}$. Finite fields are also called Galois Fields (Edwards, 1984). The mathematical notation for a finite field of order p^m is $\mathbb{F}_{p^m}$. A computer scientists might denote this same field by GF(p^m) (Galois Field of order p^m), but in this thesis we will adopt the mathematical notation $\mathbb{F}_{p^m}$.
In this thesis, we will consider the case where $p = 2$ and $m = 8$, meaning we look at $\mathbb{F}_{2^8} = \mathbb{F}_{256}$.

The finite field $\mathbb{F}_{256}$ is usually denoted by having a zero-element and elements that are powers of a primitive element α in the following way.

$$\mathbb{F}_{256} = \{0, 1, \alpha, \alpha^2, \ldots, \alpha^{254}\} \tag{2.1}$$

However, in this thesis we denote the finite field $\mathbb{F}_{256}$ and its elements in another way: using an isomorphism between $\mathbb{F}_{256}$ and $\mathbb{F}_2^8$. For this, first consider integers $0, 1, \ldots, 254, 255$ and their respective binary forms 00000000, 00000001, ..., 11111110, 11111111. We can put the binary forms in a vector $(a_7, a_6, a_5, a_4, a_3, a_2, a_1, a_0)$, this vector is an element in $\mathbb{F}_2^8$. We define the following isomorphism to formally represent mapping an element $a \in \mathbb{F}_{256}$ to a binary vector of length 8 in $\mathbb{F}_2^8$.

$$\varphi : \mathbb{F}_{256} \xrightarrow{\sim} \mathbb{F}_2^8, \quad a \mapsto (a_7, a_6, a_5, a_4, a_3, a_2, a_1, a_0) \tag{2.2}$$

Additionally, we can express an element $a \in \mathbb{F}_{256}$ as a polynomial in the indeterminate x by using $a_7, \ldots, a_0$ as coefficients in the following way.

$$a = a_7x^7 + a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0 \text{ with } a_i \in \mathbb{F}_2 \tag{2.3}$$

We have summarized the notations in Table 2.1.

Integer	**Binary form**	**Binary vector $\varphi(a)$**	**Polynomial**
0	00000000	(0,0,0,0,0,0,0,0)	0
1	00000001	(0,0,0,0,0,0,0,1)	1
2	00000010	(0,0,0,0,0,0,1,0)	x
⋮	⋮	⋮	⋮
254	11111110	(1,1,1,1,1,1,1,0)	$x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x$
255	11111111	(1,1,1,1,1,1,1,1)	$x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x + 1$

Table 2.1: Notation of the finite field $\mathbb{F}_{256}$.

The finite field $\mathbb{F}_{256}$ has 16 primitive, irreducible (generator) polynomials (Planteen, 2019) which we denote by $F_1, F_2, \ldots, F_{16}$. The primitive, irreducible (generator) polynomial of $\mathbb{F}_{256}$ that is used in this thesis is $F_4(x) = x^8 + x^6 + x^3 + x^2 + 1$, meaning that the binary vectors contain the coefficients of the polynomials in the ring $\mathbb{F}_{256} = \mathbb{F}_2[x]/_{(x^8+x^6+x^3+x^2+1)}$. The other 15 primitive, irreducible polynomials are included in Appendix A.

We will now briefly show how finite field arithmetic in $\mathbb{F}_{256}$ is done when using the notation introduced above. Let $a, b \in \mathbb{F}_{256}$ and consider their binary vector forms $\varphi(a), \varphi(b) \in \mathbb{F}_2^8$.
Adding $a + b$ and subtracting $a - b$ are done in the same way: by a bitwise exclusive OR-operation (XOR-operation) on the binary vector forms $\varphi(a)$ and $\varphi(b)$. The results of an XOR-operation are included in Table 2.2.

a_i	b_i	XOR$(a_i, b_i))$
0	0	0
0	1	1
1	0	1
1	1	0

Table 2.2: Results of the XOR-operator on elements $a_i, b_i \in \mathbb{F}_2$.

In order to explain multiplication and division, we introduce *exp* and *log* tables. For these tables we need generator polynomial $F_4(x) = x^8 + x^6 + x^3 + x^2 + 1$.
We create an *exp* table by putting $i = 0, \dots, 254$ in the first column and value $exp(i)$ in the second column, here $exp(i)$ is a value between 0 and 255. For $i = 0$, we use the convention that $exp(0) = 1$. The other values $exp(i)$ are constructed from $exp(i-1)$ as follows. We consider the binary vector representation of $exp(i-1)$ and shift all digits one position to the left. This corresponds to multiplying $exp(i-1)$ with 2. If $exp(i-1) \leq 127$, the next value $exp(i)$ will still have a binary vector representation of length 8, and $exp(i)$ is the integer value corresponding to that binary vector representation. However, if $exp(i-1) \geq 128$, the next value $exp(i)$ will have a binary vector representation of length 9 or greater. This means the binary vector representation is not an element of $\mathbb{F}_2^8$, so the integer value is not between 0 and 255. In this case we use generator polynomial $F_4(x)$ by applying a XOR-operation on the binary vector representation of $exp(i)$ and on $(1, 0, 1, 0, 0, 1, 1, 0, 1)$; the binary vector representation of $F_4(x)$. We translate the result of this XOR-operation back to an integer value, which is now between 0 and 255. A short representation of the *exp* table for $\mathbb{F}_{256}$ based on generator polynomial $F_4(x)$ is included below in Table 2.3a. The full *exp* table can be found in Appendix B.
The *log* table is created from the *exp* table. We use the values $exp(i)$ from the *exp* table and put them in the first column of the *log* table. We put their corresponding values i in the second column, and then sort the rows in the table such that the values in the first column are increasing. In other words, in the *log* table we have $j = exp(i)$ and $log(j) = log(exp(i)) = i$. A short representation of the *log* table for $\mathbb{F}_{256}$ based on generator polynomial $F_4(x)$ is included below in Table 2.3b. The full *log* table can be found in Appendix B.

i	$exp(i)$
0	1
1	2
2	4
3	8
$\vdots$	$\vdots$
253	83
254	166
255	*

(a) *exp* table

j	$log(j)$
0	*
1	0
2	1
3	23
$\vdots$	$\vdots$
253	133
254	200
255	161

(b) *log* table

Table 2.3: *exp* and *log* tables for $\mathbb{F}_{256}$ with generator polynomial $F_4(x) = x^8 + x^6 + x^3 + x^2 + 1$.

Note that in the exp table, there is no value for $exp(i)$ when $i = 255$, since $exp(255) = 1$, which is already the value for $i = 0$. We denote this by $exp(i) = *$ for $i = 255$. Similarly, there is no value i such that $exp(i) = 0$, which we denote by $*$ in the first row of the log table.

Using the exp and log tables, we will now explain how multiplication and division are done in $\mathbb{F}_{256}$. Let v and w denote the integer values of two elements a and b in $\mathbb{F}_{256}$ (first column in Table 2.1). For multiplication $v \cdot w$, we first add (according to 'normal' arithmetic) values $log(v)$ and $log(w)$ which we can find in the log table and reduce them modulo 255. We then find the exp value for that reduced result in the exp table.
For division $\frac{v}{w}$ we first subtract (according to 'normal' arithmetic) values $log(v)$ and $log(w)$, which we can find in the log table, then reduce them modulo 255. We then find the exp value for the reduced result in the exp table.

A summary of addition and subtraction of finite field elements a and b, and multiplication and division of their integer forms v and w is shown below.

Addition:	$a + b$	$= \mathrm{XOR}(\varphi(a), \varphi(b))$
Subtraction:	$a - b$	$= \mathrm{XOR}(\varphi(a), \varphi(b))$
Multiplication:	$v \cdot w$	$= exp(log(v) + log(w) \pmod{255})$
Division:	$\frac{v}{w}$	$= exp(log(v) - log(w) \pmod{255})$

Chapter 3

Information theory

In the 1940s, Claude Shannon published a seminal paper on the mathematical theory of communication (Shannon, 1948). He was the first to mathematically formalize communication theory. Nowadays, we refer to Shannon's theory of communication as *information theory.*
Information theory is a multidimensional topic in mathematics which has evolved from more than just Shannon's communication theory. For example, set theory, measure theory and probability theory also play a crucial role in today's information theory (Alencar, 2014). This combination of mathematical fields makes it an interesting, yet complicated specialty.
Countless applications of information theory exist, among which intelligence uses (Kahn, 1996), cryptography (Lagarias, 1993), cognitive neuroscience (Maurer, 2021) and coding theory (Clark & Cain, 1981). Coding theory is a mathematical field that provides ways to restore messages that have suffered errors during transmission from sender to receiver. In order to focus on coding theory, we will first treat relevant aspects of communication theory.

Let us start with a visual explanation of what we mean by *communication* by walking through a graphical representation of a communication system. In Figure 3.1, we show a schematic diagram of a general communication system, inspired by the schemes presented in Shannon's paper (Shannon, 1948) and Yates's tutorial on coding theory (Yates, 2009).

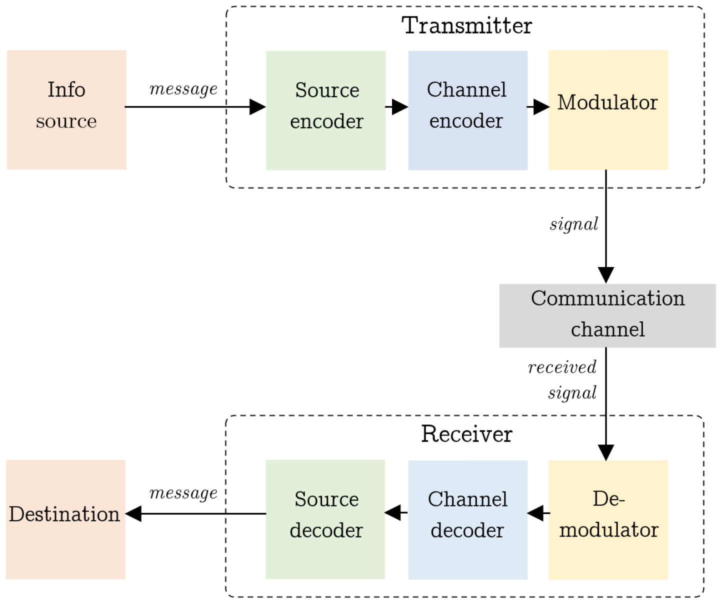


Figure 3.1: Schematic diagram of a communication system.

The communication process is initiated when the **information source** produces a *message* intended

for the receiver. The message then goes to the **transmitter**, which generally consists of three components. First, the **source encoder** compresses the data. Then the **channel encoder** improves the success of transmission of the source-encoded data through the channel by adding redundant data with an error-correcting code. The message and the redundant data together are called a *codeword*. Thirdly, the **modulator** converts the codeword into an output *signal* suitable for the communication channel. Examples of conversions that happen in the modulator are changing sound waves or text messages into electrical signals or light signals. Subsequently, the signal is sent over the **communication channel**. This channel is a medium, for example a pair of wires or a cable. This communication channel might be subject to noise coming from an external noise source, or noise generated by imperfections of the channel.
At the end of the (noisy) communication channel, the signal arrives at the **receiver**. This *received signal* might be different from the original signal because errors may have occurred. The receiver consists of the three similar components as the transmitter, but they are applied in the reverse order. First, the **demodulator** converts the received signal back to a codeword, the **channel decoder** detects errors and decodes the codeword back to a message using the same error-correcting code. It is possible that too many errors have occurred (see Proposition 1, in that case the message is not decoded but some error message is returned instead. The **source decoder** then decompresses the message. Finally, the receiver sends the original *message* to the **destination**.

Let us now take a more detailed look at the information source and the communication channel from a mathematical viewpoint.

3.1 Information source

The information source produces a message. This message contains certain content intended for the receiver. This content can be expressed and classified in different ways. We can distinguish *discrete* and *continuous* messages. The term 'discrete' refers to a system in which the message is a sequence of discrete symbols, for example letters, numbers, dots, dashes or spaces (Shannon, 1948). The set of discrete symbols of which a message is composed is called the input alphabet $\mathcal{X}$. In contrast, there exist continuous systems where the message is represented by a continuous function. Examples of this are classical, non-digital forms of communication such as radio and television (Shannon, 1948). In this thesis, we will focus on discrete information sources.

From now on we view the input alphabet $\mathcal{X}$ as a vector space and denote a message m of k symbols by a row-vector of length k over vector space $\mathcal{X}$, so $m = (m_0, \dots, m_{k-1})$, with $m_i \in \mathcal{X}$.

3.2 Communication channel

According to Shannon, a communication channel can be fully expressed by a collection of probabilities. We give the following definition of a communication channel, based on Ravagnani's lecture notes on coding theory (Ravagnani, 2022).

Definition 3.2.1. (**Communication channel**). The triple $\mathcal{K} = (\mathcal{X}, \mathcal{Y}, \mathbb{P})$ (uniquely) describes a communication channel. Here $\mathcal{X}$ is a finite non-empty set called the **input alphabet**, and $\mathcal{Y}$ is a finite non-empty set called the **output alphabet**. $\mathbb{P} : \mathcal{Y} \times \mathcal{X} \to \mathbb{R}$ is a function satisfying the following two properties:

1. Let $x \in \mathcal{X}$ and $y \in \mathcal{Y}$. Then $0 \leq \mathbb{P}(y \mid x) \leq 1$ meaning that the probability that y is received when x was sent lies between 0 and 1.
2. Let $x \in \mathcal{X}$. Then $\sum_{y \in \mathcal{Y}} \mathbb{P}(y \mid x) = 1$, meaning that if we send input symbol x, the symbol we receive must be in $\mathcal{Y}$.

The communication channel may have imperfections, be subject to noise from an external source, and be vulnerable to attacks from the outside. The consequence is that the message that the destination receives may not be the exact one that the information source sent; errors may have occurred. We distinguish two different types of errors that can occur to data when transmitted over a communication channel. The first type of error is when a symbol turns into another symbol: *symbol alterations*. The

second type of error is when a symbol is not altered but gets lost completely: called a *symbol erasure*. In particular situations, depending on what redundant data is sent along with the message by the error-correcting code, the receiver can determine the position where a symbol was erased. This is not guaranteed by all error-correcting codes. This thesis will assume knowing the position of an erased symbol. We will elaborate on this in Section 4.1.2.
The probabilities in Definition 3.2.1 describe how likely it is for a symbol x from the input alphabet $\mathcal{X}$ to turn into another symbol y from the output alphabet $\mathcal{Y}$ when transmitted over the communication channel $\mathcal{K} = (\mathcal{X}, \mathcal{Y}, \mathbb{P})$. In other words, this describes symbol alterations. The probabilities in the second property of Definition 3.2.1 sum to 1, therefore the communication channel that is defined only allows for symbol alterations, not for symbol erasures. This can be solved by adding an extra symbol that represents an erased symbol, for example '?', to the output alphabet $\mathcal{Y}$. This is done in this thesis as well.

A communication channel has several interesting features. One feature of interest is the quality of the communication channel. Literature often quantifies quality by looking at the signal-to-noise ratio, which we want to be as large as possible (Price & Goble, 1993). This ratio takes into account the power of a signal and the power of noise. Noise is defined as anything that interferes with communication over a channel. For the mathematical purposes of considering a communication channel, power is not an interesting quantity. What we will consider instead, is the ratio between the amount of data that is transmitted over the channel and the amount of data that is affected by noise, i.e. that suffer symbol alterations or erasures.

Definition 3.2.2. (Signal-to-noise ratio.) Let n be the number of symbols in the data that is sent over the communication channel. Let t be the number of symbols that have suffered errors (erasures or alterations) after transmission over the channel. Then the signal-to-noise ratio SNR is defined as follows:

$$SNR = \frac{n}{t}$$

In practice, source and destination often agree on a certain restriction on the set of admissible messages. In other words, the messages are composed from some admissible alphabet (Ravagnani, 2022). If the received message is not a message from the admissible alphabet, the receiver immediately knows something has gone wrong in the communication process. However, it may be that the received message does not exactly match the sent message, but is still a message that could have been admitted, i.e. is an element of the admissible alphabet. How can the receiver know now that the received message is incorrect?
One possibility is that the information source and the destination agree on some protocol to check whether their messages are the same. Such protocols require communication back and forth between source and destination. However, in some cases it is not possible or undesired for the destination to contact the information source due to security reasons. An example is when only one way of communication is enabled. This is the case in the MagiCtwin Diode. In situations like these, coding theory provides a way to detect and correct errors, without requiring back-and-forth communication between sender and receiver.

Chapter 4

Coding theory

When a communication channel is subject to noise from the outside or has imperfections in its design or hardware, errors can occur. For each codeword, the noise and imperfections cause a chance that during transmission over a channel, some symbols are altered or lost, resulting in errors in the received codeword (Clark & Cain, 1981). Depending on the use of the communication channel, and depending on the type of data sent over the channel, errors can raise a negligible or a far-reaching problem. Either way, errors are unwanted.
Making the communication channel perfect is an unfeasible task (Alzubi, Alzubi, & Chen, 2014), and improving it to minimize the impact of imperfections is costly (Mackay, 1995). Therefore we look for a solution that accepts the noisy, imperfect channel. This solution is error-correction coding. Error-correcting codes enforce reliable communication over an imperfect, noisy communication channel (Mackay, 1995).

4.1 Error-correcting coding

Error-correcting codes add redundant data to the message produced by the information source. This step happens in the channel encoder within the transmitter. This redundancy is used by the channel decoder to infer what the original message was. From here on we will focus on the channel encoder and channel decoder components from Figure 3.1. The redundant data both enables *detection* and *correction* of erroneous symbols.

Let us first establish that throughout this thesis, when mentioning (error-correcting) codes, we mean *linear* error-correcting codes. Linear codes have the property that any linear combination of codewords is also a codeword (Yates, 2009). Further details on linear codes are not relevant for the purpose of this thesis.
Let's take a look at the following definition of a linear error-correcting code, which is a combined version of definitions from Ravagnani's lecture notes (Ravagnani, 2022) and Bocklandt's lecture notes (Bocklandt, n.d.).

Definition 4.1.1. (**Linear error-correcting code**). Suppose we have a communication channel $\mathcal{K} = (\mathcal{X}, \mathcal{Y}, \mathbb{P})$. Let $q = p^m$ for p prime and $m \in \mathbb{N}$. Let $n, k \in \mathbb{N}$. A linear error-correcting code $\mathcal{C} \subseteq \mathcal{X}$ over the finite field $\mathbb{F}_q$ is a $k-$dimensional subspace of $\mathbb{F}_q^n$. The code $\mathcal{C}$ contains q^k distinct elements, which are called codewords. We say such a code $\mathcal{C}$ has length n and is an (n, k) linear code.

We will now give two important definitions that we need later in the thesis. They regard the distance between two codewords (van Lint, 1991) and the minimum distance of a code (van Lint, 1991).

Definition 4.1.2. (**Distance**). Let $\mathcal{X}$ be a finite non-empty set. Let $x, y \in \mathcal{X}^n$ be two elements of the vector space $\mathcal{X}^n$. The (Hamming) distance $d(x, y)$ between x and y is defined as follows.

$$d(x, y) = |\{i \mid 1 \leq i \leq n, x_i \neq y_i\}|$$

Definition 4.1.3. (**Minimum distance**). Let $\mathcal{C}$ be a linear (n, k) code. Then the minimum distance d_{min} of code $\mathcal{C}$ is the following.

$$d_{min} = min\{d(x, y) \mid x \in \mathcal{C}, y \in \mathcal{C}, x \neq y\}$$

A linear code can be represented in two manners: by a generator matrix or by a generator polynomial (Geisel, 2012). In this thesis we will adopt the matrix notation, because it is best suited for hardware and software implementations of error-correcting codes (Geisel, 2012), which is what we focus on in this thesis. Definition 4.1.4 and 4.1.5 characterize how a code is represented using matrices (Ravagnani, 2022).

Definition 4.1.4. (Generator matrix). Suppose we have an (n, k) linear error-correcting code $\mathcal{C}$. A $k \times n$ matrix G with entries in $\mathbb{F}_q$ is a generator matrix of code $\mathcal{C}$ if it has full rank and if its rows generate $\mathcal{C}$ over $\mathbb{F}_q$. In other words, the rows of the generator matrix are a basis of $\mathcal{C}$. We then say that code $\mathcal{C}$ is generated by G.

Definition 4.1.5. (**Standard generator matrix**). Suppose we have an (n, k) linear error-correcting code $\mathcal{C}$ that is represented by a generator matrix G. The reduced row-echolon form of G, denoted as G_{RREF}, is called the standard generator matrix of $\mathcal{C}$.

The generator matrix G of a code $\mathcal{C}$ is not unique, as $\mathcal{C}$ can be represented by different generator matrices G. This changes when considering the reduced row-echelon form of G: G_{RREF} is unique for a code $\mathcal{C}$ (Bocklandt, n.d.).

We distinguish different types of linear error-correcting codes. The first distinction we consider is between *block codes* and *convolutional codes.* Block codes encode messages that have a fixed length k, whereas convolutional codes can encode symbol streams of arbitrary lengths. This thesis focuses on block codes. One particular category of linear block codes are Reed-Solomon error-correcting codes (or RS codes for short) (Reed & Solomon, 1960). A formal definition can be found in paragraph 6.8 from (van Lint, 1991). We do not include it here because it is too cumbersome for the purposes of this thesis. One thing that is relevant to mention is that the minimum distance d_{min} of Reed-Solomon code $\mathcal{C}$ is $d_{min} = n - k + 1$.
Instead of giving the full definition, we will give an example of an RS code in Example 4.1.1.

Example 4.1.1. (Reed-Solomon code). Let $m \in \mathbb{N}$, take $p = 2$ and consider the finite field $\mathbb{F}_{p^m} = \mathbb{F}_{2^m}$. Let $\mathcal{C}$ be an (n, k) Reed-Solomon code over $\mathbb{F}_{2^m}$. Then one example of its generator matrix is a $k \times n$ Vandermonde matrix (Moorhouse, 2008), as shown below.

$$G = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ \alpha_0 & \alpha_1 & \alpha_2 & \cdots & \alpha_{n-1} \\ \alpha_0^2 & \alpha_1^2 & \alpha_2^2 & \cdots & \alpha_{n-1}^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \alpha_0^{k-1} & \alpha_1^{k-1} & \alpha_2^{k-1} & \cdots & \alpha_{n-1}^{k-1} \end{pmatrix} \text{ where } \alpha_i \in \mathbb{F}_{2^m}$$

Reducing G to its Row-Reduced Echelon Form (RREF) we obtain the following $k \times n$ standard generator matrix G_{RREF}.

$$G_{\text{RREF}} = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 & g_{0,k} & \cdots & g_{0,n-1} \\ 0 & 1 & 0 & \cdots & 0 & g_{1,k} & \cdots & g_{1,n-1} \\ 0 & 0 & 1 & \cdots & 0 & g_{2,k} & \cdots & g_{2,n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 & g_{k-1,k} & \cdots & g_{k-1,n-1} \end{pmatrix} \text{ where } g_{i,j} \in \mathbb{F}_{2^m}$$

The error-correcting code that is implemented in the MagiCtwin Diode is a specialized form of a Reed-Solomon code: a *Cauchy Reed-Solomon code.* Cauchy Reed-Solomon (CRS) codes have the same parameters n and k (Plank, 2005). The differences between a CRS code and an RS code lie in the format of data it encodes and in how the generator matrix is established, which will be explained in the Section 4.1.1.

We will not focus on the encoding and decoding steps of CRS codes.

4.1.1 Encoding

Suppose we have an (n, k) Cauchy Reed-Solomon code with standard generator matrix G_{RREF}. Let $m = (m_0, \ldots, m_k)$ be a message. When considering regular Reed-Solomon coding, each element $m_i \in$ would be one input digit. For Cauchy Reed-Solomon coding, we consider messages where each m_i is a block of b words of w binary digits (*bits*) (Blomer et al., 1999).

When we say we encode a message m with an (n, k) Cauchy Reed-Solomon code $\mathcal{C}$ that is represented by a standard generator matrix G_{RREF}, we mean that we assign to the message m a unique codeword c. This is done by viewing the message $m = (m_0, \ldots, m_{k-1})$ as a row vector and multiplying it by the $k \times n$ standard generator matrix G_{RREF} of $\mathcal{C}$. The result of this multiplication is a codeword $c = (c_0, \ldots, c_{n-1})$ (Bocklandt, n.d.). The codeword also consists of blocks c_i that are composed of b words of w bits each (Blomer et al., 1999). This encoding step is shown below.

$$c = m \cdot G_{\text{RREF}} = (m_0, \ldots, m_{k-1}) \cdot \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 & g_{0,k} & \cdots & g_{0,n-1} \\ 0 & 1 & 0 & \cdots & 0 & g_{1,k} & \cdots & g_{1,n-1} \\ 0 & 0 & 1 & \cdots & 0 & g_{2,k} & \cdots & g_{2,n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 & g_{k-1,k} & \cdots & g_{k-1,n-1} \end{pmatrix} = (c_0, \ldots, c_{n-1})$$

Here $g_{j,i}$ are elements of $\mathbb{F}_{2^m}$. Note that because the first part of G_{RREF} is the $k \times k$ identity matrix, the first k blocks of c correspond to the original message blocks $m_0, \ldots, m_{k-1}$.
At this point, we make an important observation. The elements $g_{j,i}$ in the second part of matrix G_{RREF} are elements of $\mathbb{F}_{2^m}$, while the elements in the message m are data blocks of $b \cdot w$ bits. Multiplication of a message block with a field element is not straightforward. We will not explain how such a multiplication is done, but refer to section 1B of (Hou & Han, 2016) for a description.

In the encoding step we recognize how the redundancy is added. The message m originally had length k, and is mapped to a codeword c of length n. The $n - k$ redundant blocks are added due to the length of the code.
It is important to realize that for this encoding step, the standard matrix G_{RREF} has to have the same number of rows as the number of blocks in the message m. This is the reason why we consider block coding. For block coding, an original message of length larger than k is divided up into multiple slices of length k (and one last slice that has less than k elements in case the message length is not divisible by k). Each block can be seen as a message $m = (m_0, \ldots, m_{k-1})$. Now the matrix multiplication can be done for each of the slices, encoding the original message per k message blocks at once. Each of the codewords is then sent over the communication channel (Yates, 2009).

We will now express some mathematical properties of an (n, k) code $\mathcal{C}$. This lies a foundation for a meaningful assessment of a code's quality. Let us start with a measure of efficiency of error-correcting codes, that compares the size of the original message m and the size of the generated codeword c (Clark & Cain, 1981).

Definition 4.1.6. (**Code rate**). Consider an (n, k) linear code $\mathcal{C}$. The code rate R of code $\mathcal{C}$ is

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

The next definition that we will give is that of the Singleton bound (Xambó-Descamps, 2003).

Definition 4.1.7. (**Singleton bound**). For an (n,k) code $\mathcal{C}$ with minimum distance d_{min}, we have

$$k + d_{min} \leq n + 1$$

When the equality in the Singleton bound is satisfied, so when $k + d_{min} = n + 1$, we say that code $\mathcal{C}$ is *maximum distance separable.*

Reed-Solomon codes have minimum distance $d_{min} = n - k + 1$ (Ravagnani, 2022), as do Cauchy Reed-Solomon codes (Blomer et al., 1999). In other words, CRS codes satisfy the equality of the Singleton bound. Thus, CRS codes are maximum distance separable. This is particularly interesting for the implementation of CRS codes for the MagiCtwin Diode. We will give more details on this in Section 5.2.1.

4.1.2 Decoding

When the received codeword arrives at the receiver, either block alterations or block erasures may have taken place. The original message now needs to be restored. The error-correcting code is applied again, this time in the channel decoder within the receiver.

There is not one general technique for decoding. There are different approaches for decoding received codewords containing *alterations* and for codewords containing *erasures* (Rao, 2019). We will focus on decoding erasures. In Section 5.2.2 it will become clear why we treat erasures only.

Suppose a codeword $c = (c_0, \dots, c_{n-1})$ was originally sent, but a codeword $c' = (c'_0, \dots, c'_{n-1})$, with $c'_i \in \{c_0, \dots, c_{n-1}, ?\}$ was received. Decoding can only be done if "enough" blocks are still intact. We will give an upper bound on the number of blocks that may have been altered or erased (Clark & Cain, 1981).

Proposition 1. (**Error-correction capability**). Let $\mathcal{C}$ be an (n,k) linear code with minimum distance d_{min}. Its error-correction capability t, the number of block *alterations* the code can correct, is bounded from above by the following upper bound:

$$t \leq \left\lfloor \frac{d_{min} - 1}{2} \right\rfloor$$

The number of block *erasures* that an error-correcting code can correct, the *erasure-correcting capability*, is $2t$ (Riley & Richardson, 2001).

A proof of error-correction capability t from Proposition 1 can be found in (Clark & Cain, 1981). Note that since the number of block erasures that a code can correct is twice as big as the number of block alterations it can correct, we get the following.

$$2t \leq 2 \left\lfloor \frac{d_{min} - 1}{2} \right\rfloor = d_{min} - 1 = n - k + 1 - 1 = n - k \tag{4.1}$$

So an (n,k) CRS code $\mathcal{C}$ with minimum distance d_{min} can correct up to $n - k$ block erasures.

In other words, decoding can only be done successfully when at most $n - k$ blocks in the received codeword c' are erased (i.e. are '?'). If more than $n - k$ blocks have been erased, there is more than one possible original codeword c (Didier, 2009), and the decoding process to determine the original codeword is not as straightforward. In this thesis, we will only elaborate on how to determine the original codeword when at most $n - k$ erasures have taken place.

Suppose we have received some codeword c' of which we know that M block erasures have occurred, assuming $M \leq n - k$. We know on what positions an erasure has occurred because the element on that position in the codeword equals '?'. We know how many erasures have occurred by counting the number of '?'s, which will result in a number for M.

The occurrence of erasures can be represented by a matrix multiplication in the following way. The received codeword c' with M block erasures equals the original message multiplied by some $k \times n$ matrix G' as shown below (Rao, 2019).

$$c' = mG' = (m_0, \dots, m_{k-1}) \begin{pmatrix} g'_{0,0} & \cdots & g'_{0,n-1} \\ \vdots & \ddots & \vdots \\ g'_{n-1,0} & \cdots & g'_{n-1,n-1} \end{pmatrix} = (c'_0, \dots, c'_{n-1})$$

Here, matrix G' corresponds to at least $n - M$ columns of matrix G, and decoding corresponds to finding the unique value of m such that $mG' = c'$.

4.2 Interleaving

Finally, we will describe an additional technique in error-correcting coding: interleaving. Interleaving is done to improve the error-correcting capability of a code, as we will see in Proposition 2.
Similar to error-correcting codes, we can distinguish *block* interleaving and *convolutional* interleaving. Since this thesis treats block codes, we will also focus on block interleaving.

A (block) interleaver is a device that rearranges the order of a sequence of input blocks at the transmitter in order to distribute errors at the receiver (Clark & Cain, 1981). This makes interleaving particularly useful for burst errors.
Interleaving can be done either periodically or randomly. In this thesis, we will only look at periodically interleaved codes. Periodically interleaved codes have an interleave factor s that determines in what way the sequence of input blocks is rearranged at the transmitter. A schematic overview of a message m going through an interleaver and deinterleaver with interleave factor s is included in Figure 4.1.

Interleaver

Input $\quad$ *write* $\longrightarrow$ $\quad$ Output

$$(m_0, m_1, \dots, m_{k-2}, m_{k-1}) \rightarrow \overset{\textit{read}}{\downarrow} \begin{pmatrix} m_0 & m_1 & \dots & m_{s-1} \\ m_s & m_{s-1} & \dots & m_{2s-1} \\ \vdots & \vdots & \ddots & \vdots \\ m_{\alpha s} & m_{\alpha s-1} & \dots & m_{k-1} \end{pmatrix} \rightarrow (m_0, m_s, \dots, m_{k-2}, m_{k-1})$$

Deinterleaver

Input $\quad$ *read* $\longrightarrow$ $\quad$ Output

$$(m_0, m_s, \dots, m_{k-2}, m_{k-1}) \rightarrow \overset{\textit{write}}{\downarrow} \begin{pmatrix} m_0 & m_1 & \dots & m_{s-1} \\ m_s & m_{s-1} & \dots & m_{2s-1} \\ \vdots & \vdots & \ddots & \vdots \\ m_{\alpha s} & m_{\alpha s-1} & \dots & m_{k-1} \end{pmatrix} \rightarrow (m_0, m_1, \dots, m_{k-2}, m_{k-1})$$

Figure 4.1: A schematic overview of an interleaver and deinterleaver with interleave factor s. The interleaver writes symbols $m_0, m_1, \dots, m_{k-1}$ into a matrix with s columns row by row. The interleaver rearranges the message symbols by reading them from the matrix column by column.
The deinterleaver performs the inverse operation: It writes message symbols $m_0, m_s, \dots, m_{k-1}$ into the same matrix format column by column, then rearranges the symbols by reading them out row by row.

The interleaving procedure can take place both before and after applying an error-correcting code $\mathcal{C}$ to a message m. Figure 4.1 shows the interleaving and deinterleaving procedure happening be-

fore applying error-correction coding. When interleaving happens before coding, a message m first goes through an interleaver in the transmitter, the interleaved message then gets encoded into an interleaved codeword. The interleaved codeword is transmitted over the channel. In the receiver, the interleaved codeword will first be decoded to an interleaved message by the error-correcting code. It then continues through a deinterleaver, outputting the original message.
When interleaving happens after encoding, the transmitter first applies the error-correcting code, converting a message m into a codeword c. The codeword c then goes through an interleaver and the interleaved codeword is sent over a communication channel. In the receiver, the interleaved codeword first passes through a deinterleaver, outputting the original codeword. Decoding the codeword into the original message is the final step.

We will mention the important property of a periodic interleaver with interleave factor s, namely its effect on the erasure-correcting capability of a code.

Proposition 2. (**Error-correcting capability of an interleaved code**). Suppose we have an (n, k) linear code $\mathcal{C}$ with error-correcting capability t. When applying a periodic interleaver with interleave factor s, we create an $(s \cdot n, s \cdot k)$ linear code with error-correcting capability $s \cdot t$.

A proof of Proposition 2 can be found in (Moon, 2005). A code with error-correcting capability st has erasure-correcting capability $2 \cdot st$. In other words, an (n, k) Cauchy Reed-Solomon code that is interleaved with factor s is able to restore $s \cdot 2t = s \cdot n - s \cdot k$ erasures.

An interleaved (n, k) Caucy Reed-Solomon code has the same code rate (see Proposition 4.1.6) and same 3.2.2. The advantage of interleaving lies purely in increasing the error-correcting capability, and therefore the erasure-correcting capability, of a code when it is applied on a communication channel where burst errors happen.

Chapter 5

Compumatica

Compumatica is a Dutch cybersecurity manufacturer based in Uden. They develop, produce and implement security solutions for network encryption, e-mail encryption, network security and network segmentation. The customers of Compumatica are governments and Top 500 companies. Their main customer industries are the military, government, transport & logistics, factories, and the electric power industry.

Many organizations separate operational technology (OT) and information technology (IT) (Piggin, 2014), Compumatica's customers included. Often, communication from OT to IT is wanted, but communication in the reverse direction is undesired. To enable secure one-way communication for its clients, Compumatica offers the MagiCtwin Diode: a device that connects two networks, and only allows one-way communication of data. A few examples of how Compumatica's clients implement the MagiCtwin Diode are the following. Suppose user data from operational systems need to be sent to an IT department. Then a communication in the direction from OT to IT must be facilitated. However, it is not necessary for an IT department to be able to remotely access OT systems. In fact, such returning communication creates a vulnerability, as this communication direction may also be targeted by malicious external parties.
Another way in which the MagiCtwin Diode can be used, is to protect back-ups from threats like ransomware from happening. With the MagiCtwin Diode implemented, ransomware cannot reach the back-ups and therefore cannot encrypt them, nor perform any other threatening actions.
A third example is enabling information classification. When different information classification levels are specified, the MagiCtwin Diode can be used to share information with a network that has a higher or lower classification level. Because of the one-way communication, information can even be filtered.
The final case we illustrate is the implementation of a MagiCtwin Diode in a system to protect underlying (either out-of-date or vulnerable) systems against threats from the outside, while still enabling an information stream from those systems to security operation centres.

It is important to realize that this does not mean that there are no possibilities of communication from receiver to sender whatsoever. In the first illustrated case for example, OT systems may still be accessed by a third party physically or through wireless connections like 4G or 5G. In the second use case, the data back-up will have to be made by someone, and has to be stored somewhere. These persons and locations may still be reached in other (physical) manners by malicious actors. For the third and fourth use case we can reason in a similar way that the MagiCtwin Diode can be bypassed in different ways.

Nevertheless, the use of a MagiCtwin Diode (or another device that enables one-way communication from sender to receiver for that matter) decreases the possibility that OT systems, back-ups, a certain information classification level, or underlying vulnerable systems can be targeted by third (malevolent) parties. The downside of this one-way communication is that the receiver cannot check with the sender whether errors have occurred during transmission. This is the reason that error-correcting coding has been deployed in the MagiCtwin Diode.

5.1 Information theory in the MagiCtwin Diode

We will now outline the MagiCtwin Diode in more details. We refer back to the introduction of communication theory in Chapter 3. The front of the MagiCtwin Diode can be seen in Figure 5.1. It consists of two independent compartments.

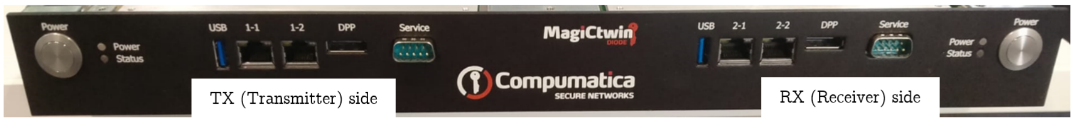


Figure 5.1: The front of the MagiCtwin Diode. On the left we have a power button and connection gates of the TX (transmitter) side. On the right we see the same for the RX (receiver) side.

A user network connects to one of the gates of the TX (transmitter) side, shown on the left of Figure 5.1. Another user network connects to one of the gates of the RX (receiver) side, shown on the right of Figure 5.1. The MagiCtwin Diode then only enables one-way communication of data, namely from left to right (so from transmitter to receiver). When the transmitting user network wants to send a data file, it has to upload the file to an FTP TX Daemon located on the TX side. This Daemon applies the error-correcting code as well. The coded data is then sent from TX to RX by means of the communication channel, and arrives at an FTP RX Daemon. This RX Daemon uses the error-correcting code to decode the received data. A receiving user network then connects to the RX Daemon using FTP to retrieve the data file.
The MagiCtwin Diode's hardware restricts the communication to only one way. Sending data from TX to RX uses an internal connection. The communication channel between them transfers data as light signals.

Referring back to the communication scheme in Figure 3.1, the **info source** is a user network that connects on the TX side of the MagiCtwin Diode, and the **destination** is a user network that connects on the RX side of it. All elements in the transmitter environment of Figure 3.1 are located on the left side, the TX side, of the MagiCtwin Diode, whereas all elements in the receiver environment are located on the RX side of the device. The communication channel is a light channel.

We will now express the information source and communication channel in the MagiCtwin Diode to our definitions from Sections 3.1 and 3.2.

Information source

First, we will elaborate on modelling the information source of the original data files that arrive at the TX Daemon. A data file is sliced into multiple slices, each slice can be seen as a message m. In the MagiCtwin Diode, one message m consists of $k = 128$ blocks, so $m = (m_0, \ldots, m_{127})$ with $m_i \in \mathcal{X}$. Each m_i consists of $b = 1456$ words of $w = 8$ bits. In other words, each block m_i consists of $b = 1456$ bytes. There are $2^8 = 256$ different bytes. The input alphabet $\mathcal{X}$ therefore contains 256^{1456} distinct elements. The output alphabet $\mathcal{Y}$ contains one more element and equals $\mathcal{Y} = \mathcal{X} \cup \{?\}$.

Communication channel

The MagiCtwin's communication channel can be denoted as $\mathcal{K}_{diode} = (\mathcal{X}, \mathcal{Y}, \mathbb{P}_{diode})$ where $\mathcal{X}$ and $\mathcal{Y}$ are defined as above. Due to imperfections of the channel, there are several probabilities that errors occur. $\mathbb{P}_{diode}$ defines a collection of probabilities that symbol (data block) alterations or erasures occur. The error-correcting code that is implemented in the MagiCtwin Diode is an erasure code, meaning block alterations cannot directly be corrected (Luby, Mitzenmacher, Shokrollahi, & Spielman, 2001). However, block alterations can be corrected in an indirect way. In order to handle them, erroneous data blocks are treated as erasures in the following way. A checksum that is implemented earlier in the communication process in the MagiCtwin Diode checks for block alterations. If a bit alteration is detected, the whole data block in which the alteration was detected will not be passed on to the channel encoder. The decoding part of the error-correcting application then registers this as a block erasure. In this way, alterations are treated as erasures.

To determine (approximate) values for the collection of probabilities $\mathbb{P}_{diode}$, experiments within the MagiCtwin Diode must be conducted. The probability strongly depends on external factors like bottlenecks on the Central Processing Unit (CPU), which makes it difficult to arrive at a definite collection of values. This is not possible within the scope of this thesis, so we will leave this for further research.

5.2 Coding theory in the MagiCtwin Diode

In 2016, one of Compumatica's employees implemented a Cauchy Reed-Solomon error-correcting code in the MagiCtwin Diode to detect and correct block erasures that occur during data transmission over the communication channel. The process of selecting this code and exact implementation was not well documented. Because of this, Compumatica is not able to substantiate the appropriateness and/or efficiency of the current RS code implementation. In other words, the current implementation might be subject to improvement, or there might be alternative error-correcting codes that are better suited for the MagiCtwin Diode. In this section, we will shed light on the limited information there is on the selection process, show how the code is implemented, and determine what properties it has.

5.2.1 Why a CRS code?

There is no documentation on why Compumatica decided to implement a CRS code, but we can reason why they are appropriate for the MagiCtwin Diode. First of all, (C)RS codes can correct single block erasures, but also burst erasures (Geisel, 2012). Burst erasures are erasures of multiple consecutive blocks. The reason that RS codes are particularly useful for burst errors is because they are maximum distance separable (van Lint, 1991). Within the MagiCtwin Diode, burst errors are a common type of error. They might occur when the Central Processing Unit (CPU) is overloaded, when the MagiCtwin Diode is overheated, or when some other hardware component fails. According to Compumatica, such hardware influences are the most probable reasons for errors. Therefore, it makes sense to choose an error-correcting code that is suitable for burst errors.

Secondly, the code rate (see Definition 4.1.6) of Reed-Solomon codes is relatively high compared to other error-correcting codes, making a Reed-Solomon code suitable for applications like data storage and data transmission (Tanwar, 2015). These are very important purposes of the MagiCtwin Diode, as explained at the beginning of Chapter 5.

Also, it was not documented why Compumatica chose to treat alterations in a block as erasures. However, erasures are easier to decode then alterations (Rao, 2019). It is plausible that this is the reason that alterations in a block are treated as full block erasures.

Additionally, there are mathematical advantages of (C)RS codes. Namely, that it operates over a relatively small finite field; "only" 256 elements (Yan, Sprintson, & Zelenko, 2014). This means that the mathematical operations are not so expensive in time and storage. Moreover, CRS codes ensure a large minimum distance d_{min} (Sklar, n.d.). This directly relates to the error-correcting capability of a code; the larger the minimum distance, the bigger the error-correcting capability (see Proposition 1).

The final advantage of (Cauchy) Reed-Solomon codes over other error-correcting codes is that CRS codes are among the codes that allow for interleaving (Clark & Cain, 1981), which increases the error-correcting capability (see Proposition 2). Periodic interleaving is implemented in the MagiCtwin Diode to be even more prepared for burst errors. The interleaver structure that is used has interleaving factor $s = 8$.
We consider l consecutive messages. Let m_i^j denote data block i of message j, with $i \in \{0, \ldots, 128\}$ and $j \in \{0, \ldots, l\}$. Figure 5.2 shows how the data blocks in l consecutive messages are interleaved and deinterleaved.

Interleaver

Input Output

$$(m_0^0, m_1^0, \dots, m_{126}^{l-1}, m_{127}^{l-1}) \to \begin{array}{cc} & \textit{write} \longrightarrow \\ \overset{\textit{read}}{\downarrow} & \begin{pmatrix} m_0^0 & m_1^0 & \dots & m_7^0 \\ m_8^0 & m_9^0 & \dots & m_{15}^0 \\ \vdots & \vdots & \ddots & \vdots \\ m_{120}^{l-1} & m_{121}^{l-1} & \dots & m_{127}^{l-1} \end{pmatrix} \end{array} \to (m_0^0, m_8^0, \dots, m_{119}^{l-1}, m_{127}^{l-1})$$

Deinterleaver

Input Output

$$(m_0^0, m_8^0, \dots, m_{119}^{l-1}, m_{127}^{l-1}) \to \begin{array}{cc} & \textit{read} \longrightarrow \\ \overset{\textit{write}}{\downarrow} & \begin{pmatrix} m_0^0 & m_1^0 & \dots & m_7^0 \\ m_8^0 & m_9^0 & \dots & m_{15}^0 \\ \vdots & \vdots & \ddots & \vdots \\ m_{120}^{l-1} & m_{121}^{l-1} & \dots & m_{127}^{l-1} \end{pmatrix} \end{array} \to (m_0^0, m_1^0, \dots, m_{126}^{l-1}, m_{127}^{l-1})$$

Figure 5.2: A schematic overview of the interleaver and deinterleaver with interleave factor 8 that is implemented in the MagiCtwin Diode. The interleaver writes data blocks $m_0^0, m_1^0, \dots, m_{126}^{l-1}, m_{127}^{l-1}$ into a matrix with 8 columns row by row. The interleaver rearranges the blocks by reading them from the matrix column by column.
The deinterleaver performs the inverse operation: It writes data blocks $m_0^0, m_8^0, \dots, m_{119}^{l-1}, m_{127}^{l-1}$ into the same matrix format, but now column by column. Then it rearranges the blocks by reading them out row by row.

As explained in Section 4.2, interleaving can happen either before or after encoding. In the MagiCtwin Diode, messages are interleaved before they are encoded (and deinterleaved after they are decoded).

5.2.2 Implementation of the CRS code

The currently used code is not developed by Compumatica employees, but is an open-source code called *Fast GF(256) Cauchy MDS Block Erasure Codec in C* (CM256 for short) that can be found on `https://github.com/catid/cm256`. It was slightly altered by Compumatica for GCC (Gnu Compiler Collection) and Linux support. Also, the interleaving protocol was specified by Compumatica. The exact error-correcting code as implemented in the MagiCtwin Diode cannot be included in this report.

CM256 library

We will briefly give an outline of the code in the CM256 library, which consists of several files. Five of them are relevant for the purposes of this thesis: `README.md, cm256.h, cm256.cpp, gf256.h` and `gf256.cpp`.
The `README.md` file provides some general details on the library, such as its parameters, its performance and an example usage. The `cm256.h` is the header of the `cm256.cpp` file. The header declares functions for encoding and decoding. The `cm256.cpp` contains the encoding and decoding steps. First, the library is initialized by performing some checks. Then, the functions for encoding a block, encoding a set of blocks, and decoding received blocks are defined. The structure of the decoder is defined, and finally a decoder function is defined. All mathematical computations during encoding and decoding are done in finite fields. The functions that describe how to do the mathematical computations at bit level are declared in the `gf256.h` header file. Lastly, the `gf256.cpp` file contains the function definitions that enable all mathematical computations on the finite field $\mathbb{F}_{256}$, which are specified for different CPU's. Almost all mathematical computations use lookup tables, which

are created using finite field arithmetic as explained in Chapter 2. Addition and subtraction do not require lookup tables, as they are performed by a XOR-operation.

In the MagiCtwin Diode's settings we find three standard settings, each with three parameters specified in the following form: `s×f/d`. Here `s` is the amount of stripes, which is relevant for the memory organization within a computer (*Main Memory Organization Computer Systems Structure*, n.d.) and is also the interleave factor s. Next, `f` denotes the number of redundant blocks generated by the code, it corresponds with the value $n-k$ in an (n,k) linear code. Finally, `d` is the number of blocks of original data, which corresponds to k in an (n,k) linear code. The possible settings are `8×16/128`, `8×16/64` and `8×8/64`. The setting `8x16/128` is used by default.

The C++ code in the CM256 library is specified by three parameters: `OriginalCount`, `RecoveryCount` and `BlockBytes`. Here, `OriginalCount` is the number of original data blocks, it corresponds to parameter k of a linear (n,k) code, and to the parameter `d` of CM256. Its default value is 128. This means that all data are sliced into slices with $k=128$ data blocks each. Next, `RecoveryCount` is the number of recovery blocks that are generated by the error-correcting code. It corresponds to $n-k$ in an (n,k) code, and to `f` in the CM256 library. Its standard value is 16, meaning there are $n-k=16$ recovery blocks generated per 128 original data blocks. We can conclude that $n=128+16=144$. Finally, `BlockBytes` is the number of bytes per block. This is the same for both the original data blocks and the recovery blocks: 1456 bytes per block. We conclude that an $(n,k)=(144,128)$ linear Cauchy Reed-Solomon error-correcting code is used in the MagiCtwin.
Note that this means that the original data that is encoded in every use of the error-correcting code is at most $128 \cdot 1456 = 186.368$ bytes (186 kB). When considering a simple text file, assuming one character is represented by one byte, this corresponds to roughly 90 A4 pages of text, or almost two low-resolution photographs (Sheldon, 2021).

Encoding

A data file sent over the MagiCtwin Diode is usually bigger than a 90 page text document or two low-resolution photographs. Therefore the data file must be sliced into slices of 128 data blocks each. Each slice can be denoted as a message $m=(m_0,\dots,m_{127})$ where each m_i is a data block from input alphabet $\mathcal{X}$ as defined at the start of Section 5.1.

Let $G_{\mathrm{RREF}}=(I_{128}\mid A)$ denote the standard generator matrix used in Compumatica's error-correcting code. Here I_{128} denotes the 128×128 identity matrix. Matrix A is based on Cauchy matrices. In a Cauchy matrix C, element c_{ij} on row i and column j is based on some x_i and y_j as follows.

$$c_{ij}=\frac{1}{x_i-y_j} \tag{5.1}$$

In the CM256 library, $x_i=i$, so each x_i corresponds to the row number in the matrix and to the index of the original data block i in message m. This means the values range from $0,\dots,k-1=0,\dots,127$. For y_j we have $y_j=j+128$, so the y_j corresponds to the column number in generator matrix G and to the index of the recovery block in the codeword. This means the values range from $k,\dots n-1=128,\dots,143$.
All matrix elements a_{ij} in A are divided by the element in the first column of row i, so divided by element $a_{i0}=\frac{1}{x_0-y_j}$, resulting in the following matrix elements.

$$a_{ij}=\frac{x_0-y_j}{x_i-y_j} \tag{5.2}$$

In the computation of matrix elements a_{ij}, the values $0,\dots,127$ and $128,\dots,143$ for x_i and y_j respectively are integer values between 0 and 255, so they are elements as in the first column of Table 2.1. The integers are put in their binary vector form to perform the subtraction by means of an XOR-operation in both the numerator and the denominator. The results of the numerator and denominator are put in their integer forms again, and division is done by using the *exp* and *log* tables to determine $exp(log(x_0-y_j)-log(x_i-y_j) \pmod{255})$ as explained in Chapter 2.

We will now continue with a step-by-step determination of the 128×16 matrix A that is used in $G_{\text{RREF}} = (I_{128} \mid A)$ in the CM256 library. First, we show in 5.3 how the matrix elements on row i and column j of matrix A are based on x_0, x_i and y_j.

$$\begin{pmatrix} \frac{x_0 - y_0}{x_0 - y_0} & \cdots & \frac{x_0 - y_{15}}{x_0 - y_{15}} \\ \vdots & \ddots & \vdots \\ \frac{x_0 - y_0}{x_{127} - y_0} & \cdots & \frac{x_0 - y_{15}}{x_{127} - y_{15}} \end{pmatrix} \tag{5.3}$$

Then, as shown in 5.4, the corresponding values $x_0 = 0$, $x_i = i$ and $y_j = j + 128$ in are filled in on each row i and column j.

$$\begin{pmatrix} \frac{0 - 128}{0 - 128} & \cdots & \frac{0 - 143}{0 - 143} \\ \vdots & \ddots & \vdots \\ \frac{0 - 128}{127 - 128} & \cdots & \frac{0 - 143}{127 - 143} \end{pmatrix} \tag{5.4}$$

The subtractions in the numerator and denominator correspond to XOR-operations on the binary vector forms of the integers, as shown below.

$$\begin{pmatrix} \frac{\text{XOR}((0,0,0,0,0,0,0,0),(1,0,0,0,0,0,0,0))}{\text{XOR}((0,0,0,0,0,0,0,0),(1,0,0,0,0,0,0,0))} & \cdots & \frac{\text{XOR}((0,0,0,0,0,0,0,0),(1,0,0,0,1,1,1,1))}{\text{XOR}((0,0,0,0,0,0,0,0),(1,0,0,0,1,1,1,1))} \\ \vdots & \ddots & \vdots \\ \frac{\text{XOR}((0,0,0,0,0,0,0,0),(1,0,0,0,0,0,0,0))}{\text{XOR}((0,1,1,1,1,1,1,1),(1,0,0,0,0,0,0,0))} & \cdots & \frac{\text{XOR}((0,0,0,0,0,0,0,0),(1,0,0,0,1,1,1,1))}{\text{XOR}((0,1,1,1,1,1,1,1),(1,0,0,0,1,1,1,1))} \end{pmatrix}$$

$$= \begin{pmatrix} \frac{(1,0,0,0,0,0,0,0)}{(1,0,0,0,0,0,0,0)} & \cdots & \frac{(1,0,0,0,1,1,1,1)}{(1,0,0,0,1,1,1,1)} \\ \vdots & \ddots & \vdots \\ \frac{(1,0,0,0,0,0,0,0)}{(1,1,1,1,1,1,1,1)} & \cdots & \frac{(1,0,0,0,1,1,1,1)}{(1,1,1,1,0,0,0,0)} \end{pmatrix} \tag{5.5}$$

Which in integer forms equal:

$$\begin{pmatrix} \frac{128}{128} & \cdots & \frac{143}{143} \\ \vdots & \ddots & \vdots \\ \frac{128}{255} & \cdots & \frac{143}{240} \end{pmatrix} \tag{5.6}$$

The divisions $\frac{a}{b}$ in 5.6 are done by using the *exp* and *log* tables to determine $exp(log(a){-}log(b) \pmod{255})$, which results in the following calculation.

$$\begin{pmatrix} exp(log(128) - log(128) \pmod{255}) & \cdots & exp(log(143) - log(143) \pmod{255}) \\ \vdots & \ddots & \vdots \\ exp(log(128) - log(255) \pmod{255}) & \cdots & exp(log(143) - log(240) \pmod{255}) \end{pmatrix}$$

$$= \begin{pmatrix} exp(7-7 \pmod{255}) & \dots & exp(252-252 \pmod{255}) \\ \vdots & \ddots & \vdots \\ exp(7-161 \pmod{255}) & \dots & exp(252-73 \pmod{255}) \end{pmatrix}$$

$$= \begin{pmatrix} exp(0) & \dots & exp(0) \\ \vdots & \ddots & \vdots \\ exp(101) & \dots & exp(179) \end{pmatrix} \tag{5.7}$$

Finally, this results in the 128×16 matrix A as shown below. The full matrix A is included in Appendix C.

$$A = \begin{pmatrix} 1 & \dots & 1 \\ \vdots & \ddots & \vdots \\ 133 & \dots & 222 \end{pmatrix} \tag{5.8}$$

The 128×144 standard generator matrix G_{RREF} is shown in 5.9.

$$G_{\text{RREF}} = (I_{128} \mid A) = \begin{pmatrix} 1 & \dots & 0 & 1 & \dots & 1 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \dots & 1 & 133 & \dots & 222 \end{pmatrix} \tag{5.9}$$

The data blocks in the codeword are generated by multiplying the blocks $m = (m_0, \dots, m_{k-1})$ by the standard generator matrix G_{RREF}. Since G is a 128×144 matrix, the result of this multiplication is a row vector of 144 elements: the codeword blocks $c = (c_0, \dots, c_{143})$.

$$c = m \cdot G = (m_0, \dots, m_{127}) \cdot \begin{pmatrix} 1 & \dots & 0 & 1 & \dots & 1 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \dots & 1 & 133 & \dots & 222 \end{pmatrix} = (c_0, \dots, c_{127}, c_{128}, \dots, c_{143})$$

Where

$$\begin{aligned} c_0 &= m_0 \\ &\vdots \\ c_{127} &= \qquad\qquad\qquad m_{127} \\ c_{128} &= m_0 + \dots + 133m_{127} \\ &\vdots \\ c_{143} &= m_0 + \dots + 222m_{127} \end{aligned}$$

We again refer to (Hou & Han, 2016) for details on the procedure of multiplying matrix elements with data blocks.

Suppose the data file was sliced into l slices. This encoding step is done for each of the slices of the original data file. The l codewords are sent over the diode with interleave factor $s = 8$.

Decoding

The channel decoder uses the error-correcting code to reconstruct the original message m from received codeword c'. Recall that at most $2t \leq n - k$ erasures can be restored, meaning that at most $n - k = 144 - 128 = 16$ blocks in c' may be '?'.

We will now explain how the reconstruction process of $M \leq 16$ erasures takes place. As explained in Section 4.1.2, the received codeword c' with M block erasures equals the original message multiplied by some $k \times n$ matrix G' in the following way.

$$c' = mG' = (m_0, \dots, m_{k-1}) \begin{pmatrix} g'_{0,0} & \cdots & g'_{0,n-1} \\ \vdots & \ddots & \vdots \\ g'_{n-1,0} & \cdots & g'_{n-1,n-1} \end{pmatrix} = (c'_0, \dots, c'_{n-1})$$

Here, matrix G' corresponds to at least $n - M$ columns of matrix G. Decoding corresponds to finding the unique value of m such that $mG' = c'$, so it is a matter of solving a system of linear equations.

In the decoding functions of CM256, two cases are distinguished: the case where $M = 1$ (only one bit erasure has taken place), and the case where $M > 1$ (more than one bit erasure has taken place). For $M = 1$, finding m is relatively easy. However, if $M > 1$, solving for m becomes more complicated. The CM256 library applied LDU-decomposition for finding m. The function for LDU-decomposition in CM256 uses Algorithm 2.5 (GS-direct-Cauchy) from (Boros, Kailath, & Olshevsky, 2002). We will not elaborate any further on the contents of this algorithm.

5.2.3 Properties of the CRS code

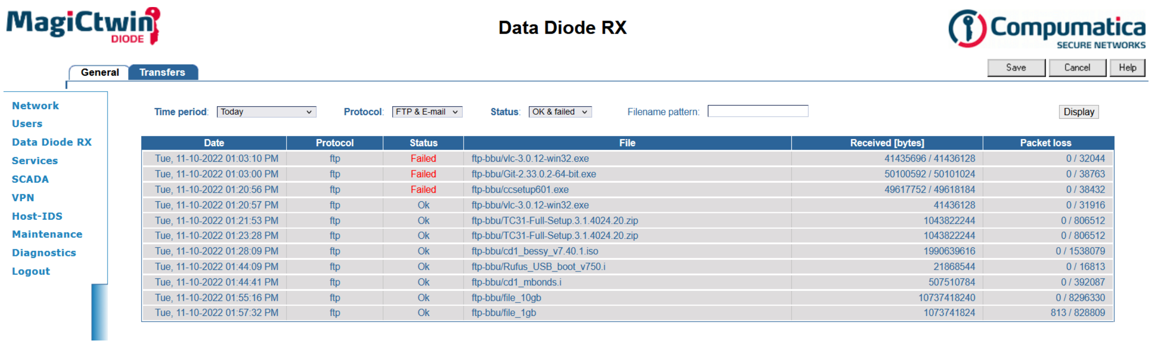
In Chapter 4, we treated some properties of error-correcting codes. We will now evaluate each of these properties for the error-correcting code that Compumatica has implemented.

Erasure-correcting capability

In Proposition 1 we stated that an error-correcting code could correct at most $2t = n - k$ block erasures. Since the MagiCtwin Diode uses an $(n, k) = (144, 128)$ linear Cauchy Reed-Solomon code, $n - k = 144 - 128 = 16$ block erasures can be restored solely by the CRS code. In case more than 16 block erasures have taken place, the error-correcting code will not restore the original codeword and corresponding message, but it will output to the receiver that the transmission was not successful.
If we also take into account the interleaving protocol that was implemented, we consider the Cauchy Reed-Solomon code as an $(s \cdot n, s \cdot k) = (8 \cdot 144, 8 \cdot 128) = (1152, 1024)$ linear code with error-restoring capability $s \cdot 2t = s \cdot (n - k) = 8 \cdot (144 - 128) = 128$. Therefore, 128 consecutive block erasures can be restored, provided that 1024 data blocks around it have been transmitted correctly.

Signal-to-noise ratio

In the MagiCtwin Diode, the signals are sent from the transmitter to the receiver in the form of light. This is one of the most reliable types of communication channels, but still bit errors occur (Alzubi et al., 2014). For an assessment of the code, it would be interesting to know how many errors occur on average in the MagiCtwin Diode in a customer environment. A small experiment in a test environment at Compumatica was done to investigate how many errors occur on average during data transmission through the MagiCtwin Diode.



Time period: Today | Protocol: FTP & E-mail | Status: OK & failed | Filename pattern:

Date	Protocol	Status	File	Received [bytes]	Packet loss
Tue, 11-10-2022 01:03:10 PM	ftp	Failed	ftp-bbu/vlc-3.0.12-win32.exe	41435696 / 41436128	0 / 32044
Tue, 11-10-2022 01:03:00 PM	ftp	Failed	ftp-bbu/Git-2.33.0.2-64-bit.exe	50100592 / 50101024	0 / 38763
Tue, 11-10-2022 01:20:56 PM	ftp	Failed	ftp-bbu/ccsetup601.exe	49617752 / 49618184	0 / 38432
Tue, 11-10-2022 01:20:57 PM	ftp	Ok	ftp-bbu/vlc-3.0.12-win32.exe	41436128	0 / 31916
Tue, 11-10-2022 01:21:53 PM	ftp	Ok	ftp-bbu/TC31-Full-Setup.3.1.4024.20.zip	1043822244	0 / 806512
Tue, 11-10-2022 01:23:28 PM	ftp	Ok	ftp-bbu/TC31-Full-Setup.3.1.4024.20.zip	1043822244	0 / 806512
Tue, 11-10-2022 01:28:09 PM	ftp	Ok	ftp-bbu/cd1_bessy_v7.40.1.iso	1990639616	0 / 1538079
Tue, 11-10-2022 01:44:09 PM	ftp	Ok	ftp-bbu/Rufus_USB_boot_v750.i	21868544	0 / 16813
Tue, 11-10-2022 01:44:41 PM	ftp	Ok	ftp-bbu/cd1_mbonds.i	507510784	0 / 392087
Tue, 11-10-2022 01:55:16 PM	ftp	Ok	ftp-bbu/file_10gb	10737418240	0 / 8296330
Tue, 11-10-2022 01:57:32 PM	ftp	Ok	ftp-bbu/file_1gb	1073741824	813 / 828809

Figure 5.3: A screenshot of the interface for the MagiCtwin data diode, after 11 test files were sent.

A user connection was established with the FTP server on the TX side and the FTP server on the RX side. Then, 8 tests were performed. In Figure 5.3, a screenshot of the diode's interface is shown. It shows 11 files sent by the user, of which the first 3 were not relevant for this small experiment.

The first three data files can be ignored for the purposes of evaluation the signal-to-noise ratio. The next 7 data files were uploaded to the FTP server on the TX side, using Filezilla. In Appendix E, a screenshot of the Filezilla interface is included in order to illustrate how this is done. All 7 data files arrived at the RX side completely, this is seen because the column 'Packet loss' indicates 0 lost packets for each of the 7 files. The last file transfer was manipulated by simulating a 0.1% packet loss, using Network Emulator (*Traffic Control Manual For Lab1*, n.d.).

We would like to know how many errors occur on average. The 8 experiments that were done do not provide a definite and usable number. Experimenting by simulation takes too much time, and designing a rigorous experiment to find out a definite signal-to-noise ratio is outside the scope of this thesis. A theoretically acceptable value of the signal-to-noise ratio in wireless communication is 15dB (Ab-Rahman, Shuhaimi, Al-Hakim Azizan, & Hassan, 2012). We will convert this value back to a ratio, using the following conversion formula from (Price & Goble, 1993).

$$SNR = 10 \cdot \log_{10}\left(\frac{S}{N}\right) \text{ [dB]} \tag{5.10}$$

Using this conversion formula, we calculate that a ratio of 15dB corresponds to the following signal-to-noise ratio $\frac{S}{N}$.

$$10 \cdot \log\left(\frac{S}{N}\right) = 15 \text{ dB} \iff \frac{S}{N} = 10^{0.1 \cdot \log(S/N)} = 10^{1.5} \approx 31.6$$

In other words, for a codeword of $S = n = 144$ blocks that is sent over the data diode, we can expect $\lceil N \rceil \approx \lceil \frac{144}{31.6} \rceil \approx \lceil 4.6 \rceil = 5$ blocks to be erased.

Code rate

In Definition 4.1.6 we gave a formula for a code's rate. Let us look at the code rate of the error-correcting code that Compumatica currently uses. We use the default parameters $k = 128$ and $n = 144$ to compute the code rate: $R = \frac{k}{n} = \frac{128}{144} \approx 0.89$.

Chapter 6

Improving Compumatica's code

We now know how the CM256 library is implemented for error-correcting purposes in the MagiCtwin Diode. We also know its relevant properties. This lays a foundation for discussing several options to improve it. There are several paths to improvements at this point. First of all, the CM256 library could be improved in terms of its efficiency. Secondly, a different default setting might improve the MagiCtwin's efficiency while still having a large enough erasure-correcting capability. Finally, another class of error-correcting codes could be implemented. We will discuss all three options.

6.1 Improving the CM256 library

When looking at the CM256 library, different improvements come to mind. First of all, the matrix-vector multiplication in the function `cm256_encode_block` (`cm256.cpp` lines `166-190`) can be done faster than it is being done now. One exemplary way in which it can be optimized is by implementing a Fast-Fourier-Transform algorithm (Rao, 2019).
The developer of the CM256 library has also compared CM256 with two different Cauchy Reed-Solomon code libraries, namely *Intel(R) Intelligent Storage Acceleration Library* (ISA-L for short, code can be accessed via `https://github.com/intel/isa-l`) and *Longhair* (code can be accessed via `https://github.com/catid/longhair`.). The ISA-L library is more efficient in this matrix multiplication, so another option to optimize the matrix-vector multiplication in Compumatica's implementation of the error-correcting code is to be inspired by the approach taken in ISA-L.

During decoding, the LDU-decomposition that is used to solve $mG' = c'$ for m is a way of performing Gaussian elimination. Gaussian elimination will always work, but it is slow compared to other solving techniques. Gaussian elimination has complexity $\mathcal{O}(n^3)$ (Boros et al., 2002). A faster decoding technique can be implemented via a Berlekamp-Massey decoding algorithm, which has complexity $\mathcal{O}(kn)$ (Shum, 2016).

Compumatica currently limits the transmission speed of data through the diode at 400 megabits per second (Mbits/sec), corresponding to 50 megabytes per second (Mbytes/sec), while the total bandwidth of the diode is 1000 Mbits/sec (125 Mbytes/sec). This is done for several reasons. First of all, in this way they make sure that the amount of data transmitted per time unit stays limited in case of hardware failures. Secondly, hardware limitations are taken into account in this manner. For instance, the CPU at the RX side must be able to decode the worst case scenario faster than the TX side is able to encode and send new information. The speed of the CPU forms such a hardware limitation.
It makes sense to wonder whether the transmission speed limit can be relieved now that we know that we can restore over three times as much errors than we expect to happen, or at least increased. However, this remaining bandwidth of 600 Mbits/sec (75 Mbytes/sec) is also necessary for sending other data and protocols. Compumatica does not know at this moment how much bandwidth these remaining data and protocols take up exactly, therefore we cannot study to what limit the transmission speed may be increased during this thesis.

Finally, if Compumatica would want to improve the CM256 library in terms of functions it has, they might want to look into the possibility of incorporating encryption. This is a relevant function when developing a MagiCtwin Diode that offers customers to implement the TX and RX side in two physically different locations. When looking for encryption possibilities, the ISA-L library could be of interest. It is a Cauchy Reed-Solomon erasure code, meaning it is the same type of code as the one in the CM256 library, but also allows for cryptological functions. The ISA-L library that contains crypto functions can be accessed via `https://github.com/intel/isa-l_crypto`).

6.2 Comparing different settings

Currently, Compumatica uses the same default setting `8x16/128` for all file sizes that are to be transmitted from the TX to RX side. It is plausible that one of the other two standard settings generates less but still sufficient redundant data, therefore being more efficient.

Assuming the 3.5% of erasures from (Ab-Rahman et al., 2012), we can significantly reduce the number of generated recovery blocks for messages of 128 blocks. When looking at one sequence of 8 codewords again, we expect $\lceil 0.035 \cdot 8n \rceil$ data blocks to be erased and know that we can restore at most $s \cdot 2t = 8 \cdot (n-k)$ erased data blocks, meaning we need $8(n-k)$ recovery blocks. Taking $k = 128$, we solve $0.035 \cdot 8n = 8(n-k)$ and get $\lceil 8n \rceil = 133$, meaning that $n - k = 133 - 128 = 5$. In other words, only 5 recovery blocks have to be generated for a total of 8 messages of 128 blocks each. This corresponds to $\lceil \frac{5}{8} \rceil = 1$ recovery block per message, so a total codeword length of $128 + 1 = 129$. It seems like a totally new setting would be more efficient than either one of the three current settings. We suggest to up to 8 recovery blocks per 128 data blocks in order to agree with the interleaving protocol with interleave factor $s = 8$. Therefore, creating a new setting `8x8/128` seems sufficient.

Let us assess the three standard settings and the newly suggested setting by calculating their code rates, expected number of block erasures and error-correcting capabilities with and without interleaving. The comparison is shown in Table 6.1.

MagiCtwin Diode setting		**`8x16/128`**	**`8x16/64`**	**`8x8/64`**	**`8x8/128`**
Corresponding parameters	(n,k)	$(144,128)$	$(80,64)$	$(72,64)$	$(136,128)$
Code rate R	$\frac{k}{n}$	$\frac{8}{9}$	$\frac{4}{5}$	$\frac{8}{9}$	$\frac{16}{17}$
Fraction converted to percentage		89%	80%	89%	94%
Expected erasures (blocks)	$\lceil 0.035 \cdot n \rceil$	6	3	3	5
Percentage of n		4.2%	3.8%	4.2%	3.7%
Erasure-correcting capability $2t$ (blocks)	$n-k$	16	16	8	8
Percentage of n		11.1 %	20%	11.1%	5.9%
Erasure-correcting capability $2st$ with interleaving factor 8 (blocks)	$8(n-k)$	128	128	64	64
Percentage of 8·n		11.1 %	20%	11.1%	5.9%

Table 6.1: A comparison of the code rate, expected block erasures and erasure-correcting capability with and without periodic interleaving for four possible settings of the MagiCtwin Diode.

In Table 6.1, the `8x16/128` setting that Compumatica currently uses is denoted by the gray column. The `8x8/128` that we suggest in this thesis is shown in the cyan column. We see that `8x8/128` has a higher code rate than the three settings the MagiCtwin Diode can now adapt. Also, we see that the erasure-correcting capability percentage of the new setting is much closer compared to the expected number of block erasures than in the other three settings, meaning it uses available bandwidth more efficiently. We consider the new setting to be more efficient since it generates less, but still enough, redundant data.

An interesting question that Compumatica posed is whether the efficiency of transmitting different file sizes might profit from different standard settings. Since the average file that is sent over the diode in the MagiCtwin Diode is significantly larger than 186 kB, the setting that is used to apply the error-correcting code does not influence the efficiency any differently for different file sizes. For any file size, new setting `8x8/128` is most efficient.

6.3 Alternative suitable codes

We have explained in Section 5.2.1 why we presume a Cauchy Reed-Solomon error-correcting code was chosen. We will now present some suitable alternatives. Any of the alternatives require further investigation in order to implement them successfully in the MagiCtwin Diode, but they provide Compumatica with a starting point.

A suitable MDS burst erasure correcting code can be a low-density parity-check (LDPC) code, for example by implementing Construction 5 or Construction 6 proposed in (Johnson, 2009). LDPC codes also allow for interleaving (Sridharan, Kumarasubramanian, Thangaraj, & Bhashyam, 2008), and are therefore compatible with the interleaver in the MagiCtwin Diode as well.
Another class of error-correcting codes that could replace the Cauchy Reed-Solomon code are Fire codes (Fire, 1959). They are very efficient for burst errors (Moon, 2005). A good starting point might be Construction 1 as proposed in (Zhou, Lin, & Abdel-Ghaffar, 2014).
The final error-correcting code that we propose that is suited for the purposes in the MagiCtwin Diode is a Turbo code. Turbo codes have good erasure correcting capabilities and have good performance when code rates are high (Mizuochi, 2006). Construction algorithms for Turbo codes are given in (Biradar & Sasi, 2018).

Chapter 7

Discussion and future research

We started this thesis by giving a structured description of information theory and a mathematical explanation of coding theory. We have introduced (Cauchy) Reed-Solomon codes and we have shown how information theory and CRS codes are implemented by MagiCtwin Diode. We have pointed out why choosing a CRS code is appropriate for the purposes and implementation of error-correcting coding in the MagiCtwin Diode. We then proposed different improvements regarding efficiency of the CM256 library, and we suggested a fourth standard setting that has advantages over the current three standard settings. Finally, we put forward three classes of alternative error-correcting codes that might be suitable for the MagiCtwin Diode.
There are several notes that we have to make before we give Compumatica a definite recommendation regarding the efficiency of the error-correcting code that is currently implemented in the MagiCtwin Diode.

In this thesis we have considered symbol alterations and symbol erasures. In practice, symbol additions are also possible (Bours, 1994). Whether this can happen in the diode is not investigated in this thesis. Also, we are not aware whether this is implemented in the CM256 library.

Definition 3.2.1 gives a mathematical definition of a communication channel, which is expressed for the MagiCtwin Diode in Section 5.1. It assumes a *memoryless* communication channel, which means that what happens in one channel use is independent from what happens in the other channel uses (Ravagnani, 2022). Whether the diode in the MagiCtwin Diode is a memoryless communication channel is not known. However, since most of the errors that happen are burst errors, so not random errors, the resulting behaviour of the communication system of the MagiCtwin Diode does not resemble a memoryless communication channel (Clark & Cain, 1981).
LDPC codes are well-suited for burst erasure channels, but specifically for memoryless burst erasure channels (Johnson, 2009). Some research into LDPC codes for burst erasure channels with memory has been done, but it is has not reached a level of implementation yet (Paolini & Chiani, 2006).

We have explained that due to the average size of customer files, the diode has to be used multiple times when transmitting a data file over the data diode. This makes it relevant to consider the communication channel as a non-memoryless channel. This affects the collection of probabilities $\mathbb{P}_{diode}$. Besides, the number of 3.5% expected erasures is based on random errors and a memoryless channel. Since the communication channel does not behave as a memoryless channel and the errors do not happen randomly but in burst, this 3.5% is not an accurate estimate.

Another issue with the percentage is that it only gives an estimate for errors that have occurred in the communication channel. In practice, errors might also happen during conversion from message to codeword, from codeword to light signal, from received signal back to received codeword, and from received codeword back to original message. Error-correcting coding does not cover these types of errors, so they are not included in this thesis. However, in the MagiCtwin Diode these errors might actually happen, so we feel obligated that this aspect must be brought to light here as well.

The suitability of a (Cauchy) Reed-Solomon error-correcting code for the MagiCtwin Diode is backed up by literature (Geisel, 2012; van Lint, 1991; Tanwar, 2015; Rao, 2019; Yan et al., 2014; Sklar, n.d.; Clark & Cain, 1981) as explained in Section 5.2.1. Each of the three alternative classes of error-correcting codes that seem suitable for the MagiCtwin Diode is supported by scientific papers as well (Sridharan et al., 2008; Zhou et al., 2014; Mizuochi, 2006). However, the alternatives have been insufficiently researched to be able to say that they will perfectly suit the MagiCtwin Diode.

While we believe that the results of this thesis are useful for Compumatica, future research should be conducted, starting with an investigation into the collection of probabilities $\mathbb{P}_{diode}$ that express the probabilities of data block alterations and erasures. With this collection of probabilities, we would have an accurate estimate of the expected number of alterations and erasures that we can expect to happen when a codeword c is transmitted over the diode in the MagiCtwin Diode. $\mathbb{P}_{diode}f$ would be significantly more appropriate to use than the 3.5% from (Ab-Rahman et al., 2012), and would give a useful value for the signal-to-noise ratio of the diode. Further research at the customer side of the MagiCtwin Diode can be done to learn how much time the MagiCtwin Diode is out of running on average, so how many consecutive block erasures actually happen. In case this is significantly less than 3.7 ms every 29.8 ms, changes can be made.
The combination of the collection of probabilities and the duration of failure of the MagiCtwin Diode should be used as a benchmark for the required erasure-correction capability of an error-correcting code.

Furthermore, one aspect of efficiency that we did not focus on in this paper is the delay that happens during the decoding of codewords with burst erasures. Particularly when applying block interleaving in combination with RS codes, we are faced with a sub-optimal decoding delay (Li, Khisti, & Girod, 2011). Several linear code constructions that achieve the Singleton bound as well as the lowest possible decoding delay are presented in (Li et al., 2011). These constructions are theoretical and therefore not yet ready implementation-wise. We suggest further research into these code constructions in order to see if one is suited for and applicable in the MagiCtwin Diode.

The Fast-Fourier-Transform algorithm is one way to accelerate matrix multiplication during encoding, but many more fast multiplication techniques exist. We suggest further research into fast (matrix) multiplication techniques for elements in finite fields. The same goes for the suggested Berlekamp-Massey algorithm that is a fast decoding technique for Cauchy Reed-Solomon codes.

Some explanations, like how to multiply a data block with a finite field element and the exact definition of a Reed-Solomon code, are not given in this thesis. It would be beneficial for the mathematical understanding of the implementation of error-correction coding in the MagiCtwin Diode to understand those details, but doing so does not make a difference on our recommendations for Compumatica. Given the limited scope of this thesis, it was therefore not included in this thesis.

Chapter 8

Recommendation

We have proposed different options to improve the efficiency of error-correcting coding in the MagiCtwin Diode in Chapter 6 and critically discussed the findings and limitations of this thesis in Chapter 7. We now have all tools to formulate a definite recommendation regarding the efficiency of error-correcting coding in the MagiCtwin Diode.

We will first advise on the approach to improve the currently implemented CM256 library. Regardless of other improvements, a faster multiplication algorithm for the vector-matrix multiplication during encoding should be implemented. For this, Compumatica may want to look at either the Fast-Fourier-Transform algorithm from (Rao, 2019), the ISA-L library from `https://github.com/intel/isa-l` or start a new research into fast multiplication. Similarly, a faster algorithm for decoding can be implemented. Compumatica may choose to look at a Berlekamp-Massey algorithm as in (Boros et al., 2002) or may choose to explore other decoding methods.

Besides this, one of the following two improvements can be implemented. One advise would be to create a new (fourth) standard setting that is `8x8/128`, so that generates 8 recovery blocks for each slice of 128 data blocks, keeping interleave factor 8. Possibly, there can be a fifth new setting with an even higher code rate, depending on further research determining the collection of probabilities $\mathbb{P}_{diode}$.

The last recommendation regarding the currently implemented code regards cryptological functions of the MagiCtwin Diode. If the possibility of incorporating encryption into the MagiCtwin Diode is of interest as well, the ISA-L library could be an example of how to do so.

Finally, Compumatica may decide to implement a different class of codes instead of Cauchy Reed-Solomon, rather than improve its currently implemented code. In this case, we advise Compumatica to do more (documented) research into the suitability and implementation methods of their preferred alternative class of codes. We suggest starting by looking at an LDPC code inspired by Construction 5 or 6 from (Johnson, 2009) or a Fire code similar to Construction 1 from (Zhou et al., 2014). Alternatively, Compumatica can look into construction methods for a Turbo code (Biradar & Sasi, 2018) or Li's code (Li et al., 2011).

References

Ab-Rahman, M. S., Shuhaimi, N. I., Al-Hakim Azizan, L., & Hassan, M. R. (2012). Analytical Study of Signal-to-Noise Ratio for Visible Light Communication by Using Single Source. *Journal of Computer Science*, *8*(1), 141–144. doi: 10.3844/jcssp.2012.141.144

Alencar, M. (2014). *Information theory* (1st ed.). Momentum Press.

Alzubi, J. A., Alzubi, O. A., & Chen, T. M. (2014). *Forward Error Correction Based on Algebraic-Geometric Theory.* Springer, Cham. Retrieved from `https://link.springer.com/chapter/10.1007/978-3-319-08293-6_1` doi: 10.1007/978-3-319-08293-6{_}1

Biradar, S., & Sasi, M. S. (2018, 6). Error Detection and Correction using Turbo Codes. *International Research Journal of Engineering and Technology*, *5*(6). Retrieved from `https://www.irjet.net/archives/V5/i6/IRJET-V5I6503.pdf`

Blomer, J., Kalfane, M., Karp, R., Karpinski, M., Luby, M., & Zuckerman, D. (1999, 10). An XOR-Based Erasure-Resilient Coding Scheme. Retrieved from `https://www.researchgate.net/publication/2643899_An_XOR-Based_Erasure-Resilient_Coding_Scheme/citation/download`

Bocklandt, R. R. J. (n.d.). *Error correcting codes 1 Motivation* (Tech. Rep.). UvA/FNWI. Retrieved from `https://staff.fnwi.uva.nl/r.r.j.bocklandt/notes/CT.pdf`

Boros, T., Kailath, T., & Olshevsky, V. (2002, 3). Pivoting and backward stability of fast algorithms for solving Cauchy linear equations. *Linear Algebra and Its Applications*, *343*, 63–99. doi: 10.1016/S0024-3795(01)00519-5

Bours, P. A. H. (1994). *Codes for correcting insertion and deletion errors* (Unpublished doctoral dissertation). Technische Universiteit Eindhoven.

Clark, G. C., & Cain, J. B. (1981). *Error-Correction Coding for Digital Communications* (Lucky R. W., Ed.). Melbourne, Florida: Springer US. Retrieved from `https://link.springer.com/content/pdf/10.1007/978-1-4899-2174-1.pdf` doi: 10.1007/978-1-4899-2174-1

Didier, F. (2009, 1). Efficient erasure decoding of Reed-Solomon codes. Retrieved from `https://arxiv.org/pdf/0901.1886.pdf`

Edwards, H. M. (1984). *Galois Theory* (3rd ed.). New York: New York: Springer-Verlag.

Fire, P. (1959). *A class of multiple-error-correcting binary codes for non-independent errors* (Unpublished doctoral dissertation). Stanford University.

Geisel, W. A. (2012, 11). Tutorial on Reed-Solomon Error Correction Coding. *NASA technical memorandum*, *102162*. Retrieved from `https://ntrs.nasa.gov/api/citations/19900019023/downloads/19900019023.pdf`

Hou, H., & Han, Y. S. (2016, 12). Cauchy MDS Array Codes With Efficient Decoding Method. *CoRR*, *abs/1611.09968*. Retrieved from `https://arxiv.org/pdf/1611.09968.pdf`

Johnson, S. (2009, 3). Burst erasure correcting LDPC codes. *IEEE Transactions on Communications*, *57*(3), 641–652. doi: 10.1109/TCOMM.2009.03.060468

Kahn, D. (1996). *The Codebreakers.* Scribner.

Lagarias, J. C. (1993, 2). Pseudorandom Numbers. *Statistical Science*, *8*(1), 31–39. doi: 10.1214/ss/1177011081

Li, Z., Khisti, A., & Girod, B. (2011). Correcting erasure bursts with minimum decoding delay. In *2011 conference record of the forty fifth asilomar conference on signals, systems and computers (asilomar)* (pp. 33–39). IEEE. Retrieved from `https://www.comm.utoronto.ca/~akhisti/burst_erasure.pdf` doi: 10.1109/ACSSC.2011.6189949

Luby, M., Mitzenmacher, M., Shokrollahi, M., & Spielman, D. (2001, 2). Efficient erasure correcting codes. *IEEE Transactions on Information Theory*, *47*(2), 569–584. Retrieved from https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=910575 doi: 10.1109/18.910575

Mackay, D. J. C. (1995). Information Theory, Inference, and Learning Algorithms. Retrieved from http://www.inference.phy.cam.ac.uk/mackay/itila/

Main Memory Organization Computer Systems Structure. (n.d.). University of Tennessee, Knoxville. Department of Electrical Engineering and Computer Science. Retrieved from http://web.eecs.utk.edu/~mbeck/classes/cs160/lectures/13_memory.pdf

Maurer, H. (2021). *Cognitive Science* (1st ed.). Boca Raton: CRC Press. doi: 10.1201/9781351043526

Mizuochi, T. (2006, 8). Recent progress in forward error correction and its interplay with transmission impairments. *IEEE Journal of Selected Topics in Quantum Electronics*, *12*(4), 544–554. Retrieved from https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1668095 doi: 10.1109/JSTQE.2006.876597

Moon, T. K. (2005). *Error Correcting Coding Mathematical Methods and Algorithms.* Hoboken, New Jersey: John Wiley & Sonds, Inc. Retrieved from https://doc.lagout.org/science/0_Computer%20Science/2_Algorithms/Error%20Correction%20Coding_%20Mathematical%20Methods%20and%20Algorithms%20%5BMoon%202005-06-06%5D.pdf

Moorhouse, E. (2008, 10). *Reed-Solomon Codes.* University of Wyoming. Retrieved from https://ericmoorhouse.org/handouts/reed_solomon.pdf

Paolini, E., & Chiani, M. (2006). Improved low-density parity-check codes for burst erasure channels. *IEEE International Conference on Communications*, *3*, 1183–1188. Retrieved from https://ieeexplore.ieee.org/document/4024300 doi: 10.1109/ICC.2006.254908

Piggin, R. (2014). Industrial systems: cyber-security's new battlefront [Information Technology Operational Technology]. *Engineering & Technology*, *9*(8), 70–74. doi: 10.1049/et.2014.0810

Plank, J. S. (2005, 12). Optimizing Cauchy Reed-Solomon Codes for Fault-Tolerant Storage Applications. Retrieved from http://www.cs.utk.edu/œplank/plank/papers/CS-05-659.pdf

Planteen, C. (2019, 7). *Reed-Solomon Codes.* Retrieved from https://codyplanteen.com/notes/rs

Price, J., & Goble, T. (1993). *Telecommunications Engineer's Reference Book* (F. Mazda, Ed.). Elsevier. doi: 10.1016/C2013-0-06529-2

Rao, A. (2019, 10). Lecture 4: Reed-Solomon Codes. Retrieved from https://homes.cs.washington.edu/~anuprao/pubs/codingtheory/lecture4.pdf

Ravagnani, A. (2022, 9). *Coding Theory.* Eindhoven University of Technology. Retrieved from https://a.ravagnani.win.tue.nl/coding%20th/Coding_theory_notes.pdf

Reed, I. S., & Solomon, G. (1960, 6). POLYNOMIAL CODES OVER CERTAIN FINITE FIELDS. *J Soc. INDUST. AL. MATH*, *8*(2). Retrieved from http://www.siam.org/journals/ojsa.php

Riley, M., & Richardson, I. (2001). *An introduction to Reed Solomon codes: principles, architecture & implementation.* Retrieved from https://www.cs.cmu.edu/~guyb/realworld/reedsolomon/reed_solomon_codes.html

Shannon, C. E. (1948). *A Mathematical Theory of Communication* (Vol. 27; Tech. Rep.).

Sheldon, R. (2021, 12). *How many bytes for . . . ?* Retrieved from https://www.techtarget.com/searchstorage/definition/How-many-bytes-for

Shum, K. (2016, 11). *Berlekamp-Massey decoding of RS code.* Retrieved from https://piazza.com/class_profile/get_resource/isgy6spmwwm3ba/iwg4az9vjjz3em

Sklar, B. (n.d.). Reed-Solomon Codes. Retrieved from https://ptgmedia.pearsoncmg.com/images/art_sklar7_reed-solomon/elementlinks/art_sklar7_reed-solomon.pdf

Sridharan, G., Kumarasubramanian, A., Thangaraj, A., & Bhashyam, S. (2008, 7). Optimizing burst erasure correction of LDPC codes by interleaving. In *2008 ieee international symposium on information theory* (pp. 1143–1147). IEEE. Retrieved from https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4595166 doi: 10.1109/ISIT.2008.4595166

Tanwar, S. (2015, 5). *Reed Soloman and convolution codes.* Retrieved from https://www.slideshare.net/shaileshtanwar/reedsoloman-and-convolution-c

Taylor, C. A. (2021). *catid/cm256: Fast GF(256) Cauchy MDS Block Erasure Codec in C.* Retrieved from https://github.com/catid/cm256

Traffic Control Manual For Lab1. (n.d.). Retrieved from https://www.cs.unm.edu/~crandall/netsfall13/TCtutorial.pdf van Lint, J. H. (1991). *Introduction to Coding Theory* (2nd ed.). Eindhoven: Springer-Verlag.

Xambó-Descamps, S. (2003). *Block Error-Correcting Codes.* Berlin, Heidelberg: Springer Berlin Heidelberg. doi: 10.1007/978-3-642-18997-5

Yan, M., Sprintson, A., & Zelenko, I. (2014). *Weakly Secure Data Exchange with Generalized Reed Solomon Codes* (Tech. Rep.). Department of Mathematics, Texas A&M University. Retrieved from `https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6875056`

Yates, R. (2009, 8). *A Coding Theory Tutorial.* Retrieved from `https://web.archive.org/web/20110710143034/http://www.digitalsignallabs.com/tutorial.pdf`

Zhou, W., Lin, S., & Abdel-Ghaffar, K. (2014). Burst or random error correction based on fire and BCH codes. *2014 Information Theory and Applications Workshop, ITA 2014 - Conference Proceedings*. doi: 10.1109/ITA.2014.6804214

Appendix A

Generator polynomials for $\mathbb{F}_{256}$

Below, we present a table that shows all 16 primitive, irreducible polynomials of $\mathbb{F}_{256}$. The polynomials are often characterized by a hexadecimal, which is included for each polynomial as well. The highlighted row is the polynomial that is used in the CM256 code.

Hexadecimal	**Generator polynomial**
`0x8e`	$F_1(x) = x^8 + x^4 + x^3 + x^2 + 1$
`0x95`	$F_2(x) = x^8 + x^5 + x^3 + x + 1$
`0x96`	$F_3(x) = x^8 + x^5 + x^3 + x^2 + 1$
`0xa6`	$F_4(x) = x^8 + x^6 + x^3 + x^2 + 1$
`0xaf`	$F_5(x) = x^8 + x^6 + x^4 + x^3 + x^2 + x + 1$
`0xb1`	$F_6(x) = x^8 + x^6 + x^5 + x + 1$
`0xb2`	$F_7(x) = x^8 + x^6 + x^5 + x^2 + 1$
`0xb4`	$F_8(x) = x^8 + x^6 + x^5 + x^3 + 1$
`0xb8`	$F_9(x) = x^8 + x^6 + x^5 + x^4 + 1$
`0xc3`	$F_{10}(x) = x^8 + x^7 + x + 2 + x + 1$
`0xc6`	$F_{11}(x) = x^8 + x^7 + x^3 + x^2 + 1$
`0xd4`	$F_{12}(x) = x^8 + x^7 + x^5 + x^3 + 1$
`0xe1`	$F_{13}(x) = x^8 + x^7 + x^6 + x + 1$
`0xe7`	$F_{14}(x) = x^8 + x^7 + x^6 + x^3 + x^2 + x + 1$
`0xf3`	$F_{15}(x) = x^8 + x^7 + x^6 + x^5 + x^2 + x + 1$
`0xfa`	$F_{16}(x) = x^8 + x^7 + x^6 + x^5 + x^4 + x^2 + 1$

Appendix B

Exp and log tables for $\mathbb{F}_{256}$

The *exp* table for finite field $\mathbb{F}_{256}$ based on generator polynomial $F_4(x) = x^8 + x^6 + x^3 + x^2 + 1$ is shown below.

i	$exp(i)$	i	$exp(i)$	i	$exp(i)$	i	$exp(i)$	i	$exp(i)$	i	$exp(i)$	i	$exp(i)$	i	$exp(i)$
0	1	32	227	64	235	96	93	128	171	160	217	192	181	224	158
1	2	33	139	65	155	97	186	129	27	161	255	193	39	225	113
2	4	34	91	66	123	98	57	130	54	162	179	194	78	226	226
3	8	35	182	67	246	99	114	131	108	163	43	195	156	227	137
4	16	36	33	68	161	100	228	132	216	164	86	196	117	228	95
5	32	37	66	69	15	101	133	133	253	165	172	197	234	229	190
6	64	38	132	70	30	102	71	134	183	166	21	198	153	230	49
7	128	39	69	71	60	103	142	135	35	167	42	199	127	231	98
8	77	40	138	72	120	104	81	136	70	168	84	200	254	232	196
9	154	41	89	73	240	105	162	137	140	169	168	201	177	233	197
10	121	42	178	74	173	106	9	138	85	170	29	202	47	234	199
11	242	43	41	75	23	107	18	139	170	171	58	203	94	235	195
12	169	44	82	76	46	108	36	140	25	172	116	204	188	236	203
13	31	45	164	77	92	109	72	141	50	173	232	205	53	237	219
14	62	46	5	78	184	110	144	142	100	174	157	206	106	238	251
15	124	47	10	79	61	111	109	143	200	175	119	207	212	239	187
16	248	48	20	80	122	112	218	144	221	176	238	208	229	240	59
17	189	49	40	81	244	113	249	145	247	177	145	209	135	241	118
18	55	50	80	82	165	114	191	146	163	178	111	210	67	242	236
19	110	51	160	83	7	115	51	147	11	179	222	211	134	243	149
20	220	52	13	84	14	116	102	148	22	180	241	212	65	244	103
21	245	53	26	85	28	117	204	149	44	181	175	213	130	245	206
22	167	54	52	86	56	118	213	150	88	182	19	214	73	246	209
23	3	55	104	87	112	119	231	151	176	183	38	215	146	247	239
24	6	56	208	88	224	120	131	152	45	184	76	216	105	248	147
25	12	57	237	89	141	121	75	153	90	185	152	217	210	249	107
26	24	58	151	90	87	122	150	154	180	186	125	218	233	250	214
27	48	59	99	91	174	123	97	155	37	187	250	219	159	251	225
28	96	60	198	92	17	124	194	156	74	188	185	220	115	252	143
29	192	61	193	93	34	125	201	157	148	189	63	221	230	253	83
30	205	62	207	94	68	126	223	158	101	190	126	222	129	254	166
31	215	63	211	95	136	127	243	159	202	191	252	223	79	255	*

The *log* table for $\mathbb{F}_{256}$ based on generator polynomial $F_4(x) = x^8 + x^6 + x^3 + x^2 + 1$ is shown below.

j	$log(j)$	j	$log(j)$	j	$log(j)$	j	$log(j)$	j	$log(j)$	j	$log(j)$	j	$log(j)$	j	$log(j)$
0	*	32	5	64	6	96	28	128	7	160	51	192	29	224	88
1	0	33	93	65	212	97	123	129	222	161	68	193	61	225	251
2	1	34	135	66	37	98	231	130	213	162	105	194	124	226	226
3	23	35	108	67	210	99	59	131	120	163	146	195	235	227	32
4	2	36	155	68	94	100	142	132	38	164	45	196	232	228	100
5	46	37	183	69	39	101	158	133	101	165	82	197	233	229	208
6	24	38	193	70	136	102	116	134	211	166	254	198	60	230	221
7	83	39	49	71	102	103	244	135	209	167	22	199	234	231	119
8	3	40	43	72	109	104	55	136	95	168	169	200	143	232	173
9	106	41	167	73	214	105	216	137	227	169	12	201	125	233	218
10	47	42	163	74	156	106	206	138	40	170	139	202	159	234	197
11	147	43	149	75	121	107	249	139	33	171	128	203	236	235	64
12	25	44	152	76	184	108	131	140	137	172	165	204	117	236	242
13	52	45	76	77	8	109	111	141	89	173	74	205	30	237	57
14	84	46	202	78	194	110	19	142	103	174	91	206	245	238	176
15	69	47	27	79	223	111	178	143	252	175	181	207	62	239	247
16	4	48	230	80	50	112	87	144	110	176	151	208	56	240	73
17	92	49	141	81	104	113	225	145	177	177	201	209	246	241	180
18	107	50	115	82	44	114	99	146	215	178	42	210	217	242	11
19	182	51	54	83	253	115	220	147	248	179	162	211	63	243	127
20	48	52	205	84	168	116	172	148	157	180	154	212	207	244	81
21	166	53	130	85	138	117	196	149	243	181	192	213	118	245	21
22	148	54	18	86	164	118	241	150	122	182	35	214	250	246	67
23	75	55	86	87	90	119	175	151	58	183	134	215	31	247	145
24	26	56	98	88	150	120	72	152	185	184	78	216	132	248	16
25	140	57	171	89	41	121	10	153	198	185	188	217	160	249	113
26	53	58	240	90	153	122	80	154	9	186	97	218	112	250	187
27	129	59	71	91	34	123	66	155	65	187	239	219	237	251	238
28	85	60	79	92	77	124	15	156	195	188	204	220	20	252	191
29	170	61	14	93	96	125	186	157	174	189	17	221	144	253	133
30	70	62	189	94	203	126	190	158	224	190	229	222	179	254	200
31	13	63	6	95	228	127	199	159	219	191	114	223	126	255	161

The second columns of the *exp* and *log* tables were calculated using the following Wolfram Mathematica code.

```
In[1]:= list = {{0, 1}};
        For[i = 1, i ≤ 254, i++, AppendTo[list, {i, If[list⟦-1, 2⟧ ≤ 127, 2 list⟦-1, 2⟧, BitXor[2 list⟦-1, 2⟧ - 256, 77]]}]];
        exp = list⟦All, 2⟧;
        log = SortBy[list, Last]⟦All, 1⟧;
```

Appendix C

Matrix A for CM256

Matrix A in generator matrix $G = (I_{128} \mid A)$ that is used in the CM256 library is included below. The elements are integers between 0 and 255, as in the first column of Table 2.1 from Chapter 2.

$$\begin{pmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
138 & 146 & 34 & 179 & 181 & 211 & 4 & 83 & 97 & 216 & 128 & 147 & 20 & 212 & 9 & 44 \\
40 & 71 & 106 & 90 & 165 & 11 & 232 & 36 & 104 & 78 & 254 & 193 & 91 & 17 & 230 & 43 \\
100 & 154 & 209 & 249 & 14 & 247 & 144 & 58 & 207 & 250 & 161 & 39 & 25 & 118 & 62 & 51 \\
158 & 75 & 4 & 21 & 215 & 183 & 83 & 141 & 130 & 85 & 181 & 33 & 178 & 204 & 211 & 159 \\
255 & 76 & 16 & 86 & 60 & 68 & 174 & 225 & 64 & 87 & 41 & 152 & 172 & 107 & 30 & 65 \\
160 & 31 & 119 & 110 & 122 & 203 & 188 & 236 & 239 & 49 & 101 & 127 & 186 & 208 & 77 & 12 \\
26 & 242 & 218 & 165 & 232 & 200 & 103 & 47 & 57 & 194 & 106 & 176 & 81 & 40 & 108 & 148 \\
42 & 214 & 232 & 112 & 74 & 169 & 36 & 65 & 224 & 32 & 165 & 84 & 114 & 149 & 11 & 41 \\
182 & 243 & 241 & 122 & 188 & 159 & 73 & 9 & 171 & 115 & 119 & 23 & 33 & 160 & 44 & 89 \\
129 & 63 & 213 & 22 & 126 & 81 & 173 & 131 & 140 & 18 & 139 & 123 & 175 & 35 & 155 & 176 \\
190 & 19 & 118 & 12 & 89 & 83 & 150 & 120 & 49 & 62 & 240 & 24 & 38 & 253 & 4 & 73 \\
201 & 253 & 144 & 97 & 153 & 27 & 58 & 238 & 237 & 223 & 14 & 61 & 54 & 150 & 247 & 216 \\
232 & 28 & 105 & 189 & 123 & 64 & 111 & 168 & 107 & 63 & 56 & 92 & 29 & 165 & 251 & 69 \\
202 & 113 & 46 & 215 & 83 & 161 & 102 & 219 & 170 & 55 & 4 & 250 & 222 & 158 & 93 & 205 \\
121 & 231 & 194 & 251 & 32 & 193 & 187 & 191 & 50 & 248 & 78 & 214 & 189 & 108 & 70 & 206 \\
45 & 164 & 83 & 243 & 70 & 249 & 141 & 176 & 172 & 192 & 215 & 79 & 234 & 29 & 183 & 139 \\
28 & 218 & 115 & 217 & 80 & 127 & 90 & 112 & 45 & 178 & 49 & 6 & 233 & 44 & 170 & 96 \\
10 & 190 & 142 & 153 & 58 & 41 & 52 & 230 & 56 & 179 & 144 & 87 & 84 & 201 & 43 & 184 \\
62 & 128 & 33 & 121 & 195 & 199 & 79 & 13 & 205 & 233 & 186 & 142 & 232 & 131 & 76 & 237 \\
162 & 199 & 174 & 207 & 38 & 222 & 225 & 105 & 37 & 26 & 60 & 15 & 212 & 163 & 68 & 250 \\
110 & 155 & 37 & 83 & 102 & 209 & 233 & 107 & 238 & 227 & 46 & 235 & 78 & 202 & 132 & 149 \\
25 & 86 & 208 & 216 & 184 & 36 & 133 & 227 & 223 & 77 & 228 & 191 & 171 & 6 & 232 & 52 \\
188 & 228 & 113 & 233 & 164 & 50 & 91 & 114 & 108 & 8 & 75 & 34 & 120 & 122 & 217 & 246 \\
92 & 6 & 188 & 104 & 57 & 19 & 236 & 206 & 59 & 80 & 122 & 138 & 195 & 133 & 203 & 78 \\
235 & 66 & 22 & 109 & 231 & 255 & 239 & 59 & 232 & 204 & 10 & 240 & 44 & 250 & 164 & 54 \\
83 & 148 & 96 & 145 & 15 & 140 & 248 & 182 & 35 & 199 & 152 & 109 & 124 & 215 & 177 & 154 \\
234 & 130 & 124 & 126 & 173 & 216 & 66 & 62 & 71 & 169 & 213 & 111 & 61 & 129 & 51 & 136 \\
246 & 20 & 103 & 74 & 36 & 101 & 47 & 221 & 39 & 187 & 232 & 49 & 48 & 42 & 200 & 16
\end{pmatrix}$$

$$\vdots$$

$$\vdots$$

$$\begin{pmatrix}
224 & 48 & 107 & 176 & 136 & 58 & 163 & 254 & 18 & 30 & 219 & 21 & 146 & 199 & 144 & 66 \\
132 & 8 & 55 & 177 & 192 & 33 & 24 & 242 & 95 & 168 & 85 & 164 & 145 & 93 & 147 & 23 \\
41 & 83 & 69 & 241 & 95 & 17 & 31 & 6 & 66 & 162 & 13 & 108 & 221 & 27 & 175 & 100 \\
134 & 180 & 36 & 190 & 147 & 110 & 65 & 250 & 175 & 146 & 74 & 217 & 137 & 124 & 169 & 60 \\
78 & 58 & 246 & 124 & 134 & 208 & 63 & 67 & 7 & 151 & 42 & 44 & 211 & 222 & 145 & 255 \\
148 & 46 & 179 & 13 & 69 & 61 & 21 & 243 & 134 & 114 & 223 & 245 & 5 & 43 & 39 & 111 \\
102 & 204 & 247 & 15 & 248 & 71 & 27 & 86 & 129 & 224 & 96 & 231 & 134 & 83 & 192 & 112 \\
210 & 25 & 73 & 57 & 236 & 139 & 9 & 211 & 152 & 90 & 188 & 18 & 79 & 92 & 159 & 194 \\
241 & 145 & 252 & 75 & 113 & 80 & 155 & 81 & 245 & 207 & 191 & 234 & 201 & 119 & 49 & 249 \\
77 & 254 & 84 & 132 & 185 & 67 & 217 & 50 & 16 & 5 & 54 & 73 & 83 & 105 & 63 & 59 \\
59 & 79 & 108 & 23 & 11 & 225 & 200 & 101 & 168 & 70 & 230 & 97 & 154 & 53 & 174 & 7 \\
34 & 67 & 173 & 239 & 171 & 48 & 131 & 96 & 82 & 202 & 126 & 100 & 17 & 128 & 81 & 49 \\
214 & 26 & 28 & 203 & 159 & 252 & 205 & 213 & 48 & 238 & 44 & 38 & 104 & 82 & 244 & 83 \\
97 & 68 & 82 & 36 & 47 & 119 & 5 & 35 & 206 & 244 & 103 & 115 & 85 & 246 & 151 & 29 \\
183 & 7 & 177 & 106 & 218 & 121 & 140 & 82 & 188 & 150 & 137 & 219 & 51 & 87 & 245 & 212 \\
186 & 22 & 150 & 78 & 194 & 141 & 120 & 244 & 80 & 247 & 89 & 242 & 56 & 245 & 83 & 9 \\
185 & 237 & 227 & 192 & 24 & 250 & 7 & 30 & 31 & 19 & 55 & 91 & 15 & 132 & 65 & 11 \\
58 & 89 & 20 & 5 & 180 & 95 & 178 & 234 & 93 & 53 & 214 & 106 & 227 & 153 & 13 & 76 \\
244 & 108 & 170 & 67 & 172 & 210 & 127 & 138 & 240 & 95 & 130 & 149 & 209 & 187 & 182 & 177 \\
109 & 245 & 58 & 130 & 170 & 86 & 238 & 23 & 189 & 69 & 153 & 40 & 185 & 120 & 27 & 85 \\
193 & 60 & 160 & 184 & 133 & 47 & 92 & 157 & 179 & 44 & 208 & 252 & 131 & 25 & 103 & 3 \\
115 & 198 & 207 & 157 & 8 & 129 & 237 & 189 & 83 & 149 & 210 & 228 & 43 & 49 & 180 & 195 \\
228 & 241 & 169 & 158 & 202 & 123 & 110 & 104 & 109 & 54 & 18 & 67 & 3 & 51 & 56 & 20 \\
36 & 205 & 111 & 118 & 100 & 37 & 168 & 10 & 163 & 67 & 123 & 157 & 99 & 74 & 64 & 31 \\
30 & 161 & 61 & 28 & 17 & 53 & 40 & 42 & 184 & 3 & 175 & 99 & 188 & 242 & 231 & 37 \\
137 & 224 & 99 & 38 & 225 & 78 & 198 & 77 & 75 & 249 & 174 & 248 & 138 & 162 & 12 & 235 \\
171 & 250 & 245 & 230 & 108 & 99 & 121 & 151 & 203 & 83 & 3 & 14 & 90 & 239 & 29 & 157 \\
76 & 91 & 102 & 70 & 141 & 14 & 219 & 136 & 127 & 24 & 83 & 223 & 251 & 45 & 161 & 213 \\
173 & 221 & 190 & 3 & 245 & 45 & 186 & 195 & 44 & 46 & 253 & 209 & 254 & 126 & 158 & 210 \\
172 & 251 & 35 & 250 & 235 & 236 & 128 & 181 & 26 & 155 & 221 & 112 & 71 & 67 & 188 & 198 \\
3 & 201 & 12 & 178 & 222 & 243 & 78 & 32 & 185 & 152 & 68 & 140 & 8 & 52 & 21 & 57 \\
151 & 53 & 187 & 64 & 146 & 84 & 191 & 113 & 117 & 182 & 32 & 180 & 118 & 200 & 193 & 87 \\
206 & 55 & 114 & 163 & 162 & 3 & 137 & 218 & 101 & 240 & 255 & 83 & 252 & 39 & 117 & 67 \\
139 & 36 & 154 & 142 & 117 & 204 & 253 & 245 & 198 & 34 & 50 & 93 & 136 & 19 & 212 & 160 \\
22 & 147 & 117 & 58 & 52 & 16 & 3 & 108 & 105 & 181 & 142 & 183 & 49 & 10 & 148 & 133 \\
197 & 255 & 141 & 25 & 193 & 97 & 176 & 49 & 212 & 71 & 70 & 13 & 220 & 99 & 249 & 126 \\
177 & 39 & 197 & 225 & 198 & 194 & 196 & 44 & 113 & 14 & 99 & 27 & 18 & 137 & 89 & 128 \\
85 & 236 & 76 & 99 & 197 & 150 & 199 & 98 & 252 & 209 & 45 & 43 & 11 & 48 & 118 & 129 \\
48 & 57 & 45 & 174 & 99 & 89 & 197 & 196 & 191 & 161 & 29 & 247 & 23 & 114 & 240 & 35 \\
205 & 103 & 90 & 50 & 154 & 138 & 112 & 190 & 197 & 234 & 80 & 226 & 107 & 159 & 127 & 248 \\
192 & 206 & 199 & 198 & 196 & 120 & 98 & 28 & 155 & 144 & 197 & 41 & 169 & 177 & 150 & 34 \\
47 & 149 & 203 & 100 & 168 & 75 & 19 & 22 & 162 & 172 & 111 & 8 & 197 & 36 & 146 & 243 \\
96 & 179 & 195 & 200 & 151 & 196 & 88 & 241 & 139 & 219 & 121 & 58 & 74 & 152 & 197 & 72
\end{pmatrix}$$

$$\vdots$$

$$\vdots$$

$$\begin{pmatrix}
231 & 186 & 52 & 170 & 238 & 60 & 230 & 11 & 123 & 21 & 58 & 26 & 217 & 109 & 41 & 55 \\
140 & 61 & 196 & 105 & 77 & 187 & 44 & 159 & 91 & 153 & 198 & 86 & 202 & 218 & 194 & 181 \\
142 & 118 & 30 & 214 & 20 & 69 & 68 & 222 & 226 & 239 & 242 & 137 & 92 & 144 & 223 & 110 \\
24 & 211 & 224 & 196 & 98 & 109 & 175 & 17 & 51 & 142 & 199 & 16 & 101 & 192 & 201 & 220 \\
247 & 181 & 79 & 151 & 88 & 98 & 13 & 95 & 213 & 107 & 195 & 52 & 36 & 96 & 199 & 189 \\
196 & 162 & 176 & 54 & 84 & 130 & 49 & 80 & 204 & 214 & 193 & 95 & 46 & 198 & 97 & 171 \\
189 & 217 & 93 & 87 & 183 & 131 & 161 & 14 & 182 & 147 & 211 & 104 & 31 & 72 & 173 & 252 \\
254 & 196 & 38 & 8 & 237 & 234 & 56 & 123 & 102 & 42 & 207 & 208 & 148 & 115 & 178 & 79 \\
106 & 98 & 225 & 237 & 56 & 251 & 105 & 111 & 233 & 246 & 38 & 160 & 204 & 254 & 222 & 223 \\
248 & 21 & 88 & 101 & 119 & 44 & 241 & 73 & 126 & 136 & 151 & 238 & 147 & 15 & 196 & 240 \\
164 & 202 & 148 & 27 & 41 & 30 & 16 & 174 & 251 & 206 & 43 & 171 & 130 & 233 & 7 & 36 \\
127 & 32 & 129 & 235 & 128 & 9 & 34 & 4 & 249 & 51 & 35 & 74 & 242 & 172 & 73 & 196 \\
104 & 81 & 233 & 141 & 219 & 144 & 107 & 163 & 23 & 7 & 102 & 179 & 32 & 76 & 209 & 124 \\
221 & 124 & 19 & 201 & 10 & 164 & 22 & 239 & 106 & 212 & 168 & 72 & 196 & 65 & 75 & 25 \\
169 & 252 & 64 & 4 & 46 & 132 & 37 & 233 & 58 & 133 & 220 & 221 & 12 & 18 & 226 & 17 \\
98 & 137 & 136 & 84 & 49 & 170 & 115 & 90 & 158 & 20 & 176 & 31 & 37 & 196 & 153 & 131 \\
54 & 207 & 133 & 85 & 55 & 65 & 227 & 7 & 69 & 203 & 184 & 113 & 152 & 226 & 36 & 230 \\
71 & 87 & 98 & 77 & 44 & 244 & 28 & 205 & 81 & 58 & 196 & 60 & 110 & 140 & 120 & 4 \\
88 & 59 & 244 & 146 & 191 & 49 & 252 & 155 & 253 & 86 & 187 & 178 & 100 & 151 & 176 & 183 \\
12 & 144 & 26 & 149 & 42 & 145 & 246 & 63 & 227 & 121 & 40 & 196 & 9 & 68 & 59 & 164 \\
236 & 184 & 91 & 107 & 255 & 117 & 114 & 137 & 200 & 72 & 164 & 4 & 244 & 57 & 50 & 63 \\
213 & 47 & 112 & 117 & 253 & 158 & 190 & 186 & 196 & 220 & 154 & 132 & 163 & 139 & 138 & 182 \\
7 & 93 & 39 & 98 & 175 & 231 & 61 & 40 & 228 & 117 & 224 & 29 & 119 & 24 & 10 & 64 \\
70 & 41 & 15 & 228 & 208 & 103 & 160 & 92 & 235 & 196 & 145 & 244 & 173 & 243 & 218 & 117 \\
157 & 226 & 236 & 224 & 39 & 22 & 206 & 87 & 145 & 154 & 57 & 158 & 88 & 227 & 19 & 32 \\
242 & 183 & 175 & 44 & 28 & 2 & 17 & 149 & 216 & 52 & 98 & 174 & 122 & 71 & 109 & 46 \\
33 & 126 & 201 & 194 & 120 & 219 & 109 & 2 & 90 & 43 & 150 & 30 & 105 & 186 & 102 & 156 \\
49 & 225 & 210 & 227 & 157 & 35 & 8 & 72 & 215 & 205 & 92 & 51 & 247 & 84 & 5 & 121 \\
179 & 62 & 239 & 2 & 53 & 162 & 59 & 145 & 36 & 29 & 231 & 89 & 159 & 223 & 255 & 185 \\
52 & 150 & 68 & 180 & 178 & 31 & 222 & 251 & 132 & 59 & 20 & 218 & 157 & 58 & 69 & 104 \\
89 & 142 & 249 & 42 & 246 & 15 & 97 & 130 & 157 & 195 & 26 & 98 & 156 & 12 & 152 & 91 \\
200 & 2 & 32 & 220 & 64 & 54 & 146 & 75 & 142 & 160 & 251 & 5 & 240 & 11 & 25 & 61 \\
141 & 16 & 248 & 208 & 160 & 82 & 182 & 210 & 128 & 98 & 15 & 2 & 66 & 70 & 140 & 253 \\
82 & 40 & 44 & 111 & 203 & 191 & 159 & 139 & 114 & 170 & 77 & 207 & 76 & 103 & 187 & 215 \\
155 & 101 & 138 & 148 & 204 & 72 & 158 & 45 & 194 & 156 & 212 & 66 & 58 & 113 & 8 & 82 \\
153 & 12 & 214 & 47 & 5 & 241 & 180 & 129 & 211 & 2 & 82 & 254 & 55 & 97 & 88 & 45 \\
220 & 172 & 66 & 171 & 131 & 85 & 62 & 247 & 214 & 110 & 173 & 168 & 40 & 34 & 216 & 115 \\
53 & 195 & 230 & 127 & 23 & 38 & 11 & 169 & 100 & 243 & 238 & 246 & 50 & 2 & 60 & 24 \\
56 & 49 & 226 & 211 & 93 & 66 & 132 & 209 & 27 & 36 & 156 & 122 & 21 & 237 & 124 & 2 \\
124 & 5 & 74 & 253 & 190 & 202 & 147 & 33 & 98 & 64 & 112 & 185 & 162 & 213 & 18 & 86 \\
63 & 178 & 47 & 147 & 65 & 122 & 221 & 235 & 61 & 191 & 36 & 80 & 177 & 134 & 101 & 174 \\
165 & 44 & 56 & 72 & 189 & 220 & 123 & 100 & 219 & 134 & 237 & 133 & 16 & 90 & 234 & 13
\end{pmatrix}$$

$$\vdots$$

$$\vdots$$

$$\begin{pmatrix}
225 & 136 & 25 & 156 & 226 & 134 & 54 & 185 & 43 & 103 & 6 & 119 & 181 & 38 & 42 & 231 \\
35 & 134 & 139 & 10 & 22 & 91 & 126 & 171 & 218 & 138 & 19 & 189 & 98 & 221 & 113 & 193 \\
175 & 177 & 163 & 49 & 115 & 238 & 254 & 165 & 202 & 68 & 136 & 243 & 75 & 98 & 58 & 62 \\
118 & 50 & 161 & 26 & 249 & 96 & 14 & 153 & 210 & 33 & 183 & 224 & 6 & 240 & 131 & 155 \\
156 & 92 & 216 & 114 & 48 & 190 & 85 & 192 & 88 & 123 & 81 & 37 & 53 & 9 & 112 & 170 \\
204 & 37 & 181 & 69 & 21 & 87 & 215 & 70 & 63 & 48 & 179 & 186 & 180 & 148 & 206 & 203 \\
209 & 72 & 24 & 140 & 71 & 79 & 242 & 20 & 167 & 10 & 192 & 255 & 208 & 161 & 33 & 18 \\
44 & 106 & 49 & 185 & 217 & 172 & 80 & 154 & 29 & 180 & 84 & 167 & 102 & 77 & 130 & 152 \\
23 & 187 & 234 & 128 & 34 & 156 & 220 & 46 & 14 & 228 & 129 & 36 & 30 & 127 & 167 & 98 \\
6 & 182 & 228 & 81 & 216 & 74 & 184 & 55 & 79 & 105 & 51 & 146 & 239 & 167 & 165 & 58 \\
60 & 141 & 31 & 73 & 167 & 149 & 6 & 226 & 62 & 106 & 95 & 200 & 235 & 86 & 17 & 201 \\
27 & 215 & 13 & 119 & 241 & 28 & 95 & 167 & 173 & 163 & 88 & 230 & 65 & 248 & 98 & 118 \\
207 & 193 & 167 & 236 & 9 & 213 & 156 & 93 & 96 & 165 & 73 & 169 & 223 & 210 & 205 & 120 \\
133 & 167 & 122 & 76 & 104 & 168 & 57 & 39 & 53 & 217 & 110 & 212 & 121 & 184 & 111 & 222
\end{pmatrix}$$

Above matrix A is calculated by the following Mathematica code.

```
In[12]:= list = {{0, 1}};
         For[i = 1, i ≤ 254, i++, AppendTo[list, {i, If[list⟦-1, 2⟧ ≤ 127, 2 list⟦-1, 2⟧, BitXor[2 list⟦-1, 2⟧ - 256, 77]]}]];
         exp = list⟦All, 2⟧;
         log = SortBy[list, Last] ⟦All, 1⟧;

In[16]:= div[a_, b_] := exp⟦Mod[log⟦a⟧ - log⟦b⟧, 255] + 1⟧

In[17]:= For[i = 0, i ≤ 127, i++, x[i] = i]
         For[j = 0, j ≤ 15, j++, y[j] = 128 + j]

In[21]:= A = Array[div[BitXor[x[0], y[#2 - 1]], BitXor[x[#1 - 1], y[#2 - 1]]] &, {128, 16}]
         alog = Map[ log⟦#⟧ &, A, {2}]
```

Appendix D

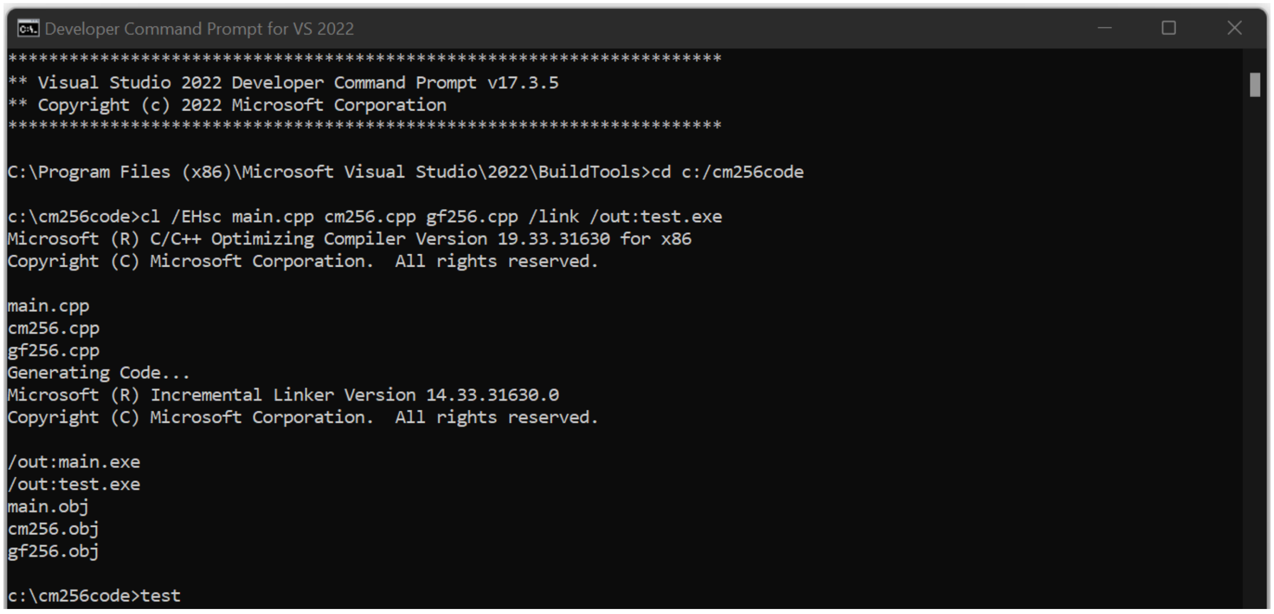
Example usage of CM256 library in C++

The error-correcting code that is used is from `https://github.com/catid/cm256`. The C++ code below is the example usage code from Github, containing adjustments that allow us to gain insight into statuses of objects during running. Full code is not included to avoid unnecessarily lengthy appendices.

```
#include "cm256.h"
#include <iostream>
#include <iomanip>
#include <bitset>
#include <string>
#include <typeinfo>
#include <cstring>
#include <climits>

using namespace std;

bool main()
{
    if (cm256_init())
    {
        cout << "error while initializing cm256 library" << endl;
    exit(1);
    }

    cm256_encoder_params params;

    int blockbytes = 5;
    params.BlockBytes    = 5;   // Number of bytes per file block
    params.OriginalCount = 6;   // Number of blocks
    params.RecoveryCount = 3;   // Number of additional recovery blocks generated by encoder

    // Size of the original file
    static const int OriginalFileBytes = params.OriginalCount * params.BlockBytes;

    // Allocate and fill the original file data
    uint8_t* originalFileData = new uint8_t[OriginalFileBytes];
  memset(originalFileData, 'a', OriginalFileBytes-1);
  memset(originalFileData, 'z', 8);
  memset(originalFileData, 'b', 4);
  memset(originalFileData, 'c', 2);

    // print originalFileData
    cout << "originalFileData = " <<originalFileData<< " (" <<OriginalFileBytes<< " bytes)\n" << endl;
    cout << "Index  Memory loc  Bits                                          Ascii" << endl;
    cout << "-----  ----------  --------------------------------------------  -----" << endl;
  for (int i = 0; i < OriginalFileBytes; ++i)
    {
        if (i % 5 == 0)
        {
            cout << setfill('0') << setw(5) << i;
            \\ printing memory (pointer) location
            cout << "  0x" << static_cast<void *>(&originalFileData[i]) << "  ";
        }
        if (i % 5 == 4)
            cout << " " << originalFileData[i - 4] << originalFileData[i - 3] << originalFileData[i - 2]
            << originalFileData[i - 1] << originalFileData[i] << endl;
  }

    // print encoder params
    cout << "\nThe encoder's parameters are: \n  " << params.BlockBytes << " bytes per block\n  " <<
    params.OriginalCount << " original blocks of data\n  " << params.RecoveryCount <<
    " additionally generated recovery blocks.\n" << endl;

    // create blocks
    cm256_block blocks[256];
    cout << "blocks[256] is created: " << endl;
    for (int i = 0; i < params.OriginalCount; ++i)
        blocks[i].Block = originalFileData + i * params.BlockBytes;

    // print blocks
    cout << "i  Pointer     Value " << endl;
    cout << "-  ----------  ----- " << endl;
    for (int i = 0; i < params.OriginalCount; ++i)
    {
        cout << i << "  0x" << blocks[i].Block << "  ";
        for (int j = 0; j < params.BlockBytes; ++j)
            cout << reinterpret_cast<char*>(blocks[i].Block)[j];
        cout << endl;
    }
    cout << "\n" << endl;

    // Print recoveryBlocks (only created, no data in there yet)
    uint8_t* recoveryBlocks = new uint8_t[params.RecoveryCount * params.BlockBytes];
    cout << "Empty array 'recoveryBlocks' created (initialized) with room for " <<
    params.RecoveryCount * params.BlockBytes << " elements:" << endl;
    cout << "Index  Memory loc  Bits     " << endl;
    cout << "-----  ----------  -------- " << endl;
    for (int i = 0; i < params.RecoveryCount * params.BlockBytes; ++i)
    {
        recoveryBlocks[i] = 0
        cout << setfill('0') << setw(5) << i;
        cout << "  0x" << static_cast<void*>(&recoveryBlocks[i]);
        cout << "  " << bitset<8>(recoveryBlocks[i]) << endl;
    }

    // Generate recovery data
    if (cm256_encode(params, blocks, recoveryBlocks))
    {
        cout << "cm256_encode(params,blocks,recoveryBlocks) was not 0, exit code." << endl;
    exit(1);
    }

    // print blocks (1/3)
    cout << "filled recoveryBlocks:      " << endl;
    cout << "Index  Memory loc  Bits     " << endl;
    cout << "-----  ----------  -------- " << endl;
    for (int i = 0; i < params.RecoveryCount * params.BlockBytes; ++i)
    {
        cout << setfill('0') << setw(5) << i;
        cout << "  0x" << static_cast<void*>(&recoveryBlocks[i]);
        cout << "  " << bitset<8>(recoveryBlocks[i]) << endl;
    }

    // Initialize the indices
    cout << "" << endl;
    cout << "We now initialize the indices." << endl;
    for (int i = 0; i < params.OriginalCount; ++i)
    {
        blocks[i].Index = cm256_get_original_block_index(params, i);
        cout << "   blocks[" << i << "].Index is " << int(blocks[i].Index) << endl;
    }

    cout << "" << endl;
    cout << "Will now simulate loss of data." << endl;
    cout << "Originally, blocks[0].Block points to memory address 0x" << blocks[0].Block <<
    " and blocks[0].Index was " << int(blocks[0].Index) << endl;
    //// Simulate loss of data, subsituting a recovery block in its place ////
    blocks[0].Block = recoveryBlocks; // First recovery block
    blocks[0].Index = cm256_get_recovery_block_index(params, 0); // First recovery block index
    //// Simulate loss of data, subsituting a recovery block in its place ////
    cout << "After simulating loss of data blocks[0].Block memory address 0x" << blocks[0].Block <<
    " and blocks[0].Index is " << int(blocks[0].Index) << endl;

    // print blocks (2/3)
    cout << "i  Pointer     Value  Index" << endl;
    cout << "-  ----------  -----  -----" << endl;
    for (int i = 0; i < params.OriginalCount; ++i)
    {
        cout << i << "  0x" << blocks[i].Block << "  ";
        for (int j = 0; j < params.BlockBytes; ++j)
            cout << reinterpret_cast<char*>(blocks[i].Block)[j];
```

```cpp
        cout << "  " << (int)blocks[i].Index << endl;
    }
    cout << "\n" << endl;

    cout << "Now on to the decoding." << endl;
    if (cm256_decode(params, blocks))
    {
        cout << "cm256_decode(params, blocks) was not 0, exiting." << endl;
        exit(1);
    }

    // print blocks (3/3)
    cout << "i  Pointer     Value  Index" << endl;
    cout << "-  ----------  -----  -----" << endl;
    for (int i = 0; i < params.OriginalCount; ++i)
    {
        cout << i << "  0x" << blocks[i].Block << "  ";
        for (int j = 0; j < params.BlockBytes; ++j)
            cout << reinterpret_cast<char*>(blocks[i].Block)[j];
        cout << "  " << (int)blocks[i].Index << endl;
    }
    cout << "\n" << endl;

    // blocks[0].Index will now be 0.

    delete[] originalFileData;
    delete[] recoveryBlocks;

    return true;
}
```

I ran the error-correcting code using C++ developer command prompt by executing the following commands in the directory where the `cm256.cpp`, `cm256.h`, `gf256.cpp`, `gf256.h` and `main.cpp` files are stored (for me those files were stored at `c:/cm256code`).

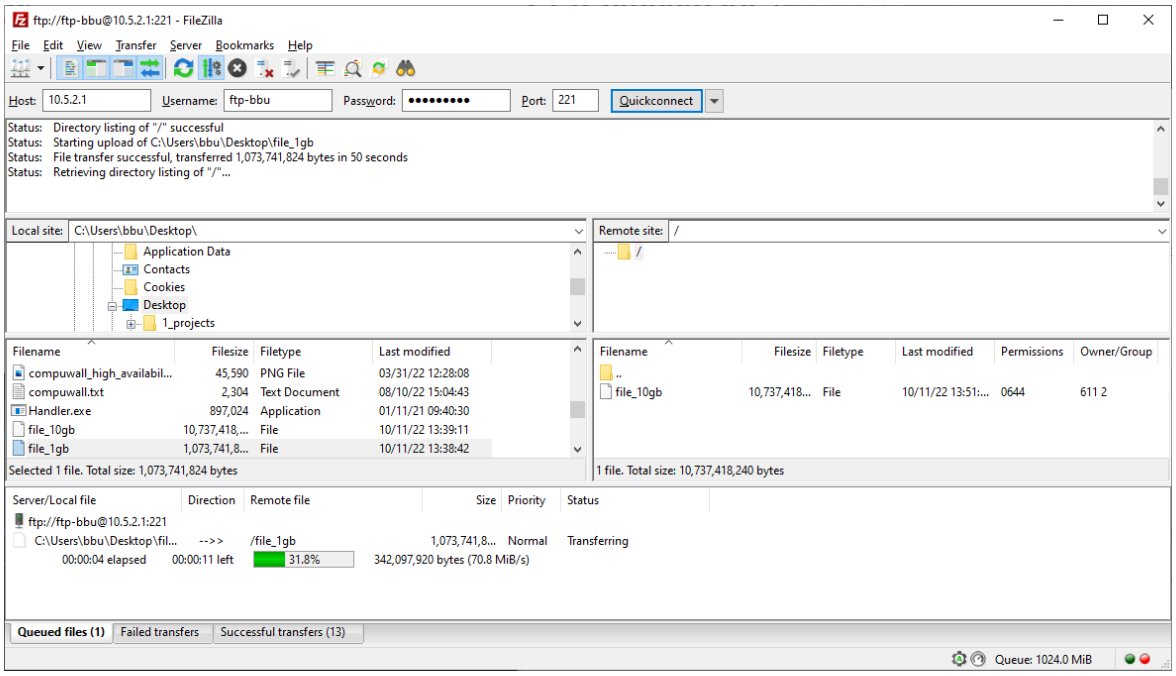


```
**********************************************************************
** Visual Studio 2022 Developer Command Prompt v17.3.5
** Copyright (c) 2022 Microsoft Corporation
**********************************************************************

C:\Program Files (x86)\Microsoft Visual Studio\2022\BuildTools>cd c:/cm256code

c:\cm256code>cl /EHsc main.cpp cm256.cpp gf256.cpp /link /out:test.exe
Microsoft (R) C/C++ Optimizing Compiler Version 19.33.31630 for x86
Copyright (C) Microsoft Corporation.  All rights reserved.

main.cpp
cm256.cpp
gf256.cpp
Generating Code...
Microsoft (R) Incremental Linker Version 14.33.31630.0
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:main.exe
/out:test.exe
main.obj
cm256.obj
gf256.obj

c:\cm256code>test
```

Appendix E

Example of MagiCtwin FTP server interface

A screenshot of a user connecting to the FTP Daemon in the MagiCtwin, using FileZilla. Other FTP applications can be used as well. In this case, a PNG file called `compuwall_high_availability.png`, a text document called `compuwall.txt`, an application called `Handler.exe` and two files called `file_10gb` and `file_1gb` are located on the users device. One file called `file_10gb` is already uploaded onto the FTP Daemon, and a file called `file_1gb` is currently being uploaded from the user's device onto the FTP Daemon. Dragging a file from left to right corresponds to uploading a data file from the user's device to the MagiCtwin's FTP Daemon on the TX side.
Connecting to the RX side can be done in a similar way. A different host, and a corresponding port number, should be entered. It is important that the username (and corresponding password) are the same for the TX and RX FTP server connections. Via a connection to the RX side's FTP server, the transmitted files can be retrieved.


ftp://ftp-bbu@10.5.2.1:221 - FileZilla
File Edit View Transfer Server Bookmarks Help
Quickconnect

Host: 10.5.2.1 Username: ftp-bbu Password: ••••••••• Port: 221

```
Status:	Directory listing of "/" successful
Status:	Starting upload of C:\Users\bbu\Desktop\file_1gb
Status:	File transfer successful, transferred 1,073,741,824 bytes in 50 seconds
Status:	Retrieving directory listing of "/"...
```

Local site: C:\Users\bbu\Desktop\

- Application Data
- Contacts
- Cookies
- Desktop
  - 1_projects

Remote site: /

- /

Filename	Filesize	Filetype	Last modified
compuwall_high_availabil...	45,590	PNG File	03/31/22 12:28:08
compuwall.txt	2,304	Text Document	08/10/22 15:04:43
Handler.exe	897,024	Application	01/11/21 09:40:30
file_10gb	10,737,418,...	File	10/11/22 13:39:11
file_1gb	1,073,741,8...	File	10/11/22 13:38:42

Selected 1 file. Total size: 1,073,741,824 bytes

Filename	Filesize	Filetype	Last modified	Permissions	Owner/Group
..					
file_10gb	10,737,418...	File	10/11/22 13:51:...	0644	611 2

1 file. Total size: 10,737,418,240 bytes

Server/Local file	Direction	Remote file	Size	Priority	Status
ftp://ftp-bbu@10.5.2.1:221					
C:\Users\bbu\Desktop\fil...	-->>	/file_1gb	1,073,741,8...	Normal	Transferring
00:00:04 elapsed	00:00:11 left	31.8%	342,097,920 bytes (70.8 MiB/s)		

Queued files (1) | Failed transfers | Successful transfers (13)

Queue: 1024.0 MiB