UNIVERSITÄT
BERN

Using polynomial systems to decode binary linear codes

Bachelor Thesis

Annina Katharina Helmy
from
Solothurn, Switzerland

Faculty of Science, University of Bern

18. September 2020

Prof. Christian Cachin
Alex Pellegrini
Cryptology and Data Security Group
Institute of Computer Science
University of Bern, Switzerland

Abstract

The Maximum Likelihood Decoding Problem (MLD) and the Multivariate Quadratic System Problem (MQ) are both NP-hard. Both problems are well known and used in various applications in cryptography. The topic of this thesis is finding a reduction from MLD to MQ and its implementation with SageMath. Given a binary linear code $\mathcal{C}$ with parity check matrix H, a syndrome s and a positive integer t, an error vector v has to be found such that $Hv^\top = s^\top$ and $w_H(v) \leq t$. This requirement is implemented and the result then expressed as an instance of MQ. Additionally, the back-transformation of the solution of the MQ instance to the solution of the MLD instance is constructed and discussed. Two different approaches for the computation of the reduction are investigated. The first approach saves all needed polynomials in different lists and composes them in the end, such that the result is a system of equations. The second approach in the paper reduces the number of variables by reusing already computed polynomials. When analysing the results, we can conduct that the resulting system of equations from the second approach is much easier to solve with SageMath. Moreover, we can observe that the first approach bases itself on more variables yet less complex polynomials while the other is built on a system of more polynomials and fewer variables. It is left open for discussion, which approach is more time efficient. Nonetheless, the second approach is more applicable when solving a given instance of a reduction from MLD to MQ in SageMath. A last step in the thesis shows how the computation works by applying it to given instance of MLD, where the error vector v is found, when applying it to the $[7, 4, 3]$-Hamming Code. The found error vector is the instance solution of the MLD problem by solving the related systems of equations, so the back-transformation from MQ to MLD is shown as well. With this, the reduction is completed.

Contents

1 Introduction **1**

2 Problems Definition and Applications to Cryptography **3**
2.1 Linear Codes Preliminaries 3
2.2 Maximum Likelihood Decoding Problem 5
2.3 Multivariate Quadratic System Problem 5
2.4 Post Quantum Cryptography 6

3 Complexity Classes P and NP **7**
3.1 Turing Machines 7
3.2 P and NP 7

4 MLD to MQ Reduction **9**
4.1 Implementation: Parity Check Constraint 9
4.2 Implementation: Weight Constraint 10
4.2.1 Weight Computation Encoding 10
4.2.2 Two Implementations - Weight Computations Encoding 13
4.2.3 Weight Constraint Encoding 18
4.2.4 Two Implementations - Weight Constraint Encoding 20
4.3 Degree Reduction Formula for Polynomials 22
4.4 MLD to MQ Reduction: Example 23
4.5 MQ to MLD 25

5 Conclusion **27**

Chapter 1

Introduction

Telecommunication involves the transmission of data. Transmission requires:

- a source
- a channel
- a receiver

Often transmissions are successful and the receiver obtains the transmitted data correctly. Sometimes however, interferences may occur, causing a so-called disturbance, also known as noise. Transmission works as follows: The source starts by sending a message. This message gets transformed into bits and these bits then get transformed into electromagnetic waves. Once these electromagnetic waves are received, they get converted back into bits and finally, the initial message is recovered. An encoder transforms a signal into bits, moreover, the encoder manipulates them (encodes them), whereas when the bits are received, a decoder has to do the reverse. To guarantee the safety of a transmission, it should (always) be possible to recover the initial signal even if during transmission corruption occurred. That is where error correcting codes come into play. One of the most used decoding techniques is the Maximum Likelihood Decoding, which was proven by Berlekamp, McEliece and van Tilborg to be an NP-complete [2] problem. The aim of this thesis is to reduce an instance of the Maximum Likelihood Decoding Problem (MLD) to an instance of a Multivariate Quadratic Equation System Problem (MQ). It will be attempted to find not only a solution for these equations but also to back-transform the solution to a solution of the original MLD instance. This reduction will be implemented in SageMath. By showing that there is a reduction and this reduction is sufficiently efficient, it follows that the second problem is at least as difficult as the first problem and subsequently, solving an instance of MLD is not "harder" to solve than an instance of MQ. Before diving into the technicalities of this work, preliminaries about linear codes and complexity are given.

Chapter 2

Problems Definition and Applications to Cryptography

When data is transmitted, an interference may occur. These are categorised into four different kinds:

- **errors**: an error occurs, when a bit is changed during transmission.
- **erasures**: an erasure happens when one bit of the code was not correctly understood by the decoder and is therefore unknown.
- **insertions**: an insertion happens when the decoder adds more bits to the code word than the originally transmitted number .
- **deletions**: a deletion happens when the decoder "deletes" bits which were originally transmitted.

The implementations given in Chapter 4 refers to a linear code (the $[7, 4, 3]$-Hamming code) which can correct only up to one occurring *error* in transmission.

2.1 Linear Codes Preliminaries

Before addressing binary linear codes, some basic linear algebra concepts have to be introduced [12].

Definition 2.1.1. A **ring** is a set R with two binary operator "$+$" and "$\cdot$", called addition and multiplication, satisfying the following axioms $\forall a, b, c \in R$:

- **associativity** of addition and multiplication: $a+(b+c) = (a+b)+c$ and $a\cdot(b\cdot c) = (a\cdot b)\cdot c$
- **commutativity** of addition: $a + b = b + a$
- **additive and multiplicative identity**: there exist different elements 0 and 1 in R such that: $a + 0 = a$ and $a \cdot 1 = a$
- **additive inverse**: for every $a \in R$ there exists an $-a \in R$ such that $a + (-a) = 0$ and $-a$ is called the **additive inverse** of a.
- **distributivity**: the following must hold: $a \cdot (b + c) = ab + ac$ and $(a + b) \cdot c = ac + bc$

Definition 2.1.2. A **field** is a ring $\mathbb{F}$. Moreover, the following *additional* axioms must be satisfied $\forall a, b, c \in \mathbb{F}$:

- **commutativity** of multiplication: $a \cdot b = b \cdot a$

- **multiplicative inverse**: for every $b \in \mathbb{F}$ there exists a $b^{-1} \in \mathbb{F}$ such that $b \cdot b^{-1} = 1$ and b^{-1} is called the multiplicative inverse of b.

Remark. A **finite field** is a field with a finite number of elements, e.g. $\mathbb{F}_p = \mathbb{Z}/p\mathbb{Z}$.

Definition 2.1.3. A **vector space** over a field $\mathbb{F}$ is a set V with two operations that satisfy the following axioms. The operations are:

- **vector addition** $+ : V \times V \to V$, where two vectors v and $w \in V$ are added, where the sum $v + w \in V$.
- **scalar multiplication** $\cdot : \mathbb{F} \times V \to V$ takes any scalar $a \in \mathbb{F}$ and $v \in V$ and generates a new vector av.

Let $u, v, w \in V$ be arbitrary vectors and let $a, b \in \mathbb{F}$ be scalars, then the following axioms must hold:

- **associativity** and **commutativity** of vector addition
- existence of **inverse and identity elements** of vector addition
- It must hold: $a(bv) = (ab)v$
- existence of an **identity element** of scalar multiplication: $1 \cdot v = v$ (1 is the multiplicative identity in $\mathbb{F}$)
- Scalar multiplication is **distributive** with respect to vector addition: $a(u + v) = au + av$
- Scalar multiplication is **distributive** with respect to scalar addition: $(a + b)v = av + bv$

Definition 2.1.4. Let V be a vector space over the field $\mathbb{F}$ and W is a subset of V, then W is a **subspace** of V if under the operations of V, W is a vector space over $\mathbb{F}$.

The above definitions will help understanding the following concepts of linear codes.

Definition 2.1.5. Let $\mathbb{F}_q$ be a finite field with q elements. Let $k, n \in \mathbb{N}$ such that $k \leq n$. Let $\mathcal{C}$ be a k-dimensional vector subspace of $(\mathbb{F}_q)^n$. We say that $\mathcal{C}$ is a $\mathbb{F}_q$ linear block code of dimension k and length n. An element of $\mathcal{C}$ is called a word of $\mathcal{C}$.

Remark. In the following, we always assume $q = 2$. A linear code over $\mathbb{F}_2$ is called a **binary** code.

Definition 2.1.6. Let $(\mathbb{F}_2)^n$ be a n-dimensional vector space. For any two vectors $x_1, x_2 \in (\mathbb{F}_2)^n$, we denote with $d(x_1, x_2)$ the number of coordinates where the two words differ and we call the integer $d(x_1, x_2)$ the **Hamming distance** between x_1 and x_2. The (Hamming) **weight** of a vector x is the number of non-zero coordinates it contains and is denote by $w_H(x)$.

Example 1. Let $x_1 = (1, 0, 1, 1)$ and $x_2 = (1, 0, 0, 0)$. Then the Hamming **distance** between x_1 and x_2 is $d(x_1, x_2) = 2$, as they differ in two coordinates.

Example 2. Let $x_1 = (1, 0, 1, 1)$. Then $w_H(x_1) = 3$ as x_1 has three nonzero coordinates.

If $\mathcal{C}$ is a linear block code, then the minimum distance between any two different words of $\mathcal{C}$ is called the distance of $\mathcal{C}$. And it follows that a binary $[n, k, d]$ code is a code of length n, dimension k and distance d with coordinates in $\mathbb{F}_2$.

Definition 2.1.7. The **parity check matrix** [9] of an $[n, k, d]$ binary linear code $\mathcal{C}$, is an $r \times n$ matrix H over $\mathbb{F}_2$ such that for every $c \in (\mathbb{F}_2)^n$ we have

$$c \in \mathcal{C} \Leftrightarrow Hc^\top = 0$$

Moreover, $\mathcal{C}$ is the kernel of H in $(\mathbb{F}_2)^n$ and we can conclude that:

$$rank(H) = n - dim\ (ker(H)) = n - k$$

Therefore, when we have linearly independent rows in H, we have that $r = n - k$ and moreover, the parity check matrix is of the size $(n - k) \times n$.

Corollary 2.1.7.1. From the above definition we can conclude that c is a code word in $\mathcal{C}$ if and only if $Hc^\top = 0$.

Definition 2.1.8. A matrix $\mathcal{G}$ whose rows form a basis of a linear code is called a **generator matrix**. The code words are all the linear combinations of the rows of $\mathcal{G}$.

Definition 2.1.9. The **dual code** of a linear code $\mathcal{C} \subset (\mathbb{F}_2)^n$ is the linear code

$$\mathcal{C}^\perp = \{x \in (\mathbb{F}_2)^n | \langle x, c\rangle = 0\ \forall c \in \mathcal{C}\}$$

With the definition of the generator matrix it follows that the parity check matrix H of a linear code $\mathcal{C}$ is the generator matrix of the dual code $\mathcal{C}^\perp$.

With the above definitions, we can now define the Maximum Likelihood Decoding Problem and the Multivariate Quadratic System Problem formally.

2.2 Maximum Likelihood Decoding Problem

The Maximum Likelihood Decoding Problem (MLD) is the standard decoding technique which allows the correction of as many errors as possible when no re-transmission of a code word is possible. The following definition of the MLD Problem considers the linearity of the code:

Definition 2.2.1. Let H be a $m \times n$ matrix over $\mathbb{F}_2$, $s \in (\mathbb{F}_2)^m$ and t a positive integer. Decide whether there is a vector $v \in (\mathbb{F}_2)^n$ of weight at most t, such that $Hv^\top = s^\top$.

The above definition can be visualized: The received word is compared to all the possible code words. The one word which is closest (minimal Hamming Distance) to the one received is likely to be the corrected code word. The weight constraint in the definition above takes care of the requirement that the compared code word is "closest" to the received. In other words: The code word $x \in C$ which is closest to the received word is then taken to be the corrected code word. Figure 2.1 shows the idea behind the MLD problem graphically.

2.3 Multivariate Quadratic System Problem

The Multivariate Quadratic System Problem is defined like the following:

Definition 2.3.1. An MQ-System of equations over $\mathbb{F}_2$ is a set of m polynomial equations of degree at most 2 in $\mathbb{F}_2[x_1, x_2, ..., x_n]$ of the form

$$S = \begin{cases} f_1(x_1, ..., x_n) & = 0 \\ f_2(x_1, ..., x_n) & = 0 \\ & \vdots \\ f_m(x_1, \ldots, x_n) & = 0 \end{cases} \tag{2.1}$$

The problem consists of the question whether the above stated multivariate quadratic equation system admits a solution.

The MQ System Problem is known to be NP-hard over any field [4].

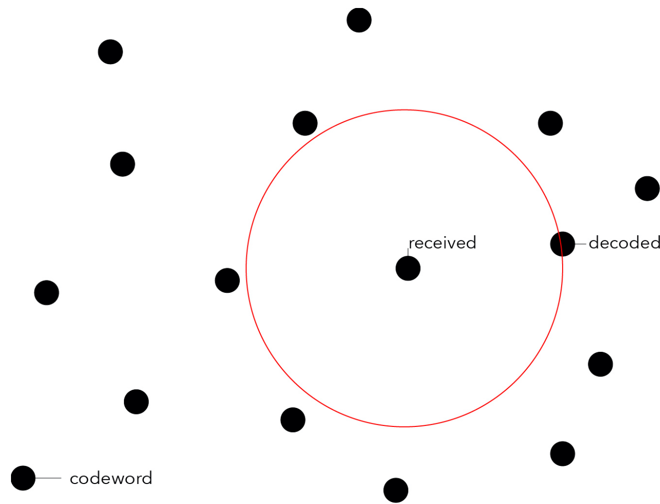

Figure 2.1. Visualization of MLD

2.4 Post Quantum Cryptography

Post-quantum cryptography involves cryptographic algorithms which are thought to be safe against an attack by a quantum computer. Good candidates for post-quantum cryptography are those cryptosystems, which are immune to attacks using the Shor Algorithm (Polynomial-time quantum computer algorithm for integer factorization). There are several approaches to finding a strong algorithm which could hold against a quantum computer. Two such approaches are related to the main focus of this thesis.

Multivariate Cryptography As the Multivariate Quadratic System Problem is known to be NP-hard over any field [4], those schemes are often very good candidates for post-quantum cryptography. Multivariate Cryptograpyh is about low-level cryptographic algorithms based on multivariate polynomials over a finite field $\mathbb{F}$. If polynomials have degree two, we talk about multivariate quadratics, solving these equations is proven to be NP-complete and believed to be hard on average. The security of multivariate schemes is based on the MQ-Problem. Multivariate Public Key Cryptography (MPKC) [10] relies on the simplicity of the operations and small fields which makes computations much easier. The main security assumption about MPKC is backed by the NP-hardness of solving nonlinear equations over finite fields. Some examples for Multivariate PKCs would be Rainbow [8] or QUARTZ [7].

Code-based Cryptography In code-based cryptography, the underlying one-way function of the cryptosystem relies on an error correcting code $\mathcal{C}$ [11] . The security of such cryptosystems rely on a variant of MLD. Some well-known code-based cryptosystems would be those from McEliece [3] or Niederreiter [5]. The McEliece cryptosystem is a candidate for post-quantum cryptography because of its immunity to attacks using the Shor algorithm. Code-based cryptography allows the construction of the most important cryptographic primitives (encryption, signature, ...) often with efficient implementations.

Chapter 3

Complexity Classes P and NP

Before diving into what the complexity classes P and NP mean, the idea of complexity in general is discussed. In computer sciences there is a branch of computational complexity theory, where the costs and resources required to solve a given computational problem are analyzed. A complexity class is the set of all computational problems which can be solved by using the same amount of certain resources. Furthermore, we call a decision problem any problem to which a solution can either be yes or no. For example the Traveling Salesman Problem: “Is this a route that visits all cities under cost c?" [14]. A fundamental part of computational complexity theory are **Turing Machines** [6].

3.1 Turing Machines

A Turing Machine (TM) models how computations work on a computer in a simple manner. A TM is a program or an algorithm which does a computation by step-wise manipulating symbols or numbers. These symbols then get written or deleted on a tape and can be read by the machine. The tape is divided into sections ("squares"). Each section can hold a symbol. One section can be found in the machine, the so called "scanned symbol". The machine remembers some scanned symbols. The behaviour of the machine is dictated by the scanned symbol and the configurations, which tell the machine what to do, when a specific symbol is scanned. [1]. One can differentiate between deterministic and non-deterministic TMs [6]:

- **deterministic**: the set of rules for a scanned symbol impose at most *one* action to be performed for any given situation
- **non-deterministic**: may have a set of rules which then prescribe one or more actions which will be performed for a given situation

3.2 P and NP

Definition 3.2.1. **NP** (Non-deterministic polynomial time)is a complexity class used to classify decision problems. It is the set of all decision problems of which the validity of any proposed solution S to an instance I of the problem can be checked *efficiently*. In other words - the instance can be checked by a deterministic TM and can be efficiently solved by a non-deterministic TM. [14]

Definition 3.2.2. A problem is in **P**, if there exists a deterministic TM which can solve it in polynomial time. It also is a subset of NP, as NP is the generalization of P. Therefore, this definition implies: $P \subseteq NP$. Most experts believe that P is a proper subset of NP, but this still remains unproven.

In the introduction it was stated, that MLD was proven to be NP-complete in 1978 by Berlekmap, McEliece and van Tilborg in [2]. Before looking at the formal definition of NP-completeness, the concept of a reduction must be presented:

Reduction Let A and B be two decision problems. The set of all instances of problem A is called Σ^* and the corresponding set of B is called Γ^* (is the set of all the instances of problem B.) We call a decision problem A *reducible* to a decision problem B if there exists a function $f : \Sigma^* \to \Gamma^*$ which can be computed in polynomial time by a deterministic algorithm such that for all $w \in \Sigma^*$ it holds that:

$$w \in A \Leftrightarrow f(w) \in B$$

Now, the formal definition of NP-completeness is given [14]:

Definition 3.2.3. A problem is **NP-complete** if it is in NP, and every problem in NP reduces to it in polynomial time (in the size of the input). A problem is **NP-hard** if an NP-complete problem reduces to it.

Within in thesis, a reduction from MLD to MQ (and vice-versa) will be implemented. Thus, we do not only compute a reduction from MLD to MQ, but this reduction should happen in polynomial time.

Chapter 4

MLD to MQ Reduction

In 2.2.1 the MLD problem was introduced. An instance of MLD must fulfill the following requirements:

1. **parity check constraint**: the solution v has to fulfill $Hv^\top = s^\top$
2. **weight constraint**: for the solution v it must hold that $w_H(v) \leq t$

Denote c as the transmitted vector. During transmission an error e may occur and the word $z = c + e$ is received. To check whether the received word contains an error, one conducts a left multiplication of the word with the parity check matrix. Since

$$Hz = H(c + e) = Hc + He = \mathbf{0} + He = s$$

it is possible to conclude that the received word contains an error if and only if s is not the zero $(n - k)$-vector. The vector s is called the *syndrome* associated to v. The syndrome only depends on the error which occurs during transmission and not on the specific word. Once e is known, it becomes trivial to decode c as $c = z - e$. By construction, v is the **error vector** we want to find.

4.1 Implementation: Parity Check Constraint

The first step in the implementation requires the initial condition which an instance of the MLD must satisfy: the **parity check constraint**. A solution v must satisfy $Hv^\top = s^\top$. The syndrome s and the matrix H are given. Therefore, a program must be implemented such that it returns m ($=$ number of rows of matrix H) linear binary equations which have the coordinates of s, i.e. $s_1, ..., s_m$, are the solutions. This results in a set of m linear equations $f_i \in \mathbb{F}_2[v_1, \ldots, v_n]$ of the form $f_i = \sum_{j=1}^{n} h_{i,j}v_j + s_i$. The following listing shows how the above described process is implemented with SageMath:

```
1 H = Matrix([[1,1,0,1,1,0,0], [1,0,1,1,0,1,0], [0,1,1,1,0,0,1]])
2 s = vector([0,1,1])
3 n = H.ncols()
4 P0 = PolynomialRing(GF(2),n,names="v"); P0
5 P0.inject_variables()
6 v = P0.gens()
7 (H*vector(P.gen(i) for i in range(n))-s).list()
```

Listing 4.1. without User Interaction

The above code snippet has the following output:

```
1 #Output:
2 [v0 + v1 + v3 + v4, v0 + v2 + v3 + v5 + 1, v1 + v2 + v3 + v6 + 1]
```

In the above code, one firstly defines a polynomial ring in which the computations should run. Because the definition of a polynomial ring is fundamental for SageMath, the formal definition of one is given:

Definition 4.1.1. Let R be ring. A polynomial with coefficients in R is a formal expression

$$\sum_{i=0}^{\infty} a_i X^i, a_i \in R \; \forall i \geq 0$$

and almost all a_i are equal to zero (that is, there exists an n such that $a_i = 0 \; \forall i \geq n$.) The a_i are called coefficients of the polynomial $\sum_{i=0}^{\infty} a_i X^i$. The set of all polynomials with coefficients in R is denoted $R[X]$, it is a ring called the polynomial ring in one indeterminate X over R.

In this scenario, we define $R = \mathbb{F}_2$. Therefore, the polynomial ring used is $\mathbb{F}_2[v_1, \ldots, v_n]$. The following code snippet returns the equations, when the matrix H is multiplied with the unknown error vector v, moreover the syndrome s is subtracted so. Consequently, the code returns polynomials.

```
1 (H*vector(P.gen(i) for i in range(n))-s).list()
```

The $gen(.)$ function returns v_i which corresponds to the i-th coordinate of the vector v which is multiplied by H. The function `list()` returns a list containing the resulting polynomials.

4.2 Implementation: Weight Constraint

The requirement of the weight constraint (the solution v has to satisfy $w_H(v) \leq t$) is implemented in two steps:

- weight computation encoding: $w_H(v) = w$
- weight constraint encoding: $w_H(v) \leq t$

4.2.1 Weight Computation Encoding

Consider the following pseudocode which, given a binary input vector v, computes the weight of vector v.

Input:binary input vector v **Output:**weight of vector v **Initialize:** n = length of vector v **and** counter = 0 **and** i = 0 ;
while $i < n$ **do**
  if $v[i] \neq 0$ **then**
    counter = counter + 1 ;
  else
    i
  end
  = i + 1 ;
  return counter;
end

Roughly said, a binary counter is implemented which stores the Hamming weight of the vector v step-wise until the i-th coordinate (see pseudocode above). Formally, it is required for the following to be implemented [13]:

Definition 4.2.1. Let $v \in (\mathbb{F}_2)^n$ and $l = \lfloor \log_2(n)+1 \rfloor$. Consider the truncation map $\pi_i : (\mathbb{F}_2)^l \rightarrow (\mathbb{F}_2)^l$ defined as $\pi_i : (v_1, ..., v_l) = (v_1, ..., v_i, 0, ..., 0)$. The set of variables $a_j^{(i)}$ for $i = 0, ..., n$ and

$j = 1, ..., l$ is defined, such that $a^{(i)}(v) = (a_1^{(i)}(v), ..., a_l^{(i)}(v)) \in (\mathbb{F}_2)^l$ can be regarded as the binary expansion of $w_H(\pi_i(v))$. Set $a^{(0)} = (0, ..., 0)$ while $a^{(i)}$ is defined recursively for $i = 1, ..., n$ by computing its coefficients $a_j^{(i)}$ as

$$a_j^{(i)}(v) = a_j^{(i-1)} + (\prod_{h=1}^{j-1} a_h^{(i-1)})v_i \tag{4.1}$$

and finally $a = a^{(n)} = (a_1^{(n)}, ..., a_l^{(n)})$.

Equation 4.1 represents a binary addition of the Hamming Weight of v. In general, the addition of binary numbers works as follows:

Example 1. Let $a = (1, 0, 1)$. The following example shows how 1 is added to vector a. When transforming 1 into binary, one has $(1)_{10} = (0, 0, 1)_2$, therefore:

$$\begin{array}{rl} 0\,0\,1\,0 & \text{(carry)} \\ 1\,0\,1 & \\ +\ \ 0\,0\,1 & \\ \hline 1\,1\,0 & \end{array} \tag{4.2}$$

The last two bits get added which gives $1 + 1 = 0$ in $\mathbb{F}_2$, but because $(0)_2 = (2)_{10}$, one has to carry 1 to the next bit. Proceeding to the second bit, one computes $0 + 1 = 1$. This means, there is no carry and one proceeds to the first bit. Again, the following addition is computed: $0 + 1 = 1$. Every time $1 + 1$ is added, one has to carry an extra 1 to the next bit.

Remark. For the remainder this thesis, the least significant bit always comes first.

The following example demonstrates how (4.1) works. In $a^{(i)}$ the hamming weight of vector v gets stored until the i-th coordinate in binary step by step.

Example 2. Let $v = (1, 0, 1, 1, 0)$. So $l = \lfloor log_2(5) + 1 \rfloor$. So j will iterate from $j = 1, \ldots, 3$ and $i = 1, \ldots, 5$. In the end, one wants to have five vectors $a^{(1)}, a^{(2)}, \ldots, a^{(5)}$ where the Hamming weight until the i-th coordinate of vector v is stored. Remember, that a is in binary and the least significant bit comes first. $i = 1$.

$$a_1^{(1)}(v) = a_1^{(0)} + (\prod_{h=1}^{0} a_h^{(0)})v_1 = 0 + 1 = 1$$

$$a_2^{(1)}(v) = a_2^{(0)} + (\prod_{h=1}^{1} a_h^{(0)})v_1 = 0 + 0 = 0$$

$$a_3^{(1)}(v) = a_3^{(0)} + (\prod_{h=1}^{2} a_h^{(0)})v_1 = 0 + 0 = 0$$

$a^{(1)} = (1, 0, 0)$. This is obviously true, as $\pi_1(v) = (1, 0, 0, 0, 0)$ and the Hamming weight of this vector is 1. $i = 2$.

$$a_1^{(2)}(v) = a_1^{(1)} + (\prod_{h=1}^{0} a_h^{(1)})v_2 = 1 + 0 = 1$$

$$a_2^{(2)}(v) = a_2^{(1)} + (\prod_{h=1}^{1} a_h^{(1)})v_2 = 0 + 0 = 0$$

$$a_3^{(2)}(v) = a_3^{(1)} + (\prod_{h=1}^{2} a_h^{(1)})v_2 = 0 + 0 = 0$$

And it follows that: $a^{(2)} = (1,0,0)$. Also true, as $\pi_2(v) = (1,0,0,0,0)$.

$i = 3$. As the example is relatively simple one can easily deduce that in the following equation a gets incremented.

$$a_1^{(3)}(v) = a_1^{(2)} + (\prod_{h=1}^{0} a_h^{(2)})v_3 = 1 + (1 * 1) = 1 + 1 = 0$$

Above, $a_1^{(2)}$ corresponds (in a long addition) to the last bit of the first addend. $(\prod_{h=1}^{0} a_h^{(2)})v_3$ will be 1, if v_3 is 1. So the last bit will be 0. The **carry** will be added in the next step:

$$a_2^{(3)}(v) = a_2^{(2)} + (\prod_{h=1}^{1} a_h^{(2)})v_3 = a_2^{(2)} + a_1^{(2)} * v_3 = 0 + 1 * 1 = 1$$

$a_2^{(2)}$ corresponds to the next bit of the first Addend. $a_1^{(2)}$ is the **carry** of the long addition. So here, the actual increment appears.

$$a_3^{(3)}(v) = a_3^{(2)} + (\prod_{h=1}^{2} a_h^{(2)})v_3 = 0 + 0 = 0$$

$a^{(3)} = (0,1,0)$. Here the first difference appears. The Hamming weight is two, which is still true as $\pi_3(v) = (1,0,1,0,0)$. $i = 4$.

$$a_1^{(4)}(v) = a_1^{(3)} + (\prod_{h=1}^{0} a_h^{(3)})v_4 = 0 + 1 = 1$$

$$a_2^{(4)}(v) = a_2^{(3)} + (\prod_{h=1}^{1} a_h^{(3)})v_4 = 1 + 0 = 1$$

$$a_3^{(4)}(v) = a_3^{(3)} + (\prod_{h=1}^{2} a_h^{(3)})v_4 = 0 + 0 = 0$$

And it follows that: $a^{(4)} = (1,1,0)$ with $\pi_4(v) = (1,0,1,1,0)$ and the corresponding Hamming weight is 3, as displayed in $a^{(4)}$. $i = 5$.

$$a_1^{(5)}(v) = a_1^{(4)} + (\prod_{h=1}^{0} a_h^{(4)})v_5 = 1 + 0 = 1$$

$$a_2^{(5)}(v) = a_2^{(4)} + (\prod_{h=1}^{1} a_h^{(4)})v_5 = 1 + 0 = 1$$

$$a_3^{(5)}(v) = a_3^{(4)} + (\prod_{h=1}^{2} a_h^{(4)})v_5 = 0 + 0 = 0$$

Lastly, $a^{(5)} = (1,1,0)$ which, as an integer is 3. This is exactly the weight of v: $w_H(v) = w_H(1,0,1,1,0)$.

4.2.2 Two Implementations - Weight Computations Encoding

In this thesis, the weight computation encoding is implemented in two ways. The first implementation uses lists. The second implementation works by updating two vectors directly. The following two listings gives an overview of the complete code and is then followed by an analysis of the different code snippets:

```
H = Matrix([[1,1,0,1,1,0,0], [1,0,1,1,0,1,0], [0,1,1,1,0,0,1]])
s = vector([0,1,1])
n = H.ncols()
P0 = PolynomialRing(GF(2),n,names="v"); P0
P0.inject_variables()
v = P0.gens()
l = floor(log(n,2)) + 1
all = (n*l)

P1 = PolynomialRing(P0, n+1,l, var_array=('a')); P1
P1.inject_variables()
a = P1.gens()

#create list of product - step by step
L = []
for i in range(l,all+l,l):
    for h in range(l):
        L.append(prod(a[k] for k in range(i-l, i-l+h)))
print(L)

#create list of vector v = (v1, v1, v1, v2, v2, v2, ...)
V = []
for m in range(n):
    for n in range(l):
        V.append(P0.gen(m))
print(V)

# create equations
EQ = []
for j in range(l,all+l):
    EQ.append(a[j-l] +V[j-l]*L[j-l] + a[j])
print(EQ)
```

Listing 4.2. Implementation 1

```
H = Matrix([[1,1,0,1,1,0,0], [1,0,1,1,0,1,0], [0,1,1,1,0,0,1]])
s = vector([0,1,1])

n = H.ncols()
l = floor(log(n,2)) + 1
all = (n*l)

P0 = PolynomialRing(GF(2),n,names="v"); P0
P0.inject_variables()
v = P0.gens()

P1 = PolynomialRing(P0, n+1,l, var_array=('a')); P1
P1.inject_variables()

P2 = PolynomialRing(P1, l, names = "b"); P2
P2.inject_variables()
b = P2.gens()

L0 = vector(P2.gen(i) for i in range(l))
L1 = vector(P2.gen(i) for i in range(l))

#product
```

```
23 for i in range(n):
24     for j in range(l):
25         if j == 0:
26             L0[j] = L0[0]
27         else:
28             L0[j] = L0[j]*L0[j-1]
29     #equations
30     for j in range(l):
31         if j == 0:
32             L1[j] = L0[j] + v[i]
33         else:
34             L1[j] = L0[j]/L0[j-1] + L0[j-1]*v[i]
35     L0 = L1[:]
36 
37 for j in range(l):
38     L0[j] = L0[j] + b[j]
39 print(L0)
```

Listing 4.3. Implementation 2

Both codes start with setting the length of vector v and also defining l (see Definition 4.2.1):

```
1 n = H.ncols() #numberOfVariables
2 l = floor(log(n,2)) + 1
3 all = (n*l)
```

Listing 4.4. Length of v and l

Then the polynomial rings, where the resulting set of polynomials will live in, are defined. One difference between the two implementations is that in Implementation 2 l new variables called $b_0, b_1, \ldots, b_l$ have to be added. The reason for this will become obvious, when talking about Implementation 2.

```
1 P0 = PolynomialRing(GF(2),numberOfVariables,names="v"); P0
2 P0.inject_variables()
3 v = P.gens()
4 
5 #only needed in Implementation 2
6 P1 = PolynomialRing(P0, l, names = "b"); P1
7 P1.inject_variables()
8 b = P1.gens()
9 
10 P2 = PolynomialRing(P1, n+1,l, var_array=('a')); P2
11 P2.inject_variables()
12 
13 #Output:
14 Defining v0, v1, v2, v3, v4, v5, v6
15 Defining b0, b1, b2
16 Defining a00, a01, a02, a10, a11, a12, a20, a21, a22, a30, a31, a32, a40, a41,
    a42, a50, a51, a52, a60, a61, a62, a70, a71, a72
```

Listing 4.5. Defining the variables

A multivariate polynomial ring in `v0, v1, v2, v3, v4, v5, v6` (as we have `n` as a constant) over $\mathbb{F}_2$ is defined. Vector v, vector b and the variables $a_j^{(i)}$ have to live in the same polynomial ring, therefore $P2$ is defined as a multivariate polynomial ring in `a00, a01, a02, a10, a11, a12, a20, a21, a22, a30, a31, a32, a40, a41, a42, a50, a51, a52, a60, a61, a62, a70, a71, a72` over a multivariate polynomial ring in `b0, b1, b2` over a multivariate polynomial ring in `v0, v1, v2, v3, v4, v5, v6` over $\mathbb{F}_2$. So this actually means, that the coefficients of polynomials in $P2$ are in $P1$ and coefficients of polynomials in $P1$ are in $P0$. This must be true, as the vectors which are constructed are actual polynomials. After those preliminaries, the above mentioned code snippets behave differently. Firstly, *Implementation 1* is discussed, followed by *Implementation 2*.

Implementation 1

The implementation bases itself on generating lists and composes those in the end as a system of equations. In the first list L, each component $L[j]$ contains the product $\prod_{h=1}^{j-1} a_h^{(i-1)}$, for each $j \in \{1, \ldots, l\}$.

```
#create list of product - step by step
L = []
for i in range(l,all+l,l):
    for h in range(l):
        L.append(prod(a[k] for k in range(i-l, i-l+h)))
```

Listing 4.6. List L

```
#Output:
[1, a00, a00*a01, 1, a10, a10*a11, 1, a20, a20*a21, 1, a30, a30*a31, 1, a40, a40
    *a41, 1, a50, a50*a51, 1, a60, a60*a61]
```

Listing 4.7. Output L

Next, the list called V is generated. It has to have the same length as L otherwise it wouldn't be possible to combine them so easily. V consists of the components of vector v, repeated l-times, before filling the $n * (l + 1)$-th spot in the list with the next component of vector v.

```
#create list of vector: (v1, v1, v1, v2, v2, v2, ...)
V = []
for m in range(n):
    for n in range(l):
        V.append(P1.gen(m))
```

Listing 4.8. List V

```
#Output:
[v0, v0, v0, v1, v1, v1, v2, v2, v2, v3, v3, v3, v4, v4, v4, v5, v5, v5, v6, v6,
     v6]
```

Listing 4.9. Output V

In the last step of the code, the polynomials with the above defined lists are generated. The polynomials are saved in list EQ.

```
EQ = []
for j in range(l,all):
    EQ.append(a[j-l] +V[j-l]*L[j-l] + a[j])
print(EQ)
```

Listing 4.10. List EQ

```
#Output:
[a00 + a10 + v0, v0*a00 + a01 + a11, v0*a00*a01 + a02 + a12, a10 + a20 + v1, v1*
    a10 + a11 + a21, v1*a10*a11 + a12 + a22, a20 + a30 + v2, v2*a20 + a21 + a31,
     v2*a20*a21 + a22 + a32, a30 + a40 + v3, v3*a30 + a31 + a41, v3*a30*a31 +
    a32 + a42, a40 + a50 + v4, v4*a40 + a41 + a51, v4*a40*a41 + a42 + a52, a50 +
     a60 + v5, v5*a50 + a51 + a61, v5*a50*a51 + a52 + a62, a60 + a70 + v6, v6*
    a60 + a61 + a71, v6*a60*a61 + a62 + a72]
```

Listing 4.11. Output EQ

Implementation 2

The second implementation updates two vectors in every iteration and may be more efficient in generating the polynomials than the first implementation. Below, every step of the implementation is discussed in detail. The variables $a_j^{(i)}$ and v_i are generated as in Implementation 1.

In Implementation 2, vectors and not lists are used. The idea behind this is, that one only needs two vectors, which then get updated in turn throughout the loop in the code. Also, when analyzing (4.1) one quickly realizes that actually only the variables $a_{00}, a_{01}, \ldots, a_{0l}$ are needed, as all the other variables are defined *recursively* on top of them. Before analysing the code, the above statement is illustrated in the following example.

Example 3. Let $v = (1, 0, 1)$. So $l = 2$ and $j = 1, 2$ and $i = 1, 2, 3$. $i = 1$

$$a_1^{(1)}(v) = a_1^{(0)} + (\prod_{h=1}^{0} a_h^{(0)})v_1 = a_1^{(0)} + v_1 \tag{4.3}$$

$$a_2^{(1)}(v) = a_2^{(0)} + (\prod_{h=1}^{1} a_h^{(0)})v_1 = a_2^{(0)} + a_1^0 * v_1 \tag{4.4}$$

$i = 2$

$$a_1^{(2)}(v) = a_1^{(1)} + (\prod_{h=1}^{0} a_h^{(1)})v_2 = a_1^{(1)} + v_2 = a_1^{(0)} + v_1 + v_2 \tag{4.5}$$

$$a_2^{(2)}(v) = a_2^{(1)} + (\prod_{h=1}^{1} a_h^{(1)})v_2 = a_2^{(1)} + a_1^{(1)} * v_2 = a_2^{(0)} + a_1^0 * v_1 + (a_1^{(0)} + v_1) * v_2 \tag{4.6}$$

$i = 3$

$$a_1^{(3)}(v) = a_1^{(2)} + (\prod_{h=1}^{0} a_h^{(2)})v_3 = a_1^{(2)} + v_3 = a_1^{(0)} + v_1 + v_2 + v_3 \tag{4.7}$$

$$a_2^{(3)}(v) = a_2^{(2)} + (\prod_{h=1}^{1} a_h^{(2)})v_3 = a_2^{(2)} + a_1^2 * v_3 = a_2^{(0)} + a_1^0 * v_1 + (a_1^{(0)} + v_1) * v_2 + (a_1^{(0)} + v_1 + v_2) * v_3 \tag{4.8}$$

The equations acquire greater complexity, however in terms of reusing code it's very elegant.

To start off, two vectors with length l are defined. Those vectors $L0$ and $L1$ will be reused and updated in the code. The very last vector in the following iterations will contain the binary representation of the weight of v.

```
L0 = vector(P2.gen(i) for i in range(l))
L1 = vector(P2.gen(i) for i in range(l))
```

Listing 4.12. Definition of two vectors for reuse

The outer loop iterates through the vector of length n. The first inner loop is in `range(l)` and generates the product.

```
for j in range(l):
        if j == 0:
            L0[j] = L0[0]
        else:
            L0[j] = L0[j]*L0[j-1]
```

Listing 4.13. Product

The output for $i = 1$ would be:

```
#Output:
(a00, a00*a01, a00*a01*a02)
```

Listing 4.14. Output L0 in first loop and i = 1

In the second loop we generate the polynomials we need.

```python
for j in range(l):
        if j == 0:
            L1[j] = L0[j] + v[i]
        else:
            L1[j] = L0[j]/L0[j-1] + L0[j-1]*v[i]

    L0 = L1[:]
```

Listing 4.15. Equations

If $j = 0$ nothing has to be computed, as the product in (4.1) would just be 1. Therefore, the first if-clause returns the first component of the vector and saves it in $L1$.
For the case $j \neq 0$ the polynomials are computed by doing a division and a multiplication. In $L0$ the polynomials of the $(i-1)$-th iteration are stored. Because $L0[j]$ is composed of $L0[j] \cdot L0[j-1]$, one can divide $L0[j]$ by $L0[j-1]$ and the result would just be the first variable in the equation, as there is no remainder.
Then, as the product is stored in $L0[j-1]$ it gets multiplied with the vector component. This polynomial will be stored in $L1[j]$. In the last step, one sets $L0 = L1$. Therefore, the vector $L0$ can be reused and the next iteration step is saved in $L1$.
The equation (4.1) encodes the weight of vector v. At the moment, the weight of the vector is only stored in $L0$ - but $L0$ is needed as a polynomial. The weight computation has to be a polynomial, which is set to zero and solved in the end. Currently, only the right side of the equation is saved in $L0$. But the left side in $L0$ has to be saved as well such that the whole set of polynomials is complete when setting those to zero. Otherwise, the weight computation is not taken into account when solving the set of equations. A small example should show the necessity of introducing new variables `b0,b1,...,bl`:

Example 4. Let $n = 3, l = 2$ and $v = (0, 1)$. In $L0$ the following is stored:

$$L0 = [(a_0 + v_0) + v_1, (a_1 + a_0 v_0) + (a_0 + v_0)v_1] \tag{4.9}$$

When trying to solve those polynomials, they have to be set to zero and the output is the following (let the solution vector be $x_i = [a_0, a_1, v_0, v_1]$):

$$a_0 + v_0 + v_1 = 0 \tag{4.10}$$
$$a_1 + a_0 v_0 + a_0 v_1 + v_0 v_1 = 0 \tag{4.11}$$

$$x_1 = [0,0,0,0], x_2 = [0,1,1,1], x_3 = [1,0,1,0], x_4 = [1,0,0,1], x_5 = [1,1,1,0], x_6 = [1,1,0,1],$$

The variables $a_0 = a_1 = 0$ have to be set to zero, such that the solution x_1 is selected, but the resulting vector $v = (0, 0)$ is not correct. The weight is correctly encoded $([(a_0 + v_0) + v_1, a_1 + a_0 v_0 + a_0 v_1 + v_0 v_1] = [(0 + 0) + 1, 0 + 0 + 0 + 0] = [1, 0])$, but when solving those equations, the encoded weight is nowhere to be found. As already mentioned, the whole equation was not taken into account, some solutions have been excluded, so one has to introduce new variables. Let those variables be called b_0, b_1. And the new set of equations would be:

$$b_0 + a_0 + v_0 + v_1 = 0 \tag{4.12}$$
$$b_1 + a_1 + a_0 v_0 + a_0 v_1 + v_0 v_1 = 0 \tag{4.13}$$

When setting the variables $a_0 = a_1 = 0$, the resulting system is

$$b_0 + v_0 + v_1 = 0 \tag{4.14}$$
$$b_1 + v_0 v_1 = 0 \tag{4.15}$$

The solution vectors of the form $x_i = [b_0, b_1, v_0, v_1]$ would be:

$$x_1 = [0,0,0,0], x_2 = [1,0,1,0], x_3 = [1,0,0,1], x_4 = [0,1,1,1].$$

It can be seen that in x_3 the first two entries are the correctly encoded weight of vector v. So now the weight is correctly encoded and the constraint is satisfied. So when introducing those new variables $b_0, b_1, ...b_l$, $L0$ is used as a polynomial and not only for storage. So with the following code snippet, $L0$ is introduced as a polynomial, with the weight of the vector correctly encoded:

```
for j in range(l):
    L0[j] = L0[j] + b[j]
```

The output of this code would be the following:

```
(a00 + b0 + v0 + v1 + v2 + v3 + v4 + v5 + v6, (v0 + v1 + v2 + v3 + v4 + v5 + v6)
    *a00 + a01 + b1 + v0*v1 + v0*v2 + v1*v2 + v0*v3 + v1*v3 + v2*v3 + v0*v4 + v1
    *v4 + v2*v4 + v3*v4 + v0*v5 + v1*v5 + v2*v5 + v3*v5 + v4*v5 + v0*v6 + v1*v6
    + v2*v6 + v3*v6 + v4*v6 + v5*v6, (v0*v1 + v0*v2 + v1*v2 + v0*v3 + v1*v3 + v2
    *v3 + v0*v4 + v1*v4 + v2*v4 + v3*v4 + v0*v5 + v1*v5 + v2*v5 + v3*v5 + v4*v5
    + v0*v6 + v1*v6 + v2*v6 + v3*v6 + v4*v6 + v5*v6)*a00^2 + (v0 + v1 + v2 + v3
    + v4 + v5 + v6)*a00*a01 + (v0^2*v1 + v0^2*v2 + v0*v1*v2 + v1^2*v2 + v0^2*v3
    + v0*v1*v3 + v1^2*v3 + v0*v2*v3 + v1*v2*v3 + v2^2*v3 + v0^2*v4 + v0*v1*v4 +
    v1^2*v4 + v0*v2*v4 + v1*v2*v4 + v2^2*v4 + v0*v3*v4 + v1*v3*v4 + v2*v3*v4 +
    v3^2*v4 + v0^2*v5 + v0*v1*v5 + v1^2*v5 + v0*v2*v5 + v1*v2*v5 + v2^2*v5 + v0*
    v3*v5 + [...]
```

Listing 4.16. Equations

Running the code for $v = (1, 0, 1, 1, 0, 0, 0)$ returns:

```
(a00 + b0 + 1, a00 + a01 + b1 + 1, a00^2 + a00*a01 + a01 + a02 + b2)
```

Listing 4.17. Binary weight of vector v

And setting `a00, a01, a02` to zero (as this is their original value), and solving the equations for b_0, b_1, b_2 (the b_i's represent the binary weight of vector v) the weight is: $(1, 1, 0)_2 = (3)_{10}$ (which is true).

4.2.3 Weight Constraint Encoding

In the next step a polynomial is defined which can compare two integers $c_1, c_2 \in \mathbb{Z}$ and returns 0 if and only if $c_1 \leq c_2$. When representing an integer in binary, the most significant bit is decisive for comparison. Let two vectors $a, t \in (\mathbb{F}_2)^l$ be defined and let their most significant bits be a_l and t_l respectively. To do so, the following procedure is applied, starting from $j = l$:

- if $a_j = t_j$ then move to the next bits a_{j-1} and t_{j-1}.
- if $a_j \neq t_j$, output a_j
- if all bits are checked, output 0.

Now, a polynomial $F \in \mathbb{F}_2[a_1, ..., a_l, t_1, ..., t_l]$ is defined such that:

$$F(a,t) = \begin{cases} 0 \text{ if } int(a) \leq int(t) \\ 1 \text{ otherwise} \end{cases} \tag{4.16}$$

When defining $g_h(a,t) = (a_h + t_h) \in \mathbb{F}_2[a_h, t_h]$ for every $h = 1, ..., l$, one can notice that $g_h(a,t) = 0$ if $a_h = t_h$ (this follows, because if they are equal, it is either $0 + 0 = 0$ or $1 + 1 = 0(mod\ 2)$ for every a_h, t_h). In the next part of the implementation for the weight constraint encoding, the polynomials for $j = 1, ..., l$ are defined:

$$f_j = \left(\prod_{h=j+1}^{l} (g_h + 1) \right) g_j \tag{4.17}$$

where $f_j \in \mathbb{F}_2[a_1, ..., a_l, t_1, ..., t_l]$. The polynomial $f_j(a,t)$ will only be 0 if either the product is 0 or $g_j(a,t) = 0$. Moreover, the polynomial $f_j(a,t) = 1$ for at most one value of j. One has that $f_j(a,t) = 1$ if a and t differ in the j-th coordinate. This works only if the product is 1 and the polynomial $g_j(a,t) = 1$. The product is 1 only if all the factors are equal to 1. This is the case, when $a_h = t_h$ for $h = j+1, ...l$. To demonstrate the purpose of the above set of polynomials which is locating the most significant bit in where a and t differ, another example is shown: Let $a = (1,0,1,0)$ and $t = (1,1,1,0)$. So $l = 4$ and:

$$f_j = \left(\prod_{h=j+1}^{4} (g_h + 1) \right) g_j \tag{4.18}$$

$j = 1$.

$$\begin{aligned} f_1 = \left(\prod_{h=2}^{4} (g_h + 1) \right) g_j &= ((g_2+1)*(g_3+1)*(g_4+1))*g_1 \\ &= ((a_2+t_2+1)(a_3+t_3+1)(a_4+t_4+1))*(a_1+t_1) \\ &= (0+1+1)*(1+1+1)*(0+0+1)*(1+1) = 0 \end{aligned} \tag{4.19}$$

$j = 2$.

$$\begin{aligned} f_2 = \left(\prod_{h=3}^{4} (g_h + 1) \right) g_j &= ((g_3+1)*(g_4+1))*g_2 \\ &= ((a_3+t_3+1)(a_4+t_4+1))*(a_2+t_2) \\ &= (1+1+1)*(0+0+1)*(0+1) = 1 \end{aligned} \tag{4.20}$$

$j = 3$.

$$\begin{aligned} f_3 = \left(\prod_{h=4}^{4} (g_h + 1) \right) g_j &= ((g_4+1))*g_3 \\ &= ((a_4+t_4+1))*(a_3+t_3) \\ &= (0+0+1)*(1+1) = 0 \end{aligned} \tag{4.21}$$

$j = 4$.

$$\begin{aligned} f_4 = \left(\prod_{h=5}^{4} (g_h + 1) \right) g_j &= (1)*g_4 \\ &= ((1))*(a_4+t_4) \\ &= (1)*(0+0) = 0 \end{aligned} \tag{4.22}$$

One can directly see that a and t differ in the second bit, because it is known from the above computations that $f_2 = 1$. To check whether $int(a) \leq int(t)$ the polynomial F given in (25) has to be implemented. Therefore, let

$$F = \sum_{j=1}^{l} f_j \cdot (t_j + 1) \in \mathbb{F}_2[a_1, ..., a_l, t_1, ..., t_l] \tag{4.23}$$

where a and t belong to $(\mathbb{F}_2)^l$. Then, $F(a,t) = 0$ if and only if $int(a) \leq int(t)$. A product is zero exactly when one of the factors is zero. Consequently, $F(a,t) = 0$ only if either

- $f_j = 0 \; \forall j \in (1, \ldots l)$ or

- $t_j + 1 = 0$

Case 1: $f_j = 0 \; \forall j \in (1, \ldots l)$ This is only true if $a = t$. Otherwise, there is at most one value of j for which $f_j(a,t) = 1$. **Case 2:** $t_j + 1 = 0$ This is only true, when $t_j = 1$. If this is the case (and assuming $f_j = 1$), then $int(a) \leq int(t)$. Assuming $f_j = 1$, it can be concluded, that a and t differ in this bit. Moreover, when $t_j = 1$ it is known that at that bit where a and t differ, t must be 1 and a must be 0, concluding that $a \leq t$.

The above described step is now added to the example:

$$F = \sum_{j=1}^{4} f_j \cdot (t_j + 1) = f_1 \cdot (t_1 + 1) + f_2 \cdot (t_2 + 1) + f_3 \cdot (t_3 + 1) + f_4 \cdot (t_4 + 1) = 0 \tag{4.24}$$

Because F is zero, it can be deduced that a must be smaller than t. In the case of the weight constraint encoding, two different implementations were computed depending on the ideas of the weight computation encoding in 4.2.2.

4.2.4 Two Implementations - Weight Constraint Encoding

Implementation 1

The idea of Implementation 1 is based on lists. The constraint encoding pushes this idea further and the result is the following code:

```
1 P3 = PolynomialRing(P1, 't', l); P3
2 P3.inject_variables()
3 T = P3.gens()
4 f = []
5 #create f_j
6 for j in range(l):
7     f.append(prod((a[all+h] + T[h] + 1) for h in range (j+1,l)) * (a[all+j] + T[
      j]))
8 #create polynomial F
9 F = sum(f[j] * (T[j] + 1) for j in range(0,l) )
```

Listing 4.18. Weight Constraint Encoding

The computed weight which is stored in the last l entries of EQ, should now be compared with an integer t, therefore one has to define where the variables $t_1, \ldots, t_l$ live. Again, a polynomial ring *over* a polynomial ring is defined to represent polynomials in $a_1, \ldots, a_l, t_1, \ldots, t_l$. The variables $t_1, \ldots, t_l$ are components in the vector T. Moreover, an empty lists f is defined. This list will contain the polynomials f_j. In the following code snippet he polynomials f_j are created.

```
1 #create f\_j
2 for j in range(l):
3     f.append(prod((a[all+h] + T[h] + 1) for h in range (j+1,l)) * (a[all+j] + T[
      j]))
```

Listing 4.19. f_j

f_j gets stored in f. The implementation follows the equation defined in 4.17. The output would be the following.

```
1 [t0*t1*t2 + (a72 + 1)*t0*t1 + (a71 + 1)*t0*t2 + a70*t1*t2 + (a71*a72 + a71 + a72
      + 1)*t0 + (a70*a72 + a70)*t1 + (a70*a71 + a70)*t2 + a70*a71*a72 + a70*a71 +
      a70*a72 + a70, t1*t2 + (a72 + 1)*t1 + a71*t2 + a71*a72 + a71, t2 + a72]
```

Listing 4.20. Output of f_j

Lastly, F gets created.

```
1 #create polynomial F
2 F = sum(f[j] * (T[j] + 1) for j in range(0,l) )
```

Listing 4.21. Polynomial F

F returns 0 if the weight of vector v is smaller or equal to the proposed integer t, 1 otherwise. The output would look like the following:

```
#Output:
    t0^2*t1*t2 + (a72 + 1)*t0^2*t1 + (a71 + 1)*t0^2*t2 + (a70 + 1)*t0*t1*t2 + t1
    ^2*t2 + (a71*a72 + a71 + a72 + 1)*t0^2 + (a70*a72 + a70 + a72 + 1)*t0*t1 + (
    a72 + 1)*t1^2 + (a70*a71 + a70 + a71 + 1)*t0*t2 + (a70 + a71 + 1)*t1*t2 + t2
    ^2 + (a70*a71*a72 + a70*a71 + a70*a72 + a71*a72 + a70 + a71 + a72 + 1)*t0 +
    (a70*a72 + a71*a72 + a70 + a71 + a72 + 1)*t1 + (a70*a71 + a70 + a71 + a72 +
    1)*t2 + a70*a71*a72 + a70*a71 + a70*a72 + a71*a72 + a70 + a71 + a72
```

Listing 4.22. Output of F

If one evaluates for an integer $t = (1, 0, 0)$ the result would be the following:

```
#Output:
   a71*a72 + a71 + a72
```

The output is quite compact and concise. However, it can already been seen that solving this polynomial, a lot of variables have to be stored. Moreover, the solution depends on the last computation. This means, all the other $a_i j's$ have to be computed, before the resulting polynomial is given. The next implementation uses fewer variables which makes it easier to solve with SageMath.

Implementation 2

The second implementation is constructed by updating the vectors step by step. Recall that $L0$ contains the computed weight.

```
P3 = PolynomialRing(P2, 't', l); P3
P3.inject_variables()
T = vector(P3.gen(i) for i in range(l))
f = []

#create f_j
for j in range(l):
    f.append(prod((b[h] + T[h] + 1) for h in range (j+1,l)) * (b[j] + T[j]))

#create polynomial F
F = sum(f[j] * (T[j] + 1) for j in range(0,l) )
print("\n Polynomial F. F is 0 iff weight of a is smaller than t, 1 otherwise: \
    n ")
print(F)
```

Listing 4.23. Weight Constraint Encoding Implementation 2

Again, as in Implementation 1, a third polynomial ring $P3$ is generated over $P2$. Creating f_j for j in `range(j+1,l)` the product function of Sage can be used. This is actually the implementation of the equation defined in 4.17.

```
    #create f_j
    for j in range(l):
        f[j] = prod((L0[h] + T[h] + 1) for h in range (j+1,l)) * (L0[j] + T[j])
    print(f)
```

Listing 4.24. f_j

When f is returned, one can observe where $L0$ and t differ.

```
#Output:
    [t0*t1*t2 + (b2 + 1)*t0*t1 + (b1 + 1)*t0*t2 + b0*t1*t2 + (b1*b2 + b1 + b2 +
    1)*t0 + (b0*b2 + b0)*t1 + (b0*b1 + b0)*t2 + b0*b1*b2 + b0*b1 + b0*b2 + b0,
    t1*t2 + (b2 + 1)*t1 + b1*t2 + b1*b2 + b1, t2 + b2]
```

Listing 4.25. Output f_j

The Polynomial F is rather trivially constructed by using the `sum` function provided by Sage.

```
#create polynomial F
F = sum(f[j] * (T[j] + 1) for j in range(0,l) )
print(F)
```

Listing 4.26. Polynomial F

F returns 0 if the weight stored in $L0$ is smaller or equal to the proposed integer t, otherwise it will return 1. The output of F would look as follows:

```
#Output:
t0^2*t1*t2 + (b2 + 1)*t0^2*t1 + (b1 + 1)*t0^2*t2 + (b0 + 1)*t0*t1*t2 + t1^2*
t2 + (b1*b2 + b1 + b2 + 1)*t0^2 + (b0*b2 + b0 + b2 + 1)*t0*t1 + (b2 + 1)*t1
^2 + (b0*b1 + b0 + b1 + 1)*t0*t2 + (b0 + b1 + 1)*t1*t2 + t2^2 + (b0*b1*b2 +
b0*b1 + b0*b2 + b1*b2 + b0 + b1 + b2 + 1)*t0 + (b0*b2 + b1*b2 + b0 + b1 + b2
 + 1)*t1 + (b0*b1 + b0 + b1 + b2 + 1)*t2 + b0*b1*b2 + b0*b1 + b0*b2 + b1*b2
+ b0 + b1 + b2
```

Listing 4.27. Output F

If evaluating an integer $t = (1, 0, 0)$, the output would be:

```
#Output:
b1*b2 + b1 + b2
```

Note that although the size of the polynomials is way larger in Implementation 1 than in Implementation 2, fewer variables are needed.

4.3 Degree Reduction Formula for Polynomials

The computed polynomials are not quadratic but are of order l. With the following process these equations can be transformed into quadratic equations in polynomial time. Let $m = x_{i_1}x_{i_2}x_{i_3}\dots x_{i_l}$ be a monomial with $deg(m) = l$. A set of $l-2$ variables is introduced as follows:

$$\begin{cases} y_1 = x_{i_1}x_{i_2} \\ y_2 = y_1 x_{i_2} \\ y_{l-2} = y_{d-3}x_{i_{l-1}} \end{cases} \tag{4.25}$$

and m can be re-written as $m = y_{l-2}x_l$. A monomial m of degree l is substituted by a set of $l-1$ quadratic equations by introducing $l-2$ new variables. Applying the same argument to every monomial one obtains a system of quadratic equations as required. In the following example, the above construction is shown.

Example 1. Let $xyz + z = 0$ be an equation. The monomial has degree 3 and is therefore not quadratic. If we introduce a new variable w and set $w = xy$, then the equation reduces to $wz + z = 0$. So the degree of the equation is reduced by one, despite having to add an additional variable and equation. The resulting system is an instance of MQ. From the following theorem

it is known that the reduction can be computed in polynomial time.

Theorem 4.3.1. The memory space needed by the reduction is polynomial and bounded by $\mathcal{O}(n^2 log_2 n + m)$ where n and m are the dimensions of $H \in (\mathbb{F}_2)^{(n\times m)}$.

Proof. The proof of the above theorem can be looked up in [13]. □

4.4 MLD to MQ Reduction: Example

To demonstrate the correctness of the computations, observe the Hamming Code which encodes four data bits into seven bits by adding three parity bits. Exactly one error can be detected and corrected. Let the generator matrix $\mathcal{G}$ and parity check matrix $\mathcal{H}$ be the following:

$$\mathcal{G} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}, \mathcal{H} = \begin{pmatrix} 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{pmatrix} \tag{4.26}$$

If the message $c = (0,0,0,1,1,1,1)$ is sent, but the vector $z = (1,0,0,1,1,1,1)$ is received, the corresponding syndrome is: $\mathcal{H}z = s = (1,1,0)$. If the syndrome and the parity check matrix is added to one of the implementations, the result should be the error vector $e = (1,0,0,0,0,0,0)$ and one would have: $c + e = z$. The weight t is set to $(1,0,0)$. For the evaluation of the code, Implementation 2 is used:

```
H = Matrix([[1,1,0,1,1,0,0], [1,0,1,1,0,1,0], [0,1,1,1,0,0,1]])
s = vector([1,1,0])
numberOfVariables = H.ncols()
P.<v0,v1,v2,v3,v4,v5,v6,a0,a1,a2, b0,b1,b2> = PolynomialRing(GF(2));
P.inject_variables()
v = [P.gen(i) for i in range(7)]
b = [P.gen(i) for i in range(10,13)]
n = numberOfVariables
l = floor(log(n,2)) + 1
all = (n*l)

L0 = vector(P.gen(i) for i in range(7,10))
L1 = vector(P.gen(i) for i in range(7,10))


#product
for i in range(n):
    for j in range(l):
        if j == 0:
            L0[j] = L0[0]
        else:
            L0[j] = L0[j]*L0[j-1]
    #equations
    for j in range(l):
        if j == 0:
            L1[j] = L0[j] + v[i]
        else:
            L1[j] = L0[j]/L0[j-1] + L0[j-1]*v[i]
    L0 = L1[:]

for j in range(l):
    L0[j] = L0[j] + b[j]

t = vector([1,0,0])
f = []

#create f_j
for j in range(l):
    f.append(prod((b[h] + t[h] + 1) for h in range (j+1,l)) * (b[j] + t[j]))

#create polynomial F
F = sum((f[j] * (t[j] + 1)) for j in range(0,l) )


S = (H*vector(P.gen(i) for i in range(numberOfVariables))-s).list()
```

```
46 S = S + L0.list()
47 S = S+ [F]
48 for i in range(13):
49     S.append(P.gen(i)^2 + P.gen(i))
50
51 print(S)
52 I = ideal(S)
53 for sol in I.variety():
54     print(sol[a0],sol[a1],sol[a2],sol[v0],sol[v1],sol[v2],sol[v3],sol[v4],sol[v5
       ],sol[v6],sol[b0],sol[b1],sol[b2])
```

The code is completed with the construction of the system of equations $\mathcal{S}$. Moreover, by adding the `for`-Loop in Line 48, one rests assure that every variable can only hold the values 0 or 1. This further ensures, that the set of solutions of the system will be finite. Now consider an ideal I generated by $\mathcal{S}$. The command `variety` will compute all the possible solutions of the system. In this specific case, the solutions of the system will be:

```
1  0 1 0 0 1 1 1 1 1 1 0 0 0
2  1 1 0 1 1 1 0 1 1 0 0 0 0
3  1 1 0 1 1 1 1 0 0 1 0 0 0
4  1 1 0 1 0 0 1 1 1 1 0 0 0
5  0 0 1 1 0 1 1 1 0 0 0 0 0
6  0 0 1 1 1 0 1 0 1 0 0 0 0
7  0 0 1 1 1 0 0 1 0 1 0 0 0
8  0 0 1 1 0 1 0 0 1 1 0 0 0
9  1 0 1 0 1 0 1 1 0 0 0 0 0
10 1 0 1 0 0 1 1 0 1 0 0 0 0
11 1 0 1 0 0 1 0 1 0 1 0 0 0
12 1 0 1 0 1 0 0 0 1 1 0 0 0
13 0 1 1 0 1 1 0 0 0 0 0 0 0
14 0 1 1 0 0 0 0 1 1 0 0 0 0
15 0 1 1 0 0 0 1 0 0 1 0 0 0
16 1 1 1 1 0 0 0 0 0 0 0 0 0
17 0 0 0 1 0 0 0 0 0 0 1 0 0
18 1 1 0 0 1 1 1 1 1 1 1 0 0
19 0 0 1 1 1 1 0 1 1 0 1 0 0
20 0 0 1 1 1 1 1 0 0 1 1 0 0
21 0 0 1 1 0 0 1 1 1 1 1 0 0
22 1 0 1 1 0 1 1 1 0 0 1 0 0
23 1 0 1 1 1 0 1 0 1 0 1 0 0
24 1 0 1 1 1 0 0 1 0 1 1 0 0
25 1 0 1 1 0 1 0 0 1 1 1 0 0
26 0 1 1 0 1 0 1 1 0 0 1 0 0
27 0 1 1 0 0 1 1 0 1 0 1 0 0
28 0 1 1 0 0 1 0 1 0 1 1 0 0
29 0 1 1 0 1 0 0 0 1 1 1 0 0
30 1 1 1 0 1 1 0 0 0 0 1 0 0
31 1 1 1 0 0 0 0 1 1 0 1 0 0
32 1 1 1 0 0 0 1 0 0 1 1 0 0
```

Listing 4.28. Solutions of System of Equations

The first three digits are the possible solutions for a_{0j}. Because the variables $a_0 = a_1 = a_2$ are set to zero, the only possible solution is found in line 17. The next seven digits is the error vector $e = (1,0,0,0,0,0,0)$ and the last seven digits is the computed weight stored in b_0, b_1, b_2. It is known that the error vector has to be $e = (1,0,0,0,0,0,0)$, so the correct solution is returned. If `S = S + [a0,a1,a2]` is added to the system of equations, only the correct error vector gets returned, because all the equations are set to zero. Hence, the possible solution for $a_0 = a_1 = a_2 = 0$ is directly filtered. Again, let $c = (0,0,0,1,1,1,1)$ but the vector $z = (1,0,0,1,1,1,1)$ is received. The syndrome is $s = (1,1,0)$. Then the solutions is:

```
1 0 0 0 1 0 0 0 0 0 0 1 0 0
```

The correct error vector is $v = (1, 0, 0, 0, 0, 0, 0)$ - which is exactly what the found output is. This means, the implementation for a reduction from MLD to MQ works.

4.5 MQ to MLD

The previous sections have shown a way to reduce an instance of MLD to an instance of MQ. If the system of equations can be solved, the error vector v is returned, which, in fact, solves the instance of the MLD problem. The following code snippets directly returns the error vector after the computing the system of equations `S`:

```
I = ideal(S)
for sol in I.variety():
    print(sol[v0],sol[v1],sol[v2],sol[v3],sol[v4],sol[v5],sol[v6])
```

Listing 4.29. error vector v

So the error vector v is found and it holds that $Hv^\top = s^\top$ and v is of weight at most t. This solves the MLD problem defined in 2.2.1. With this, the reduction is completed.

Chapter 5

Conclusion

In this thesis a reduction between MLD and MQ was implemented. There is also an efficient way to construct a reduction from MQ to MLD [13]. If the resulting equations of the reduction from MLD to MQ can be decoded, each MLD instance may be solved. The implementation of the reduction does return a solution for the MLD problem, by reducing it to an instance of the MQ problem - and if an error vector v is found, the MQ instance is solvable as well. Furthermore, the existing reduction proves that solving an instance of MQ is not "harder" than solving an instance of MQ. In this thesis two different approaches for the reduction were investigated. It would also have been interesting to look at whether Implementation 1 or Implementation 2 is more efficient in computing the reduction. Implementation 1 returns a higher number of variables, but less complex equations, whereas Implementation 2 returns more complex equations, but only uses l variables $(a_1^0, \ldots, a_l^0)$. Bench-marking the computations may provide insight into which implementation is more applicable dependant on efficiency, memory usage and/or computation time. In conclusion, a reduction between the Maximum Likelihood Decoding Problem and the Multivariate Quadratic System Problem was found and implemented. When evaluating a specific example, up to one error was found and corrected. Therefore, the computations may solve instances of MLD and also of MQ and return correct results for the error vector.

Bibliography

[1] A. M. Turing, "On computable numbers, with an application to the Entscheidungsproblem," *Proceedings of the London Mathematical Society*, vol. 2, no. 42, pp. 230–265, 1936.

[2] E. R. Berlekamp, R. J. McEliece, and H. C. A. van Tilborg, "On the inherent intractability of certain coding problems (corresp.)," *IEEE Trans. Inf. Theory*, vol. 24, no. 3, pp. 384–386, 1978.

[3] R. J. McEliece, "A Public-Key Cryptosystem Based On Algebraic Coding Theory," *Deep Space Network Progress Report*, vol. 44, pp. 114–116, Jan. 1978.

[4] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[5] H. Niederreiter, "Knapsack-type cryptosystems and algebraic coding theory," *Prob. Contr. Inform. Theory*, vol. 15, no. 2, pp. 157–166, 1986.

[6] M. Sipser, *Introduction to the theory of computation*. PWS Publishing Company, 1997.

[7] N. T. Courtois, "Generic attacks and the security of quartz," in *Public Key Cryptography - PKC 2003, 6th International Workshop on Theory and Practice in Public Key Cryptography, Miami, FL, USA, January 6-8, 2003, Proceedings*, ser. Lecture Notes in Computer Science, vol. 2567, Springer, 2003, pp. 351–364.

[8] J. Ding and D. Schmidt, "Rainbow, a new multivariable polynomial signature scheme," in *Applied Cryptography and Network Security*, J. Ioannidis, A. Keromytis, and M. Yung, Eds., Springer Berlin Heidelberg, 2005, pp. 164–175.

[9] R. M. Roth, *Introduction to coding theory*. Cambridge University Press, 2006.

[10] J. Ding and B.-Y. Yang, "Multivariate public key cryptography," in *Post-Quantum Cryptography*, D. J. Bernstein, J. Buchmann, and E. Dahmen, Eds. Springer Berlin Heidelberg, 2009, pp. 193–241.

[11] R. Overbeck and N. Sendrier, "Code-based cryptography," in *Post-Quantum Cryptography*, D. J. Bernstein, J. Buchmann, and E. Dahmen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 95–145.

[12] G. Fischer, "1 grundbegriffe," in *Lineare Algebra: Eine Einführung für Studienanfänger*. Springer Fachmedien Wiesbaden, 2014, pp. 32–105.

[13] A. P. Alessio Meneghetti and M. Sala, *A reduction between MQ and MLD*, Unpublished manuscript, 2020.

[14] S. Hogan, "A gentle introduction to computational complexity theory, and a little bit more," Sep. 2020.

Erklärung

Erklärung gemäss Art. 30 RSL Phil.-nat. 18

Ich erkläre hiermit, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäss aus Quellen entnommen wurden, habe ich als solche gekennzeichnet. Mir ist bekannt, dass andernfalls der Senat gemäss Artikel 36 Absatz 1 Buchstabe r des Gesetzes vom 5. September 1996 über die Universität zum Entzug des auf Grund dieser Arbeit verliehenen Titels berechtigt ist.

Für die Zwecke der Begutachtung und der Überprüfung der Einhaltung der Selbständigkeitserklärung bzw. der Reglemente betreffend Plagiate erteile ich der Universität Bern das Recht, die dazu erforderlichen Personendaten zu bearbeiten und Nutzungshandlungen vorzunehmen, insbesondere die schriftliche Arbeit zu vervielfältigen und dauerhaft in einer Datenbank zu speichern sowie diese zur Überprüfung von Arbeiten Dritter zu verwenden oder hierzu zur Verfügung zu stellen.

Ort/Datum

Unterschrift