FAST GENERATION AND COVERING RADIUS OF REED-MULLER CODES

by

Argyrios Alexopoulos

December 2009

Thesis Co-Advisors: Pantelimon Stanica
                       Jon T. Butler

Approved for public release; distribution is unlimited
**Abstract**

Reed-Muller codes are known to be some of the oldest, simplest and most elegant error correcting codes. Reed-Muller codes were invented in 1954 by D. E. Muller and I. S. Reed, and were an important extension of the Hamming and Golay codes because they gave more flexibility in the size of the codeword and the number of errors that could be correct.

The covering radius of these codes, as well as the fast construction of covering codes, is the main subject of this thesis. The covering radius problem is important because of the problem of constructing codes having a specified length and dimension. Codes with a reasonably small covering radius are highly desired in digital communication environments.

In addition, a new algorithm is presented that allows the use of a compact way to represent Reed-Muller codes. Using this algorithm, a new method for fast, less complex, and memory efficient generation of 1st and 2nd order Reed-Muller codes and their hardware implementation is possible. It also allows the fast construction of a new subcode class of 2nd order Reed-Muller codes with good properties. Finally, by reversing this algorithm, we introduce a code compression method, and at the same time a fast, efficient, and promising error-correction process.
THIS PAGE INTENTIONALLY LEFT BLANK
ABSTRACT

Reed-Muller codes are known to be some of the oldest, simplest and most elegant error correcting codes. Reed-Muller codes were invented in 1954 by D. E. Muller and I. S. Reed, and were an important extension of the Hamming and Golay codes because they gave more flexibility in the size of the codeword and the number of errors that could be correct.

The covering radius of these codes, as well as the fast construction of covering codes, is the main subject of this thesis. The covering radius problem is important because of the problem of constructing codes having a specified length and dimension. Codes with a reasonably small covering radius are highly desired in digital communication environments.

In addition, a new algorithm is presented that allows the use of a compact way to represent Reed-Muller codes. Using this algorithm, a new method for fast, less complex, and memory efficient generation of 1st and 2nd order Reed - Muller codes and their hardware implementation is possible. It is also allows the fast construction of a new subcode class of 2nd order Reed-Muller codes with good properties. Finally, by reversing this algorithm, we introduce a code compression method, and at the same time a fast, efficient, and promising error-correction process.
# TABLE OF CONTENTS

## I. INTRODUCTION
- A. THESIS OBJECTIVE ................................................................. 1
- B. THESIS OUTLINE ................................................................. 2

## II. BACKGROUND
- A. DEFINITIONS ................................................................. 3
  1. Groups.................................................................................. 3
     a. Closure ........................................................................... 3
     b. Associativity ............................................................... 3
     c. Identity .......................................................................... 3
     d. Invertibility ................................................................. 3
     e. Commutativity .............................................................. 3
  2. Fields .................................................................................. 3
     a. Group Under Addition .................................................. 4
     b. Group Under Multiplication ......................................... 4
     c. Distributive Law ......................................................... 4
  3. Vector Space ....................................................................... 4
     a. Addition .......................................................................... 6
     b. Vector Intersection ..................................................... 6
     c. Dot Product ................................................................... 6
- B. REVIEW OF BOOLEAN FUNCTIONS .................................... 6
- C. AFFINE BOOLEAN FUNCTIONS ........................................ 9
- D. NONLINEARITY AND BENT FUNCTIONS ......................... 10
- E. HAMMING DISTANCE AND HAMMING WEIGHT .............. 10
  1. Definition ........................................................................... 10
  2. Definition ........................................................................... 11
- F. ERROR DETECTING AND CORRECTING CAPABILITIES OF CODES ......................................................... 11
  1. Definition ........................................................................... 11
  2. Definition ........................................................................... 11
- G. CHAPTER SUMMARY ........................................................ 12

## III. REED-MULLER CODES
- A. DEFINING REED-MULLER CODES ..................................... 13
- B. APPLICATIONS OF REED-MULLER CODES ..................... 14
- C. GENERATION/ENCODING METHODS ................................. 16
  1. Generation Methods ....................................................... 16
     a. Using Boolean Polynomials ........................................ 16
     b. Example R(1,3) ........................................................... 16
     c. Using Direct Sum Construction ................................. 17
     d. Using (u,u+v)-Construction ...................................... 18
  2. Encoding Methods ............................................................ 18
     a. Example R(1,3) ........................................................... 19
b. Example $R(2,3)$ ................................................................. 19

c. Example $R(3,3)$ ................................................................. 19

d. Example $R(2,4)$ ................................................................. 19

e. Example Encoding with $R(1,3)$ ......................................... 20

f. Example Encoding with $R(2,4)$ ......................................... 20

3. Decoding Methods ................................................................. 21

a. Decoding Algorithm ......................................................... 21

b. Example of Decoding Using $R(1,3)$ ................................... 22

D. CHAPTER SUMMARY ................................................................. 23

IV. COVERING RADIUS ........................................................................ 25

A. INTRODUCTION ........................................................................ 25

B. METHODS OF COMPUTATIONS OF COVERING RADIUS .......... 27

1. 1st Method Using Translate .................................................. 27

2. 2nd Method of Using Direct Sum of Codes ............................. 29

a. Definition of Norm of a Code $C$ ........................................... 29

b. Definition of a Normal Code .............................................. 29

c. Example $R_1 + R_2$ .......................................................... 30

3. 3rd Method Using Bounds ..................................................... 30

4. 4th Method Using Norm ....................................................... 31

C. COVERING RADIUS FOR 1ST ORDER REED-MULLER CODES ... 32

D. COVERING RADIUS FOR 2ND ORDER REED-MULLER CODES ... 32

E. COVERING RADIUS FOR $R^TH$ ORDER REED-MULLER CODES ... 33

F. CHAPTER SUMMARY ................................................................. 33

V. FAST ALGORITHM OF GENERATION OF 1ST—2ND ORDER REED- MULLER CODES, LINEAR SUBCODES WITH GOOD PROPERTIES, AND THE “REVERSE” ALGORITHM ......................................................... 35

A. FAST GENERATION OF 1ST ORDER REED-MULLER CODES ....... 35

1. New Algorithm for Fast Generating 1st Order RM Codes ........ 36

2. Example R(1,3) ................................................................. 36

3. Example R(1,4) ................................................................. 37

B. HARDWARE IMPLEMENTATION OF ALGORITHM .................. 39

C. FAST GENERATION OF 2ND ORDER REED-MULLER CODES ...... 41

D. FAST GENERATION OF LINEAR SUBCODES WITH GOOD PROPERTIES ......................................................... 42

1. Algorithm ........................................................................... 43

2. Example $R(2,3)$ Subcode ................................................. 43

3. Example $R(2,5)$ Subcode ................................................. 44

4. Theorem ........................................................................... 46

a. Example of Calculating the ANF of a Function ..................... 46

b. Example of Calculating the Coefficient Vector .................... 47

c. Proof of Theorem ......................................................... 48

E. THE DECODING “REVERSE” ALGORITHM ............................ 48

1. Conjecture ........................................................................... 49

2. Algorithm ........................................................................... 49
**LIST OF FIGURES**

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>3-bit binary cube for finding Hamming distance (From [2])</td>
<td>10</td>
</tr>
<tr>
<td>Figure 2</td>
<td>4-bit binary hypercube for finding Hamming distance (From [2])</td>
<td>11</td>
</tr>
<tr>
<td>Figure 3</td>
<td>Picture Elements.</td>
<td>15</td>
</tr>
<tr>
<td>Figure 4</td>
<td>Sphere of radius ( \rho ).</td>
<td>25</td>
</tr>
<tr>
<td>Figure 5</td>
<td>Covering radius ( \rho ).</td>
<td>26</td>
</tr>
<tr>
<td>Figure 6</td>
<td>Hardware implementation of algorithm ( n=3 )</td>
<td>40</td>
</tr>
</tbody>
</table>
LIST OF TABLES

Table 1. Addition in $F_2$ ........................................................................................................... 4
Table 2. Multiplication in $F_2$ ........................................................................................... 4
Table 3. Truth Table of a Boolean Function ...................................................................... 7
Table 4. Truth Table of an Affine Boolean Function ...................................................... 9
Table 5. RM(1,3) codewords ......................................................................................... 17
Table 6. 1st method of Covering Radius computation ................................................... 28
Table 7. 3rd method of Covering Radius computation .................................................. 31
Table 8. Fast Generation of $R(1,3)$.................................................................................. 37
Table 9. Fast Generation of $R(1,4)$.................................................................................. 38
Table 10. Fast Generation of $R(2,3)$.............................................................................. 42
Table 11. Fast Generation of a $R(2,3)$ subcode............................................................ 44
Table 12. Fast Generation of $R(2,5)$ subcode............................................................ 45
EXECUTIVE SUMMARY

Error-Correcting codes play a vital role in every real digital communication environment and storage process. Reed-Muller codes are among the oldest, simplest and most elegant error-correcting codes. When information is sent through a network over long distances or through a variety of channels, where errors might occur in the transition, error-correcting codes, like Reed-Muller codes, can correct these errors. This correction process provides our network with an improvement in throughput and efficiency. Therefore, the efficient use of these codes is more than a critical issue.

A contribution of this thesis is a new way of fast generation of 1st and 2nd order Reed-Muller codes and a category of 2nd order subcodes. In addition, this new algorithm allows a compact way to represent 1st and 2nd order Reed-Muller codes.

This expansion algorithm is appropriate where the fast, real-time generation of low order Reed-Muller codes needed. Using this highly compressed form of codewords, we can quickly expand to any full codeword. In this thesis, we also demonstrate the hardware implementation for this algorithm.

It is also shown, that using just eight blocks of 4-bits, all 1st order Reed-Muller codes can be quickly generated. In addition, for 2nd order Reed-Muller codes, a new concatenation method using all sixteen possible 4-bit combinations is presented. Finally, using eight 4-bits words, we can quickly construct, a new category of subcodes of 2nd order Reed-Muller code with minimum distance $d=8$ and some other good properties.

Additionally, it is proven in this thesis, that the format of the Algebraic Normal Form of our fast construction of 2nd order Reed-Muller subcodes is $affine + x_{n-1}x_n$. Combining this property with the low distance of these subcodes, makes them worthy for further investigation concerning their performance.

In addition, by reversing the new algorithm, we demonstrate a new efficient way to correct errors occurring in this word. This is equivalent to compressing the received word. The “reverse” algorithm applies to cases of storage processes and to communication-oriented applications where Automatic Response Request (ARQ) is used.
Furthermore, the state of the art of the covering radius problem for Reed-Muller codes is presented in this thesis. This has been the subject of investigation for many researchers in the area, and a complete resolution of the problem still eludes us. Some recently found results of estimates of covering radius of Reed-Muller codes are summarized and presented. Some of the methods of computations, even without using the help of computers are also presented. In addition, the main properties of Reed-Muller codes are analyzed.

The covering radius problem is very important since it gives insight into the practical problem of constructing codes having a specified length and dimension.

Based on the analysis of this thesis, we conclude that the proposed methods of fast and memory efficient low order Reed-Muller codes, as well as some category of subcodes of 2\textsuperscript{nd} order Reed-Muller codes, is quite challenging and promising.
ACKNOWLEDGMENTS

I dedicate this work to my wife, Evgenia, and my children, Dimitrios and Konstantina, for their continuous love and support.

In addition, I would like to thank Professor Pantelimon Stanica and Professor Jon T. Butler for their guidance and patience during the work in performing this investigation.
I. INTRODUCTION

A. THESIS OBJECTIVE

A binary code $C$ of length $n$ is a nonempty subset of the set of all binary $n$-tuples. The (Hamming) distance between two codewords is the number of bits in which they differ. The covering radius of a code $C$ is the smallest integer $R$ such that every binary vector of length $n$ is within distance $R$ from at least one codeword. In other words, the space of all binary $n$-tuples is completely covered by spheres of radii $R$ having centers at the codewords of our code $C$.

The covering radius is one of the most important properties of error correcting codes. It will be clarified throughout this thesis. We seek codes having a specified length and dimension with reasonably small covering radius; in this way, no vector of the space is very far from a nearest codeword.

It is worth mentioning that covering radius is a basic geometric parameter of a code. Topics that are currently under research by the coding community are the following:

1. Given the length and dimension of a linear code, it should be determine what the covering radius is.
2. Construct efficient codes that have small covering radius.
3. Develop computational methods to determine the covering radius of well-known error-correction codes.

Specifically, Reed-Muller codes are an extremely interesting class of error-correction codes, and therefore, many researchers have studied Reed-Muller codes. Nevertheless, due to the complexity of computations methods, overall knowledge is still quite limited. We will focus on some of these methods and point out all published results of covering radius of $1^{\text{st}}, 2^{\text{nd}}$ and $k$-th order Reed-Muller codes.

We survey the main properties of the Reed-Muller codes, investigate previous methods for the computation of covering radius, and propose a fast, less complex, and memory efficient algorithm to derive $1^{\text{st}}$ and $2^{\text{nd}}$ order Reed-Muller codes, as well as a
subcode of 2nd order Reed-Muller code with good properties. This new algorithm allows the use of a compact way to represent Reed-Muller codes.

Reversing the new algorithm, in other words, compressing the codewords of low order Reed-Muller codes and of a new subcode, we introduce an efficient way to correct errors occurring in any codeword of these codes. The hardware implementation of expansion algorithm is presented, and analyzed.

B. THESIS OUTLINE

This thesis is organized as follows: we start with the introduction, background (Chapter II) and four additional chapters. Chapter III contains a detailed analysis of Reed-Muller codes, some applications of these codes, and generation/encoding/decoding methods. In Chapter IV, the covering radius is defined and its importance in error correction is discussed. Some important methods of computations are discussed. Further, in this chapter, some existing covering radius results concerning 1st, 2nd and k-th order Reed-Muller codes are presented. In Chapter V, we develop a new algorithm for fast generation of 1st – 2nd order Reed-Muller codes, and a new construction of a subcode is analyzed. The hardware implementation of this algorithm is also presented and analyzed. In addition, a “reverse” of this new algorithm is presented and evaluated. In Chapter VI, the conclusions based on the observations obtained from the analysis in the previous chapters are presented, as well as proposed future work.
II. BACKGROUND

In this chapter, some background knowledge and concepts for the analysis of Reed-Muller codes and their subcodes are introduced.

A. DEFINITIONS

1. Groups

A group, denoted by \([G] = (G,\ast)\) is a set of elements \(G\) combined with a binary operation \(\ast\) on \(G\), satisfying the following conditions:

   a. Closure
   \[\forall a, b \in G; a \ast b = c \in G\]

   b. Associativity
   \[\forall a, b, c \in G; a \ast (b \ast c) = (a \ast b) \ast c\]

   c. Identity
   \[\exists e \in G \mid \forall a \in G; e \ast a = a \ast e = a\]

   d. Invertibility
   \[\forall a \in G, \exists a^{-1} \in G \mid a \ast a^{-1} = a^{-1} \ast a = e\]

   Groups that also satisfy the following commutative property are referred to as commutative or Abelian groups.

   e. Commutativity
   \[\forall a, b \in G; a \ast b = b \ast a\]

2. Fields

A field, denoted by \([F] = (F, +, \ast)\), is a set of elements \(F\) combined with two binary operations \(+\) and \(\ast\) on \(F\), satisfying the following conditions:
a. **Group Under Addition**

\((F, +)\) is an Abelian group with identity 0.

b. **Group Under Multiplication**

\((F - \{0\}, \cdot)\) is an Abelian group with identity 1.

c. **Distributive Law**

\[ \forall a, b, c \in F; a \cdot (b + c) = (a \cdot b) + (a \cdot c) \]

In this thesis, all manipulations will be on the two-element (binary) field, \(F_2 = \{0, 1\}\) in which the usual operations of addition and multiplication modulo 2 hold:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 1. **Addition in** \(F_2\)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 2. **Multiplication in** \(F_2\)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3. **Vector Space**

A *vector space* over a field \(F\) is a non-empty set \(V\) together with two binary operations:
Addition, denoted by $+$.  

Scalar multiplication, denoted by juxtaposition, is a function from $F \times V$ to $V$; that is the scalar product of $a \in F$ and $x \in V$ is written as $ax$.

Furthermore, these two operations satisfy the following conditions:

**Closure under vector addition,**

$$\forall u, v \in V; u + v = w \in V$$

**Closure under scalar multiplication,**

$$\forall u \in V, \forall a \in F; au = v \in V$$

**Associative law for vector addition,**

$$\forall u, v, w \in V; \ u + (v + w) = (u + v) + w$$

**Commutative law for vector addition,**

$$\forall u, v \in V; u + v = v + u$$

**Identity element in addition,**

$$\exists 0 \in V \mid \forall u \in V; u + 0 = u$$

**Additive inverse,**

$$\forall u \in V, \exists (-u) \in V; u + (-u) = (-u) + u = 0$$

**Distributive law for scalar multiplication over vector addition,**

$$\forall u, v \in V, \forall a \in F; a(u + v) = au + av$$

**Distributive law for vector multiplication over scalar addition,**

$$\forall u \in V, \forall a, b \in F; (a + b)u = au + bv$$

**Associative law for scalar multiplication with a vector,**

$$\forall u \in V, \forall a, b \in F; (ab)u = a(bu)$$

**Identity element in vector multiplication,**

$$\exists 1 \in V; \forall u \in V, 1u = u$$

The vector spaces $V = F_2^m$ used in this thesis consist of binary strings of length $2^m$, where $m$ is a positive integer, with the usual bitwise operations, described below. The codewords of the Reed-Muller codes and other linear subcodes are subspaces of such a vectors space $V$.

Vectors in such spaces can be manipulated by three main operations.
a. **Addition**

For two vectors \( x = (x_1, x_2, \ldots, x_n) \) and \( y = (y_1, y_2, \ldots, y_n) \), addition is defined by, \( x + y = (x_1 + y_1, x_2 + y_2, \ldots, x_n + y_n) \) where each \( x_i \) or \( y_i \) is either 1 or 0. The **complement** \( \overline{x} \) of a vector \( x \) is the vector equal to \((1 \ 1 \ 1 \ldots 1) + x\). An example of the complement of a vector is: \((0 \ 0 \ 0 \ 1 \ 1 \ 1) + (1 \ 1 \ 1 \ 1 \ 1 \ 1) = (1 \ 1 \ 1 \ 0 \ 0 \ 0)\)

b. **Vector Intersection**

\( x \ast y = (x_1 \ast y_1, x_2 \ast y_2, \ldots, x_n \ast y_n) \), where each \( x_i \) and \( y_i \) is either 1 or 0.

The multiplication of a vector \( x \) by a constant \( \alpha \in \mathbb{F}_2 \) is defined by \( \alpha \ast x = (\alpha \ast x_1, \alpha \ast x_2, \ldots, \alpha \ast x_n) \). An example is \( 0 \ast (11001) = (00000) \).

c. **Dot Product**

The dot product of \( x \) and \( y \) is \( x \cdot y = x_1 \ast y_1 + x_2 \ast y_2 + \ldots + x_n \ast y_n \).

It is clear that addition, vector intersection and dot product require vectors with the same number of coordinates.

B. **REVIEW OF BOOLEAN FUNCTIONS**

A Boolean function of \( m \) variables \( (x_1, x_2, \ldots, x_m) \) is a function \( f(x_1, x_2, \ldots, x_m) \) from \( \mathbb{F}_2^m \) to \( \mathbb{F}_2 \), where \( \mathbb{F}_2 = \{0,1\} \). This kind of function can be completely described by its truth table, which is simply the sequence of its outputs, where the input is ordered lexicographically. Precisely, we order \( \mathbb{F}_2^m \) as \( \{v_1 = (0, 0, \ldots, 0), \ v_2 = (1, 0, \ldots, 1) \ldots, (0, 1, \ldots, 1)\} \); the truth table of \( f \) is the sequence \( f(v_1), f(v_2), \ldots \). Table 3 specifies a Boolean function of four variables.
Table 3. Truth Table of a Boolean Function

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>$x_4$</th>
<th>$f$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

From the right most column of this table (beginning from top), we get the binary string (truth table) $1110100001110001$ of length 16. For $m$-variable functions, this string has length $2^m$.

The two constant Boolean functions are $(x_1,x_2,\ldots,x_m)=(1,1,\ldots,1)$ and $(x_1,x_2,\ldots,x_m)=(0,0,\ldots,0)$. In this thesis, two logical operations are used on Boolean functions: conjunction (that corresponds to multiplication in $F_2$) and exclusive OR, or xor (that corresponds to addition in $F_2$). Consequently, the string versions of these operations are given by:

$(x_1,x_2,\ldots,x_m)$ conjunction $(y_1,y_2,\ldots,y_m) = (x_1,x_2,\ldots,x_m) \cap (y_1,y_2,\ldots,y_m)$ and

$(x_1,x_2,\ldots,x_m)$ exclusive OR $(y_1,y_2,\ldots,y_m) = (x_1,x_2,\ldots,x_m) \cup (y_1,y_2,\ldots,y_m)$.
It is obvious that, under the exclusive OR operation, the set of Boolean functions of \( m \) variables forms a vector space over \( F_2 \), of size \( 2^{2^m} \).

In addition, a Boolean monomial with variables \( x_1, x_2, \ldots, x_m \) is an expression of the form \( p = x_{i_1} x_{i_2} \ldots x_{i_k} \). The reduced form of \( p \) is obtained using the rule: \( x_{i_k} = x_{i_k} \) until the factors become distinct. The degree of \( p \) is the number of variables in the reduced version of \( p \).

An example of a Boolean polynomial in reduced form of degree three is \( p' = x_1 + x_2 + x_3 + x_1 x_2 x_3 \).

On the other hand, a Boolean polynomial is a linear combination of Boolean monomials, with coefficients in \( F_2 \). A reduced polynomial is obtained using the rule: \( a + a = 0 \), until all the monomials become distinct.

Since there are \( \binom{m}{k} \) distinct Boolean monomials of degree \( k \) on \( m \) variables, the total number of distinct Boolean monomials is \( 2^m \), and, therefore, the total number of distinct Boolean polynomials in \( m \) variables is \( 2^{2^m} \).

At this point, we need to associate a Boolean monomial in \( m \) variables to a vector with \( 2^m \) elements. The degree-zero monomial is 1, and the degree-one monomials are \( x_1, x_2, \ldots, \) and \( x_m \). First, we define the vectors associated with these monomials. The vector associated with the monomial 1 is simply a vector of length \( 2^m \), whose components are all 1. So, in a space of size \( 2^4 \), the vector associated with 1 is (1111111111111111). The vector associated with the monomial \( x_i \) is \( 2^{m-1} \) 1's, followed by \( 2^{m-1} \) 0's. The vector associated with the monomial \( x_2 \) has \( 2^{m-2} \) 1's, followed by \( 2^{m-2} \) 0's, then another \( 2^{m-2} \) 1’s, followed by another \( 2^{m-2} \) 0’s.

In general, the vector associated with a monomial \( x_i \) is a pattern of \( 2^{m-i} \) ones followed by \( 2^{m-i} \) zeros, repeated until \( 2^n \) values have been defined. For example, in a space of size \( 2^4 \), the vector associated with \( x_4 \) is (10101010101010).
C. AFFINE BOOLEAN FUNCTIONS

An affine Boolean function of \( m \) variables \((x_1, x_2, \ldots, x_m)\) is a function
\[
f(x_1, x_2, \ldots, x_m) = f_0 + \sum_{1 \leq i \leq m} f_i x_i
\]
from \( F_2^m \) to \( F_2 \), where the coefficients \( f_i \) belong to \( F_2 = \{0,1\} \). The set of all \( n \)-variable affine functions is denoted by \( A_n \). As we mentioned previously, in Table 4, the truth tables of every 3-variable affine function is shown.

Table 4. Truth Table of an Affine Boolean Function

<table>
<thead>
<tr>
<th>Affine Function</th>
<th>Truth Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00000000</td>
</tr>
<tr>
<td>( x_1 )</td>
<td>00001111</td>
</tr>
<tr>
<td>( x_2 )</td>
<td>00110011</td>
</tr>
<tr>
<td>( x_3 )</td>
<td>01010101</td>
</tr>
<tr>
<td>( x_1 + x_2 )</td>
<td>00111100</td>
</tr>
<tr>
<td>( x_1 + x_3 )</td>
<td>01011010</td>
</tr>
<tr>
<td>( x_2 + x_3 )</td>
<td>01100110</td>
</tr>
<tr>
<td>( x_1 + x_2 + x_3 )</td>
<td>01101001</td>
</tr>
<tr>
<td>1</td>
<td>11111111</td>
</tr>
<tr>
<td>1 + ( x_1 )</td>
<td>11100000</td>
</tr>
<tr>
<td>1 + ( x_2 )</td>
<td>11001100</td>
</tr>
<tr>
<td>1 + ( x_3 )</td>
<td>10101010</td>
</tr>
<tr>
<td>1 + ( x_1 + x_2 )</td>
<td>11000111</td>
</tr>
<tr>
<td>1 + ( x_1 + x_3 )</td>
<td>10100101</td>
</tr>
<tr>
<td>1 + ( x_2 + x_3 )</td>
<td>10011001</td>
</tr>
<tr>
<td>1 + ( x_1 + x_2 + x_3 )</td>
<td>10010110</td>
</tr>
</tbody>
</table>
In Chapter IV the importance of affine Boolean functions in conjunction with Reed-Muller codes is clarified.

D. NONLINEARITY AND BENT FUNCTIONS

The nonlinearity of a Boolean function $f$ is defined as $N(f) = \min\{d(f, \beta) | \beta \in A_n\}$, where $d$ (Hamming distance) is the number of different coordinates of vectors in which $f$ differs from $\beta$. It is known (see [1]) that the nonlinearity is upper bounded by: $N(f) \leq 2^{n-1} - 2^{\frac{n-1}{2}}$. The concept of nonlinearity is a very important cryptographic property.

The Boolean functions on an even $n$ number of variables, whose nonlinearity is maximum, are called bent functions. The importance of bent functions is due to their correspondence to the words of length $2^n$ whose distance to the 1st order Reed-Muller codes is equal to the covering radius of this code. Bent functions play a significant role in cryptographic environments.

E. HAMMING DISTANCE AND HAMMING WEIGHT

1. Definition

Let $x = (x_1, x_2, ..., x_n)$ and $y = (y_1, y_2, ..., y_n)$ be two vectors in $F^n$. The Hamming distance $d(x, y)$, between $x$ and $y$ is the number of coordinate places in which they differ. For a fixed length $n$, the Hamming distance is a metric on the vector space of the words of that length. For words of length 3 and 4, Figures 1 and 2 can be used for calculating this Hamming distance.

$$d(110, 001) = 3$$

Figure 1. 3-bit binary cube for finding Hamming distance (From [2]).
Figure 2. 4-bit binary hypercube for finding Hamming distance (From [2]).

In this thesis, we will refer to the *Hamming distance* as *distance* since it is nonnegative, symmetric, and triangular:

- \( d(x, y) \geq 0 \) and \( d(x, y) = 0 \) iff \( x = y \)
- \( d(x, y) = d(y, x) \) for all \( x, y \) in \( F^n \)
- \( d(x, y) \leq d(x, z) + d(z, y) \) for all \( x, y, z \) in \( F^n \)

2. **Definition**

Hamming weight of a binary word \( w \) is the number of "1" bits in \( w \). For example \( \text{wt}(1100101110) = 7 \).

F. **ERROR DETECTING AND CORRECTING CAPABILITIES OF CODES**

Having defined the Hamming distance of two vectors, we can now clearly describe the distance of a code \( C \) as the minimum distance between any two valid codewords of this code.

1. **Definition**

Let \( C \) be a code. Then, \( d(C) = \min \{ d(x, y) \mid x, y \in C \} \).

2. **Definition**

A code \( C \) is exactly *t-error-detecting* if and only if \( d(C) = t + 1 \) and *t-error-correcting* if and only if \( d(C) = 2t + 1 \) or \( d(C) = 2t + 2 \).
G. CHAPTER SUMMARY

In this chapter, we discussed the basic principles and properties of the error correction codes, and the background and important concepts necessary to understand their performance. In Chapter III, Reed-Muller codes and their properties, as well as encoding-decoding-generation methods will be introduced and analyzed.
III. REED-MULLER CODES

A. DEFINING REED-MULLER CODES

Let \( 0 \leq r \leq m \). The \( r \)-th order Reed-Muller code \( R(r,m) \) is the set \( p \) of all binary strings of length \( n = 2^m \) associated with the Boolean polynomials \( p(x_1, x_2, ..., x_m) \) of degree at most \( r \).

Consequently, the 0-th order Reed-Muller code \( R(0,m) \) consists of the binary strings associated with the constant polynomials 0 and 1. This code is the repetition code of length \( 2^m \), \( R(0,m) = \{0^m, 1^m\} = \{0\ldots0, 1\ldots1\} = \text{Rep}(2^m) \).

The other extreme situation is the \( m \)-th order Reed-Muller code \( R(m,m) \), consisting of all binary strings of length \( 2^m \), that is, \( R(m,m) = F_2^n \), where \( n = 2^m \).

The number of codewords can be found easily from the count of binary monomials in \( R(r,m) \) of degree at most \( r \). There are \( \sum_{k=1}^{r} \binom{m}{k} \) such monomials, and so there are \( 2^\sum_{k=1}^{r} \binom{m}{k} \) linear combinations of these. It is obvious that the closer \( r \) is to \( m \) the more codewords there are. In conclusion, the \( r \)-th order Reed-Muller code \( R(r,m) \) has the following properties:

- **Length of codewords:** \( 2^m \)
- **Number of codewords:** \( 2^{\sum_{k=1}^{r} \binom{m}{k}} \)
- **Minimum distance between codewords:** \( 2^{m-r} \) [3]

Reed-Muller codes are among the most useful and interesting binary, linear, block codes. As we will discuss in the next paragraph, first order Reed-Muller codes of length 32 were used in space missions. In order to achieve greater performance than these codes offer, we have to extend their length. The limited bandwidth of communication channels is one thing that we have to take into account. Therefore, the use of very large codes in narrow channels is prohibited. On the other hand, Reed-Muller codes of higher order require significantly less bandwidth than the first order ones.
Many researchers have investigated the weight distribution of Reed-Muller codes, that is, the sequence of codeword weights. The weight spectrum for the first order Reed-Muller codes is found easily, since, as we will see in the next chapter, all codewords in \( R(1,m) \) codes have the same number of 0’s and 1’s (are balanced) except for the all 0’s and all 1’s codewords. For example, in \( R(1,5) \), there are \( 2^{1+5} = 2^6 = 64 \) codewords of length \( 2^5 = 32 \). Among them, there is a codeword of 32 1’s, a codeword of 32 0’s and 62 codewords of weight 16 (half 1’s, half 0’s).

Understanding the weight distribution for higher order Reed-Muller codes is complicated, and very little is known about that. Much work has been done on 2\(^{nd}\) and 3\(^{rd}\) order Reed-Muller codes [4].

B. APPLICATIONS OF REED-MULLER CODES

The first order Reed-Muller codes \( R(1,m) \), was used by Mariner 9 to transmit black and white photographs of Mars in 1972 [5]. A simplified example giving a flavor of code use in digitally transferred data is given below.

The main idea behind applying coding in digital technologies is to break up a picture or a sound into small pieces and to use a binary sequence to represent each of these small pieces, adding at the same time, some redundant bits. This redundancy is used to correct errors that might be caused by noise when the information is sent over a noisy channel.

For example, the pixels (picture elements) shown in Figure 3 could be sent via a channel by coding a white pixel with 111111, a black pixel with 000000 and a gray pixel with 111000. Assuming that the receiver knows the size of the image, in this example 6x6, and that the pixels are being sent row by row, then the picture can be accurately decoded if no more than one error occurs during the transmission process. This happens because the distance between any pair of codewords is at least 3.
In the case of Mariner 9, the actual scenario is more complicated and finally the error-correcting code used is “heavier.” This means that the additional bits used (redundant bits) are repeated information bits. In the case of Mariner 9, the codewords were 32 bits long, consisting of 6 information bits and 26 additional bits.

Another significant application of error-correcting codes is in the compact disc (CD) technology [5]. On CDs, the signal is encoded digitally. To protect from errors because of scratches, cracks and similar damage, several kind of codes are used which can correct up to 4,000 consecutive errors (about 2.5 mm of track). Similar error correction techniques are also used on DVDs and Blue-Ray discs.

We cannot ignore the contribution of codes in compression. Compression is the process of transforming information from one representation to another smaller representation. In many cases, compression and decompression processes are often referred to as encoding and decoding. It is obvious that data compression has application to data storage and data transmission. Since using a process of reducing the amount of data required to represent a given quantity of information, different amounts of data might be used to communicate the same amount of information. If the exact information can be represented with different amounts of data, it is reasonable to believe that the representation that requires more data contains some kind of data redundancy. Image
compression and coding techniques use three types of redundancies: coding redundancy, spatial redundancy, and psychovisual redundancy.

Another great concern of coding theory is synchronization. In many industrial and military activities, such as navigation, mapping, positioning, power distribution, telecommunication, weather station, and digital radio, one of the most important exchanged information is the precise time of action taking place (time tag). Synchronization between these tags is something that can be fixed and controlled by codes. With the use of specific codes any “shift” in phase of a signal can be detected and corrected, enabling the transmission of multiple signals through the same channel.

C. GENERATION/ENCODING METHODS

1. Generation Methods

a. Using Boolean Polynomials

An $r$-th order Reed-Muller code $R(r,m)$ is the set of all binary strings of length $2^m$ associated with Boolean polynomials $x_1, x_2, \ldots, x_m$ of degree at most $r$. Consequently, the first order Reed-Muller code of length $n = 2^3$ is the set of all binary strings associated with the Boolean polynomials $x_1, x_2,$ and $x_3$ of degree at most 1. These polynomials have the form $a_0 + a_1 x_1 + a_2 x_2 + a_3 x_3$ where $a_i = 0$ or 1. The binary string corresponding to this polynomial is $a_0(1111111) + a_1(0001111) + a_2(0011001) + a_3(01010101)$.

b. Example $R(1,3)$

We can list the codewords in $R(1,3)$ as follows:
Table 5. RM(1,3) codewords

<table>
<thead>
<tr>
<th>Polynomial</th>
<th>Codeword</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00000000</td>
</tr>
<tr>
<td>$x_1$</td>
<td>00001111</td>
</tr>
<tr>
<td>$x_2$</td>
<td>00110011</td>
</tr>
<tr>
<td>$x_3$</td>
<td>01010101</td>
</tr>
<tr>
<td>$x_1 + x_2$</td>
<td>00111100</td>
</tr>
<tr>
<td>$x_1 + x_3$</td>
<td>01011010</td>
</tr>
<tr>
<td>$x_2 + x_3$</td>
<td>01100110</td>
</tr>
<tr>
<td>$x_1 + x_2 + x_3$</td>
<td>01101001</td>
</tr>
<tr>
<td>1</td>
<td>11111111</td>
</tr>
<tr>
<td>1 + $x_1$</td>
<td>11110000</td>
</tr>
<tr>
<td>1 + $x_2$</td>
<td>11001100</td>
</tr>
<tr>
<td>1 + $x_3$</td>
<td>10101010</td>
</tr>
<tr>
<td>1 + $x_1 + x_2$</td>
<td>11000011</td>
</tr>
<tr>
<td>1 + $x_1 + x_3$</td>
<td>10100101</td>
</tr>
<tr>
<td>1 + $x_2 + x_3$</td>
<td>10011001</td>
</tr>
<tr>
<td>1 + $x_1 + x_2 + x_3$</td>
<td>10010110</td>
</tr>
</tbody>
</table>

Note that all codewords in $R(1,m)$ except 0 and 1 have weight $2^{m-1}$. Thus, in the previous example of $R(1,3)$, the weight of all nontrivial codewords, except 00000000 and 11111111, is $2^{3-1} = 4$.

c. **Using Direct Sum Construction**

If $C_1$ is an $R(r_1, m_1)$ code and $C_2$ is an $R(r_2, m_2)$ code, then the direct sum $C_3$ is the code $C_3 = \{cd \mid c \in C_1, d \in C_2\}$ with the following parameters:
Length of codewords: \(2^m + 2^m\)

Number of codewords: \(2^{1+\binom{m}{1} + \binom{m}{2} + \cdots + \binom{m}{r}}\)

Minimum distance between codewords: \(\min\{2^{m-r}, 2^{m-r}\}\)

d. Using \((u,u+v)\)-Construction

This construction, for many reasons, is more useful than the direct sum construction. If \(C_1\) is an \(R(r_1,m_1)\) code and \(C_2\) is an \(R(r_2,m_2)\) code, both of which are over the same alphabet (\(C_1\) and \(C_2\) have the same length), then we can define a code \(C_1 \oplus C_2\) by: \(C_1 \oplus C_2 = \{c(c+d) \mid c \in C_1, d \in C_2\}\) with the following properties [5]:

Length of codewords: \(2^{m+1} = 2^{m+1}\)

Number of codewords: \(2^{1+\binom{m}{1} + \binom{m}{2} + \cdots + \binom{m}{r}}\)

Minimum distance between codewords: \(d(C_1 \oplus C_2) \geq \min\{2^{m-r+1}, 2^{m-r}\}\)

2. Encoding Methods

To define the encoding matrix of \(R(r,m)\), let the first row of the encoding matrix be 11...1 (the vector with length 2^m with all entries equal to 1). If \(r\) is equal to 0, then this row is unique in the encoding matrix. On the other hand, if \(r\) is equal to 1, then we add \(m\) rows corresponding to the vectors \(x_1, x_2, \ldots, x_m\) to the \(R(0,m)\) encoding matrix.

Thus, in order to form an \(R(r,m)\) encoding matrix, where \(r\) is greater than 1, we have to add \(\binom{m}{r}\) rows to the \(R(r-1,m)\) encoding matrix. These added rows consist of all the possible reduced degree \(r\) monomials that can be formed using the rows \(x_1, x_2, \ldots, x_m\)
a. **Example R(1,3)**

When \( m=3 \) we then have:

\[
\begin{array}{cccccccc}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
x_1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\
x_2 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\
x_3 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\
\end{array}
\]

b. **Example R(2,3)**

Thus, adding the rows

\[
x_1x_2 = 11000000, \quad x_1x_3 = 10100000 \quad \text{and} \quad x_2x_3 = 10001000
\]

we obtain:

\[
\begin{array}{cccccccc}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
x_1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\
x_2 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\
x_3 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
x_1x_2 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
x_1x_3 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
x_2x_3 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\
\end{array}
\]

c. **Example R(3,3)**

Note that, the row \( x_1x_2x_3 = 10000000 \) can be added to form: \( R(3,3) \)

d. **Example R(2,4)**

Using exactly the same steps, we can obtain:
It is obvious that the number of rows of these encoding matrices is 

\[ k = 1 + \binom{m}{1} + \binom{m}{2} + \ldots + \binom{m}{r} \].

So, the sent message must be in blocks of length \( k \). Let \( m = (m_1, m_2, \ldots, m_k) \) be such a block. Then the encoded message \( M \) is the sum \( \sum_{i=1}^{k} m_i R_i \),

where \( R_i \) indicates the rows of the encoding matrix of \( R(r,m) \).

e. Example Encoding with \( R(1,3) \)

Using \( R(1,3) \) to encode \( m =(0011) \) gives:

\[ 0(11111111) + 0(11110000) + 1(11001100) + 1(10101010) = (01100110) \]

as the encoded word.

f. Example Encoding with \( R(2,4) \)

Similarly, using \( R(2,4) \) to encode \( m=(10101110010) \) gives:

\[ 1*(1111111111111111) + 0*(1111111000000000) + 1*(1111000001110000) + 0*(1100110011001100) \\
+ 1*(1010101010101010) + 1*(1111000000000000) + 1*(1100110000000000) + 0*(1010101000000000) \\
+ 0*(1100000011000000) + 1*(1010000010100000) + 0*(1000100010010000) = (0011100100000101) \]
3. Decoding Methods

There are few methods for decoding Reed-Muller codes. In this thesis, the most widely used is analyzed. Decoding is more complex than encoding. The theory behind both encoding and decoding is based on Hamming distance between vectors.

The decoding method checks which row $R_i$ of the encoding matrix was used to form the encoded message. The implementation of this method requires the use of characteristic vectors of the encoding matrix rows. In order to find the characteristic vector, we work on the monomial $r$ associated with the row of the matrix. After that, we take the set of all $x_i$ that are not in $r$, but only in the encoding matrix. The characteristic vectors are those vectors that correspond to monomials $x_i, \overline{x_i}$, such that exactly one of $x_i$ or $\overline{x_i}$ belongs to each monomial for all elements of the set of all $x_i$. The dot product of these characteristic vectors with all the rows of the used code matrix yields 0, except the row to which the vector corresponds.

a. Decoding Algorithm

This method is precisely described in the following three steps of an algorithm [3]:

Step 1

Choose a row of the given encoding matrix code and find $2^{m-r}$ characteristic vectors for that row. Then, form the dot product of these vectors with the encoded message.

Step 2

Compute the majority value (either 1 or 0) of the dot products, and assign it to each row.

Step 3

Executing steps 1, 2 from the bottom of the matrix to the top, multiply the majority value assigned to each row by its corresponding row. Add the results altogether, and then sum this up to the received encoded message. If there is a majority of 1’s in the
resulting vector, then assign 1 to the top row. Otherwise, if there is a majority of 0’s, then
assign 0 to the top row. Adding the top row, multiplied by the assigned value, leads to the
original encoded message. Using this algorithm, it is obvious that we can identify the
errors occurred during the transmission of encoded message. The vector that is formed
using the assigned values of each row, from the top row all the way to the bottom row of
the encoding matrix, is the original message.

b. Example of Decoding Using \( R(1,3) \)

Assuming an original message \( m=(0110) \), using the \( R(1,3) \) encoded matrix
we get the encoded message \( M=(00111100) \). As it is already mentioned, the distance in
this code is \( 2^{3-1} = 4 \), and therefore, it can correct one error. Assuming that, during
message transmission, one error occurred at the first leftmost bit, the encoded message
after the error is \( M’=(10111100) \). The characteristic vectors of the last row of the encoded
matrix are \( x_1, x_2, x_1x_2, x_1 \bar{x}_2 \) and \( \bar{x}_1, x_2 \).

The vector related to \( x_1 \) is (11110000), thus \( \bar{x}_1 \) is (00001111). Similarly,
\( x_2 \) is (11001100), and thus \( \bar{x}_2 \) is (00110011). Therefore, \( x_1x_2 \) is (11000000), \( x_1 \bar{x}_2 \) is
(00110000), \( \bar{x}_1x_2 \) is (00001100) and \( \bar{x}_1 \bar{x}_2 \) is (00000011). Computing the dot product of
these vectors with \( M’ \), we get the values 1,0,0,0 respectively, leading to majority value of
0 for \( x_1 \). Repeating the process for the second to last row of the matrix, we get the values
0,1,1,1 respectively, leading to majority value 1 for \( x_2 \). Working similarly, we conclude
that the coefficient of \( x_1 \) is also 1. Adding 0*(10101010) and 1*(11001100) and
1*(11110000) we get \( M’’=(00111100) \). Then, we notice, that adding \( M’ \) and \( M’’ \) we get
(10000000), which has more 0’s than 1’s, leading to 0 for the coefficient of the first row of
the used matrix.

Putting together the four coefficients that correspond to four rows 0,1,1,0
we get the original message. Additionally, we can determine the position of the error at
the first leftmost bit.
D. CHAPTER SUMMARY

In this chapter, a detailed discussion of Reed-Muller codes was presented. Some methods of generation, encoding and decoding are also analyzed. This will help us explain later in the thesis the simplicity of a new method of fast construction of these codes. In addition, some examples were examined to help understanding each method. In Chapter IV, the concept of covering radius is presented, and several methods for its computation are examined.
IV. COVERING RADIUS

A. INTRODUCTION

We can trace the origin of error correcting codes in a paper from the 1940s by Claude Shannon [6], who proposed some error detection/correction techniques, to achieve error-free communication through a noisy channel. Data to be sent over a noisy channel is first “encoded,” plaintext is turned into a codeword by adding extra data (redundancy). This enlarged codeword is sent via the communication channel and the received data is “decoded” by the receiver. The critical point of this last process is that the decoded data has to be as close as possible to sent data. At this point, covering radius takes its role, since the “quality” of the code, in relation to the channel, depends on how small the code’s covering radius is.

In coding theory, the covering radius plays a critical role in every code. In addition, good covering codes have a number of applications in various areas of mathematics and electrical engineering. Though the minimum distance has a more central role for error-correction codes, the covering radius is also related to the error correction capability of the code, since if it is less than the distance, no vector in the space can be added without worsening the code’s distance [7].

Since $\mathbb{F}_2^n$ has a distance metric, it makes sense to use spheres that are centered at a valid codeword $x$ with a given radius $\rho$. One sample of these spheres is depicted in Figure 4.

![Sphere of radius $\rho$.](image)

Figure 4. Sphere of radius $\rho$. 
Let \( C \) be a subset of \( F_2^n \), in which all the distances are integers. The \textit{covering radius} of a code \( C \) is the smallest radius \( \rho \) (Figure 5) such that every word of the space is contained in some (at least one) sphere of radius \( \rho \) centered at a codeword.

It is obvious that the covering radius problem is important since it helps in investigating the constructing codes having a specified length and dimension such that no vector of the space is very far from the nearest codeword.

![Figure 5. Covering radius \( \rho \).](image)

Each codeword of a code \( C \subseteq F_2^n \) represents a message. When that message is transmitted, errors may occur. However, if the used code \( C \) has the property that all the spheres of radius \( \rho \) around codewords are completely disjoint, then any received message \( x \) that has no more than \( \rho \) coordinates in error is within distance \( \rho \) from a unique codeword \( c \) in \( C \). Therefore, we conclude that the codeword that was originally sent is \( c \). Consequently, we say that \( C \) can correct up to \( \rho \) errors. It is obvious that the largest value of \( \rho \) cannot be greater than \( d \) (the distance between any two codewords of \( C \)). The critical point here is the ability of constructing error-correcting codes, having specified length and dimension (number of codewords in linear cases) with large minimal \( d \). This is actually one of the central problems in theory of error-correction codes.

In addition, it is worth mentioning that covering radius is a basic geometric parameter of a code. Topics that are currently under research by the coding community are the following:
1. Given the length and dimension of a linear code, it should be determined what the covering radius is.
2. Construct efficient codes that have small covering radius.
3. Develop computational methods to determine the covering radius of well-known error-correction codes.

Specifically, Reed-Muller codes are an extremely interesting class of error-correction codes, and therefore, many researchers have studied Reed-Muller codes [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24], [25]. Nevertheless, due to the complexity of computations methods, overall knowledge is still quite limited. We will focus on some of these methods and point out all published results of covering radius of 1, 2, and k-th order Reed-Muller codes later in the chapter.

B. METHODS OF COMPUTATIONS OF COVERING RADIUS

1. 1st Method Using Translate

When we construct an error-correction code with large minimum distance $d$, our focus is in the structure of the code. In addition, the corresponding codewords must be chosen, so that no vector of $F_2^n$ (since in this thesis we only work with binary spaces) has its distance too large from any codeword.

On the other hand, the design of a decoding scheme focuses on the exterior part of the code. If we have a code $C \subset F_2^n$, and we decide to send some data in the form of a codeword, then, on the receiver we may get a vector $x$ that is different from $c$.

Thus, we can now introduce the concept of a translate $x+C$ of $C$. This is the set of all codewords of the code $C$ xoring with a specific received word $x$. The weight of the translate $x+C$ is the minimal weight of any vector in $C$. Knowing the weights of translates, is very critical in the decoding problem since the covering radius is the largest among all weights of translates.

In the following example, we pick the code $C=\{00000,11000,00111,11111\}$, and we calculate the covering radius using the method we just introduced (see Table 6). Note, that the code we use in this part is an arbitrary code with no specific properties. We use this code for the sake of simplicity of our example.
Having the codewords of the code $C$ and all vectors that can be received (received words), we can calculate the translates of the code, and thus the weights of these translates. Therefore, the maximum of these weights, 2 in our case, is the covering radius of the code. In Table 6, we see this method in detail.

Table 6. 1st method of Covering Radius computation

<table>
<thead>
<tr>
<th>Codewords of $C$ (transmitted words)</th>
<th>Vectors $x$ (received words)</th>
<th>Minimum $wt(x+C)$ ($x+C$ translates)</th>
<th>${\text{Max}[\text{min}(wt(x+C))]}$ (covering radius)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>00001</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>00010</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>00011</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>00100</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>00101</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>00110</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>00111</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>01000</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>01001</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>01010</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>01011</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>01100</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>01101</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>01110</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>01111</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10000</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10001</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10010</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10011</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10100</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10101</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10110</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10111</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>11000</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>11001</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>11010</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>11011</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>11100</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>11101</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>11111</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>00000</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Furthermore, applying the same method to the following code $C’=\{00000,11000,00111\}$ we realize that even though we decrease the dimension of the code from 4 to 3, the covering radius remains the same. This code is actually a trivial sub-code of the given code. In reality, constructing a sub-code with “nice characteristics,” and properties is not a triviality.
In our case, the covering radius of the random code is 2. As we have already mentioned above, the covering radius is also the smallest integer $r$ such that any vector in $F_2^n$ is within distance $r$ from a codeword.

This method works well for codes with small length and dimension. When these parameters become larger, the computation complexity of the method increases exponentially, and the use of computers is necessary.

2. **2\textsuperscript{nd} Method of Using Direct Sum of Codes**

Before we describe the 2\textsuperscript{nd} method of covering radius computation [11], we give several definitions.

\textbf{a. Definition of Norm of a Code C}

Let $C \subset F_2^n$ be a linear code of length $r$, dimension $m$ and covering radius $R$. Let $j$ be one of the $m$ coordinates, and $C_j$ denote the set of codewords in which the $j$-th coordinate is 0. Similarly, let $C_j$ denote the set of codewords in which the $j$-th coordinate is 1. In accordance with [22], if $C_j$ is not empty, then both $C_1, C_2$ contain $2^{m-1}$ codewords. For any vector $x$ in $F_2^n$, let $d_1 = d(x, C_1)$ and $d_2 = d(x, C_2)$. Also let $D = \max(d_1, d_2)$. Then, $D$ is called the norm of $C$. Norm does not depend on the choice of $x$ or $j$.

\textbf{b. Definition of a Normal Code}

A code is normal when its norm satisfies $D \leq 2R + 1$. In other words [22], given a code with norm $D$, then there is a coordinate $i$ such that, for any vector $x$, the sum of the $d$'s from $x$ to the nearest codeword having in $i$-th place 0 and to the nearest codeword having in $i$-th place 1, cannot be greater than $D$.

Having defined the critical concepts of the norm of a code, and normal codes, we can now proceed to the second method of computation of covering radius that is combined with a code construction method.
Let $C_1$ be an $R(r_1,n_1)$ code with covering radius $R_1$, and $C_2$ be an $R(r_2,n_2)$ code with covering radius $R_2$. The direct sum of these codes [11] is another code of length $2^{n_1} + 2^{n_2}$ vectors $u|v$, where $u \in C_1$ and $v \in C_2$. Then, this direct sum is a new code with covering radius $R_1 + R_2$. Additionally, if $C_1$ and $C_2$ are normal, we can construct their amalgamated direct sum [11] that is a code with one less coordinate, one less dimension, and $R_1 + R_2$ covering radius.

c. Example $R_1 + R_2$

Consider the code $C = \{00000, 11000, 00111, 11111\}$ that is the direct sum of the following codes: $C_1 = \{00, 11\}$ and $C_2 = \{000, 111\}$. Using the last method of direct sum, we conclude that the covering radius is $R = R_1 + R_2 = 2$.

3. 3rd Method Using Bounds

Let $C_1$ be any code of length $2^{n_1}$ and $b$ any vector of the same length. If $b \not\in C_1$ and $C = C_1 \cup (b + C_1)$ and if we can find a vector $y \not\in C_1$ such that $d(y, C_1) = r$, then from [11] the covering radius of $C$ is at least $\left\lceil \frac{r}{3} \right\rceil$.

Pick an arbitrary code, say $C_1 = \{0000, 1100, 0011, 1111\}$, we can calculate the covering radius of code $C = C_1 \cup (b + C_1)$ (Table 7) using the 3rd method we just introduced. The code we use is a random code with no specific properties.
We picked a vector $b \not\in C_1$, and we construct the code $C = C_1 \cup (b + C_1)$. Choosing a vector $y \not\in C_1$ with $d(y, C_1) = 1$ and using the 3rd method of computation, we conclude that the covering radius of $C$ is at least $\left\lceil \frac{1}{3} \right\rceil = 1$. Thus, a lower bound of the covering radius of $C$ is 1.

It is very difficult to find the covering radius of a large code, or even to bound it [18], [22]. Therefore, in the majority of the cases, the 3rd method for the computation of the covering radius is very useful.

Table 7. 3rd method of Covering Radius computation

<table>
<thead>
<tr>
<th>Codewords of $C_1$</th>
<th>Vector $b \not\in C_1$</th>
<th>Codewords of $C = C_1 \cup (b + C_1)$</th>
<th>Vector $y \not\in C_1$ ($d(y, C_1) = 1$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>1010</td>
<td>0000</td>
<td>1000</td>
</tr>
<tr>
<td>1100</td>
<td></td>
<td>1100</td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td></td>
<td>0011</td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td></td>
<td>1111</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1010</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0110</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1001</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0101</td>
<td></td>
</tr>
</tbody>
</table>

4. 4th Method Using Norm

This method relies on Theorem 1 [11], which states that every code of norm $N$ has a covering radius $\rho \leq \left\lceil \frac{N}{2} \right\rceil$. The equality holds for normal codes.
C. COVERING RADIUS FOR 1ST ORDER REED-MULLER CODES

Recall that, $R(r,m)$ is an $r$th order Reed-Muller code of length $2^m$, and $\rho(r,m)$ is its covering radius. One of the challenging problems in coding theory is to find precisely the covering radius of 1st order Reed-Muller codes.

The first expression for $\rho(1,m)$ was published in 1978 [22].

$$\rho(1,m) = 2^{m-1} - 2^{\frac{m-1}{2}} \text{ for even } m,$$

$$2^{m-1} - 2^{\frac{m-1}{2}} \leq \rho(1,m) \leq 2^{m-1} - 2^{\frac{m}{2}} \text{ for odd } m.$$ 

For the first odd values of $m$, we have that $\rho(1,1)=0$, $\rho(1,3)=2$, $\rho(1,5)=12$ (also proved in [8]) and $\rho(1,7)=56$ (also proved in [14]). An easy but unsafe conclusion [14] was that with odd values $\rho(1,2t+1)$ is equal to $2^{2t} - 2^{t}$, thus to the lower bound of last inequality.

In 1983, the last conjecture was finally disproved [24], and it was shown that:

$$\rho(1,m) \geq 2^{m-1} - \frac{27}{32} 2^{\frac{m-1}{2}} \text{ for odd } m \geq 5.$$ 

In 1990, a correction of this proof is also provided [26].

D. COVERING RADIUS FOR 2ND ORDER REED-MULLER CODES

One of the first detailed studies to find the covering radius for 2nd order Reed-Muller codes was in [15], where it was proved that $\rho(2,6)=18$. In the same paper, some bounds are also provided:

$$36 \leq \rho(2,7) \leq 46, \text{ and } \rho(2,8) \geq 72.$$ 

Recently, in [27] a new upper bound of 2nd order Reed-Muller codes was published: $\rho(2,m) \leq 2^{m-1} - \sqrt{152 \frac{m}{3}} + O(1)$. 

E. COVERING RADIUS FOR RTH ORDER REED-MULLER CODES

Some known results for rth order Reed-Muller codes are shown below. In the following trivial cases, we have: $\rho(m, m) = 0$, $\rho(m - 1, m) = 1$, and $\rho(m - 2, m) = 2$.

In [13], it is proved that:

\[
\rho(m - 3, m) = m + 2, \text{ for } m \text{ even and } m \geq 3, \text{ and }
\]

\[
\rho(m - 3, m) = m + 1, \text{ for } m \text{ odd and } m \geq 3.
\]

F. CHAPTER SUMMARY

In this chapter, the concept of covering radius of a code is introduced. In addition, some methods of covering radius computation are presented and finally some known results for Reed-Muller codes are reported. In the next chapter, a new simplified algorithm of fast generation of all 1st order and some of 2nd order Reed-Muller codes is analyzed and a fast construction of a linear subcode with good properties is presented and analyzed. In addition, the “reverse” of this new algorithm is presented.
V. FAST ALGORITHM OF GENERATION OF 1ST—2ND ORDER REED-MULLER CODES, LINEAR SUBCODES WITH GOOD PROPERTIES, AND THE "REVERSE" ALGORITHM

A. FAST GENERATION OF 1ST ORDER REED-MULLER CODES

Comparing Tables 4 and 5, and the constructing method of Reed-Muller codes, we notice that all of the 1st order Reed-Muller codes are Affine Boolean Functions. Also, all 1st order Reed-Muller codewords are balanced, except the all 0’s and all 1’s codewords. The construction of these codewords using a conventional method is time and memory consuming. Therefore, a new algorithm for fast generation is introduced in this chapter. The algorithm is useful for hardware coding applications.

Using a Lemma in [28] which states: “An affine function in more than 2 variables is a linear string made up of the 8 4-bit blocks: \( T_i = \{ A=0000, \bar{A}=1111, B=0011, \bar{B}=1100, C=1001, \bar{C}=0110, D=0101, \bar{D}=1010 \} \) in a block sequence \( I_1, I_2, \ldots, I_{2^{n-2}} \) given as follows:

- The first block \( I_1 \) is one of \( A, \bar{A}, B, \bar{B}, C, \bar{C}, D \) or \( \bar{D} \).
- The second block \( I_2 \) is either \( I_1 \) or \( \bar{I}_1 \).
- The next two blocks \( I_3, I_4 \) are \( I_1, I_2 \) or \( \bar{I}_1, \bar{I}_2 \).
- The next four blocks \( I_5, I_6, I_7, I_8 \) are \( I_1, I_2, I_3, I_4 \) or \( \bar{I}_1, \bar{I}_2, \bar{I}_3, \bar{I}_4 \).
- The last \( 2^{n-3} \) blocks \( I_2^{n-3+1}, \ldots, I_2^{n-2} \) are \( I_1, \ldots, I_2^{n-3} \) or \( \bar{F}_{2^{n-2}} | T_1 \).

We can construct all 1st order Reed-Muller codes using the algorithm described below. In our case when a 1 occurs, we complement and, when 0 occurs, we just copy the block as it is.
1. **New Algorithm for Fast Generating 1st Order RM Codes**

**Step 1**

We begin with the codewords of $R(1,2)$ that are identical to the 4-bit blocks used in previous lemma: $T_i = \{ A = 0000, \overline{A} = 1111, B = 0011, \overline{B} = 1100, C = 1001, \overline{C} = 0110, D = 0101, \overline{D} = 1010\}$.

**Step 2**

We construct the following concatenation for each $R(1,2)$ codeword: $F_2^{n-2} \mid T_i$ in order to construct the $R(1,n)$ code. The first part of this structure will play the role of “guide” word.

**Step 3**

Starting from the leftmost bit of “guide” word, we just complement the bits of right part when we find 1 and just repeating these bits when we find 0, until we take the last rightmost bit of “guide” word.

We repeat step 3 for every block of $T_i$ using every “guide” word. In Table 8, we generate $R(1,3)$ using the new algorithm for fast generating 1st order RM codes. Therefore, for this case, we repeat step 3 twice since there are two “guide” words for every block of $T_1$. On the contrary, in Table 9, step 3 is repeated four times, since $F_2^{n-2} = F_2^{4-2} = F_2^2 = \{00,11,01,10\}$.

2. **Example R(1,3)**

Using the above algorithm we construct $R(1,3)$. 
Table 8. Fast Generation of $R(1,3)$

<table>
<thead>
<tr>
<th>Step 1 $(R(1,2))$</th>
<th>Step 2</th>
<th>Step 3 $(R(1,3))$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0000</td>
<td>00000000</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>00011111</td>
</tr>
<tr>
<td>1111</td>
<td>0111</td>
<td>11111111</td>
</tr>
<tr>
<td></td>
<td>1111</td>
<td>11110000</td>
</tr>
<tr>
<td>1100</td>
<td>0100</td>
<td>11001100</td>
</tr>
<tr>
<td></td>
<td>1100</td>
<td>11000011</td>
</tr>
<tr>
<td>0011</td>
<td>0011</td>
<td>00110011</td>
</tr>
<tr>
<td></td>
<td>1100</td>
<td>00111100</td>
</tr>
<tr>
<td>1001</td>
<td>0101</td>
<td>10011001</td>
</tr>
<tr>
<td></td>
<td>1101</td>
<td>10010110</td>
</tr>
<tr>
<td>0110</td>
<td>0110</td>
<td>01100110</td>
</tr>
<tr>
<td></td>
<td>1011</td>
<td>01101001</td>
</tr>
<tr>
<td>1010</td>
<td>0101</td>
<td>10101010</td>
</tr>
<tr>
<td></td>
<td>1010</td>
<td>10100101</td>
</tr>
<tr>
<td>0101</td>
<td>0101</td>
<td>01010101</td>
</tr>
<tr>
<td></td>
<td>1010</td>
<td>01011010</td>
</tr>
</tbody>
</table>

3. Example $R(1,4)$

Using the same algorithm we construct $R(1,4)$. 
Table 9. Fast Generation of $R(1,4)$

<table>
<thead>
<tr>
<th>Step 1 $(R(1,2))$</th>
<th>Step 2</th>
<th>Step 3 $(R(1,4))$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>00</td>
<td>0000</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>0000</td>
</tr>
<tr>
<td></td>
<td>01</td>
<td>0000</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>0000</td>
</tr>
<tr>
<td>1111</td>
<td>00</td>
<td>1111</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>1111</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1100</td>
<td>00</td>
<td>1100</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>1100</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0011</td>
<td>00</td>
<td>0011</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1001</td>
<td>00</td>
<td>1001</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0110</td>
<td>00</td>
<td>0110</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1010</td>
<td>00</td>
<td>1010</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0101</td>
<td>00</td>
<td>0101</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

The complexity of constructing the 1\textsuperscript{st} order Reed-Muller codes using this algorithm is significantly lower than the complexity of the method that is introduced in Chapter III, by using Boolean polynomials.
In addition, it is obvious that this compact representation of 1st order Reed-Muller codewords is highly memory efficient because it can actually store a great amount of information in a small word. For example, using a “guide” word of 8 bits, we can compress a codeword of 512 bits to a string of 12 bits. The compression ratio for each codeword in this example is 43:1, and the memory saving is 97.65%. The compressed string includes all the information of the expanded codeword, and moreover, as we analyze below, using the “reverse” algorithm, we can reconstruct a damaged codeword, correcting some errors occurred during the transmission.

B. HARDWARE IMPLEMENTATION OF ALGORITHM

In general, in our algorithm, in order to store a compact representation of $2^n$-bits codeword, $n+1$ bits are needed. This implies that the storage ratio is

$$\frac{\text{bits of complete codeword}}{\text{bits of compact codeword}} : 1 = \frac{2^n}{n+1} : 1,$$

and the storage saving is

$$\left(1 - \frac{\text{bits of compact codeword}}{\text{bits of complete codeword}}\right) \% = \left(1 - \frac{n+1}{2^n}\right) \%.$$ It is obvious that the storage ratio, and the storage saving are very high. The critical point is that the fast generation of complete codewords from compact form cannot be efficiently supported by a program running on a conventional computer. On the other hand, the hardware implementation of this expansion (see Figure 6, for the case of $n=3$) is faster and more compact.

The logic circuit of Figure 6 generates only one codeword at a time. In order to obtain the whole code, we have to repeat this circuit for each word of either $T_1$ or $T_2$ and for each “guide” word.
Figure 6. Hardware implementation of algorithm \((n=3)\)

The exclusive OR gates implemented in Figure 6, either complements or leaves uncomplemented the corresponding bits depending on the value of inputs \(s_4\). If \(s_4 = 1\), the output of the gate is complemented, otherwise stays unchanged.

The number of two-input exclusive OR gates that are needed for the implementation of our algorithm is \(2^n - 4\), where \(n\) is the number of variables used. Although exponential in \(n\), this is close to minimal mostly because \(2^n\) outputs are needed, four of which are driven directly by their inputs and thus, require no gate. The delay associated with this logic circuit is also small.

From the above, we conclude that our conversion algorithm gives to any communication user the ability to produce complete low order Reed-Muller codewords on-the-fly from a compressed representation.
C. FAST GENERATION OF 2ND ORDER REED-MULLER CODES

On the other hand, fast generation of 2nd order Reed-Muller codes is more complicated. This problem is comparable to the construction of all $n$-variable quadratic functions: $2^{\binom{n}{2} + 1}$. We just demonstrate the fast construction of $R(2,3)$, since the generation of $RM(2,n)$ for $n>3$ is quite complicated, and we have not been able to achieve it in its generality.

We define $T_2 = \{ E = 1000, \overline{E} = 0111, F = 0001, \overline{F} = 1110, G = 0100, \overline{G} = 1011, H = 0010, \overline{H} = 1101 \}$.

Any codeword of $R(2,3)$ has the structure $T_1 | T_1$ or $T_2 | T_2$, as mentioned in [29]. Thus, in Table 10, we see this fast generation. This way of construction is less complicated and less memory consuming than the normal way.
Table 10. Fast Generation of $R(2,3)$

|       | $T_1|T_1$ | $T_2|T_2$ |
|-------|---------|---------|
| 0000  | 0000111 | 1000011 |
| 1111  | 0000011 | 1000001 |
| 0011  | 00001100| 10001110|
| 1100  | 00001001| 10000100|
| 1001  | 00000110| 10001011|
| 0110  | 00000101| 10000010|
| 0101  | 00001010| 10001101|
| 1010  | 11111111| 01110111|
| 1000  | 11100111| 01100011|
| 0111  | 11111100| 01111110|
| 0001  | 11110011| 01110100|
| 1110  | 11110001| 01110011|
| 0100  | 11110111| 01110010|
| 1011  | 11111010| 01111101|
| 1011  | 11110101| 01110010|
| 0100  | 11111010| 01111101|
|       | ...     | ...     |

D. FAST GENERATION OF LINEAR SUBCODES WITH GOOD PROPERTIES

Having fast constructed all 1$^\text{st}$ order Reed-Muller codes using the eight 4-bit blocks: $T_1=\{A=0000, \overline{A}=1111, B=0011, \overline{B}=1100, C=1001, \overline{C}=0110, D=0101, \overline{D}=1010\}$ and the given algorithm, we demonstrate a fast generation of $R(2,3)$ using the eight 4-bit blocks: $T_2=\{E=1000, \overline{E}=0111, F=0001, \overline{F}=1110, G=0100, \overline{G}=1011, H=0010, \overline{H}=1101\}$.

Again using the algorithm:
1. Algorithm

Step 1

We begin with the 4-bit blocks given above: $T_2 = \{ E = 1000, \overline{E} = 0111, \; F = 0001, \; \overline{F} = 1110, \; G = 0100, \; \overline{G} = 1011, \; H = 0010, \; \overline{H} = 1101 \}$

Step 2

We construct the following concatenation for each of $T_2$ blocks: $F_{2^n-2} | T_2$ in order to construct a new category of error correction codes with good properties. The first part of this structure plays the role of “guide” word.

Step 3

Starting from the leftmost bit of “guide” word, we complement the bits of right part when we find 1 and repeat these bits when we find 0, until we reach the last rightmost bit of “guide” word.

We repeat step 3 for every block of $T_2$ using every “guide” word.

We prove that the properties for this construction hold for any $n$. Therefore, all codewords of any such construction are of the form: $affine + x_{n-1}x_n$. Consequently, the sum of any two codewords is an affine function, and also the sum of any three codewords belongs to the code.

2. Example R(2,3) Subcode

Using the above algorithm, we construct the new subcode as shown in the Table 11.
Table 11. Fast Generation of a $R(2,3)$ subcode

<table>
<thead>
<tr>
<th>Step 1</th>
<th>Step 2</th>
<th>Step 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0</td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1000</td>
</tr>
<tr>
<td>0111</td>
<td>0</td>
<td>0111</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0111</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>0001</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>1110</td>
<td>0</td>
<td>1110</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1110</td>
</tr>
<tr>
<td>0100</td>
<td>0</td>
<td>0100</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0100</td>
</tr>
<tr>
<td>1011</td>
<td>0</td>
<td>1011</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1011</td>
</tr>
<tr>
<td>0010</td>
<td>0</td>
<td>0010</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0010</td>
</tr>
<tr>
<td>1101</td>
<td>0</td>
<td>1101</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1101</td>
</tr>
</tbody>
</table>

3. Example $R(2,5)$ Subcode

Using the same algorithm, we generate another code that has 32 codewords and some important properties, as described below. Table 12 shows the Truth Table and Algebraic Normal Form of this construction:
### Table 12. Fast Generation of $R(2,5)$ subcode

<table>
<thead>
<tr>
<th>Step 1</th>
<th>Step 2</th>
<th>Step 3</th>
<th>Algebraic Normal Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>00</td>
<td>1000</td>
<td>00010001000100010001</td>
</tr>
<tr>
<td>11</td>
<td>1000</td>
<td>10000111011101110000</td>
<td>$1 + x_4 + x_3 + x_3x_4 + x_2 + x_1$</td>
</tr>
<tr>
<td>01</td>
<td>1000</td>
<td>10000100001110111011</td>
<td>$1 + x_4 + x_3 + x_2x_4 + x_1$</td>
</tr>
<tr>
<td>10</td>
<td>1000</td>
<td>10000111000011101111</td>
<td>$1 + x_4 + x_3 + x_3x_4 + x_2$</td>
</tr>
<tr>
<td>0111</td>
<td>00</td>
<td>0111</td>
<td>01110111011101110111</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>0111</td>
<td>01110001000011101110</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0001</td>
<td>00</td>
<td>0001</td>
<td>00010001000100010110</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>0001</td>
<td>00011110111000100001</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1110</td>
<td>00</td>
<td>1110</td>
<td>11101110111011101110</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0100</td>
<td>00</td>
<td>0100</td>
<td>01000100010001000100</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1011</td>
<td>00</td>
<td>1011</td>
<td>10111011101110111111</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0010</td>
<td>00</td>
<td>0010</td>
<td>001000100010000100010</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1101</td>
<td>00</td>
<td>1101</td>
<td>11011011011101111011</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
All codewords in this construction are affine functions with the term $x_3x_4$. In addition, there is an even number of 1’s in every codeword of this construction. The first property implies that the sum of any two codewords is an affine function. An interesting property of this error-correcting code is that xoring any three codewords gives another codeword. In addition, the minimum distance $d$ of this subcode is 8.

4. Theorem

All codewords of any subcode generated by this algorithm, are of the form

$$\text{affine} + x_{n-1}x_n.$$ 

Before we prove the theorem we have to present an algorithm for calculating the Algebraic Normal Form from the Truth Table of a function and vice versa [30]. Let $D = [d_0\ d_1\ d_2\ldots d_{2^n-1}]$ be the coefficient vector of the polynomial representing the Boolean function $f$ (the theorem that helps us calculate the coefficient vector is presented below). If $d_i = 1$, where $0 \leq i \leq 2^n - 1$, then the monomial $x_0^{i_0}x_1^{i_1}x_2^{i_2} \ldots x_{n-1}^{i_{n-1}}$ appears in the Algebraic Normal Form of $f$. On the contrary, when $d_i = 0$, no monomial appears, where $(i_0, i_1, i_2, \ldots, i_{n-1})$ is the binary representation of pointer $i$.

a. Example of Calculating the ANF of a Function

$D = [0\ 0\ 1\ 0\ 0\ 1\ 1]$,

means $d_0 = 0, d_1 = 0, d_2 = 1, d_3 = 0, d_4 = 0, d_5 = 0, d_6 = 1, d_7 = 1$. We conclude that:

- due to $d_5$ (i=01000000 in binary representation), one of the monomials that appears in the Algebraic Normal Form is $x_0^0x_1^1x_2^0x_3^0x_4^0x_5^0x_6^0x_7^0 = x_1$.

- due to $d_5$ (i=10100000 in binary representation), one of the monomials that appears in the Algebraic Normal Form is $x_0^1x_1^0x_2^1x_3^0x_4^0x_5^0x_6^0x_7^0 = x_0x_2$. 

46
due to \( d_6 (i = 01100000 \text{ in binary representation}) \), one of the monomials that appears in the Algebraic Normal Form is \( x_0^0 x_1^1 x_2^1 x_3^0 x_4^0 x_5^0 x_6^0 x_7^0 = x_1 x_2 \).

due to \( d_7 (i = 11100000 \text{ in binary representation}) \), one of the monomials that appears in the Algebraic Normal Form is

\[
x_0^1 x_1^1 x_2^1 x_3^0 x_4^0 x_5^0 x_6^0 x_7^0 = x_0 x_1 x_2.
\]

Finally, the Algebraic Normal Form of the given coefficient vector is \( x_1 + x_1 x_2 + x_0 x_2 + x_0 x_1 x_2 \). Now, we have to connect the coefficient vector with the Truth Table of the function, using the theorem in [30]. This theorem states that if we have an \( n \)-variable Boolean function \( f \), and \( D \) the coefficient vector of this function, then \( D = f \ast A_n \), where

\[
A_i = \begin{pmatrix} A_0 & A_0 \\ 0 & A_0 \end{pmatrix} \text{ and } A_0 = [1].
\]

b. Example of Calculating the Coefficient Vector

Given a Truth Table of a 3-variable Boolean function \( f = 01100101 \) and working in accordance to the above theorem, we can obtain the coefficient vector. Since,

\[
A_3 = \begin{bmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{bmatrix}
\]

we obtain \( D = [0 \ 1 \ 1 \ 0 \ 0 \ 0 \ 1 \ 0] \).

Our previous theorem claims that all codewords of any such construction are of the form: \( affine + x_{n-1} x_n \).
c. Proof of Theorem

Indeed, every function of our construction corresponds to coefficient vectors of the form: \( D = [d_0 \ d_1 \ d_2 \ 1 \ d_4 \ 0 \ 0 \ d_8...0...d_{2^n-1}...0...] \), where:

1. \( d_0 \) can be either 1 or 0 since the 1\(^{st}\) bit of our codewords is either 1 or 0 and the 1\(^{st}\) column of \( A_n \) is \([1 \ 0 \ 0 \ 0...]^T\).

2. \( d_1 \) can be either 1 or 0 since the first 2 bits of our codewords are 00, 01, 10 or 11 and the 2\(^{nd}\) column of \( A_n \) is \([1 \ 1 \ 0 \ 0...]^T\).

3. \( d_2 \) can be either 1 or 0 since the first three bits of our codewords are 000, 001, 111, 110, 100, 011, 010 or 101 and the 3\(^{rd}\) column of \( A_n \) is \([1 \ 0 \ 1 \ 0...]^T\).

4. \( d_3 \) can only be 1 since \( T_2 \) consists of words of odd number of 1’s and the 4\(^{th}\) column of \( A_n \) is \([1 \ 1 \ 1 \ 0 \ 0 \ 0...]^T\).

5. \( d_4 \) can be either 1 or 0 since the first bit and the \( 2^{n-1} \) th bit of our codewords are 00, 01, 10 or 11 and 5\(^{th}\) column of \( A_n \) is \([1 \ 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 1...]^T\).

For the other bits of \( D \), except \( 2^n \) th bits, it is obvious that they are all 0’s. On the other hand, all \( 2^n \) th bits of \( D \) can be either 1 or 0. That format of coefficient vector confirms that the Algebraic Normal Form of our construction is \( affine + x_{n-1}x_n \).

QED

E. THE DECODING “REVERSE” ALGORITHM

Reversing the algorithm introduced at the very beginning of this chapter, we show that not only we can highly compress any codeword of 1\(^{st}\) order Reed-Muller codes, and of new construction of subcodes of 2\(^{nd}\) order Reed-Muller codes, but we can also reconstruct a damaged codeword, correcting some errors that occurred during transmission or storage.
1. Conjecture

This new correction method can correct \( n-2 \) errors, or similarly the number of “guide” word bits.

The general compression ratio of this “reverse” algorithm, as it is already mentioned, is \( \frac{2^n}{n+1} \), and the memory saving is \( \left(1 - \frac{n+1}{2^n}\right)\% \). It is obvious that, for high \( n \), the compression ratio is extremely high. For example, we can imagine the memory saving in a realistic case for \( n=15 \), where we can compress \( 2^{15} = 32768 \) bits to only 16 bits, without losing any information of the codeword. The memory saving in this particular case is 99.95\%. These calculations highlight the importance of this algorithm in environments where the memory efficiency is critical.

2. Algorithm

Step 1

We split the word in two halves, and both of these halves in halves and so on, until we reach 4 bit chunks.

Step 2

We xor bitwise the first two halves, and using the majority value of either 0’s or 1’s, we obtain the rightmost bit of “guide” word. At the same time, the minority of either 1’s or 0’s indicates the probable positions of errors in both halves of our word. It is obvious that on first xoring we either/both miss some errors due to double errors occurred on two xored bits, or/and over count some of them due to the fact that error candidates are in both halves.

In order to accurately locate and correct all of the errors, we xor the other halves, and we work as described on the second step. The bit positions that take the majority of candidate errors are the errors.

Our decision about the position of errors can be verified by our construction of the Reed-Muller code used. It is known that using our algorithm, all 1st order Reed-Muller codes can be generated by \( T_1 \) set of words, and all subcodes of 2nd order Reed-Muller
codes that we quickly generated, by $T_2$ set of words. Therefore, all 4-bits words, after the very last split, should be either of $T_1$ set, in our construction of 1st order Reed-Muller codes, or $T_2$ set, in our construction of subset of 2nd order Reed-Muller codes. Note, all the $T_1$ set of words has distance 1 or 3 from the $T_2$ set of words.

Having corrected all $n-2$ errors, we finally obtain the complete compressed word.

It is obvious that the compression ratio is as high as the size of “guide” word.

a. Example 16 Bits

Suppose we transmit the codeword 1101001011010010, and we actually receive the word 1111001010010010. This word has two errors on 3rd and 10th bits. Splitting the received word in two halves, we obtain 11110010 | 10010010. Xoring these halves bitwise, we take 01100000. This string gives us the information that the rightmost bit of “guide” word is 0, and errors might occur in the 2nd bits of one of the halves (2nd or 10th bit of the word), and in the 3rd bits of one of the halves (3rd or 11th bit of the word).

In order to detect in which of the first halves the errors located, we xor the subsequent halves bitwise. Thus, for the left halves we obtain 1101, and the information we obtain is that the left most bit of our “guide” word is 1, and the probable errors are in 3rd or 7th bit of our word. Working identically for the right halves, we obtain 1011, and the information we obtain is that the probable errors are in 2nd or 6th bit of right half (10th or 14th bit of our word). Combining the information of three xorings, getting two votes for 3rd and 10th positions, we conclude that the errors are in 3rd, and 10th bits. Thus, the compressed word is 10|1101.

In all combinations of $n-2=4-2=2$ errors in the received word, the xoring manipulation can inform us for their position. In case there are no errors in the received word, the algorithm proceeds without the correction process.

b. Example 32 Bits

In this example, we use 32-bit codewords, and we correct three errors that occurred in the same set of 4-bits. Suppose we transmit the codeword
10110100101101000100101101001011, and we actually receive the word 101101010110010010110101001011. This word has three errors at the 5th, 6th, and 7th bits. Splitting the received word into two halves, we obtain 1011101010110100 | 0100101101001011. Xoring these halves bitwise, we obtain 11110001111111. This string gives us the information that the rightmost bit of the “guide” word is 1, and errors might occur in 5th bits of one of the halves (5th or 21st bit of the word), in 6th bits of one of the halves (6th or 22nd bit of the word), and in the 7th bits of one of the halves (7th or 23rd bit of the word).

In order to detect in which of the first halves the errors are located, we xor the subsequent halves bitwise. Thus, for the left 16-bits half 1011101010110100 we obtain 00001110, and the information we obtain is that the middle bit of our “guide” word is 0, and the probable errors are in 5th or 13th bit of our word, 6th or 14th bit of our word, and 7th or 15th bit of our word. Working identically on the right 16-bit half 0100101101001011, we obtain 00000000, and there is no useful information.

At this point, the information we have for the position of errors in our word seems sufficient, but for the sake of the completion of our algorithm, we keep on xoring until we get the 4-bits sets. Therefore, xoring the very first 8-bits set we obtain 0001, and the information we obtain is that the leftmost bit of “guide” word is 0, and the probable errors are in the 4th or 8th bit. Xoring the next 8-bits set 10110100 we obtain 1111 and there is no usable information.

Combining the information of the xor processes, we obtain two votes for 5th, 6th, and 7th positions, we conclude that the errors are in these bits. Thus, the compressed word is 101|1011.

In all combinations of n-2=5-2=3 errors in the received word, the xor manipulation can inform us of their position. It is obvious that there are many cases where our “reverse” algorithm can correct more than n-2 errors, but we cannot generalize based on these cases only. In case there are no errors in the received word, the algorithm proceeds without the correction process.
F. CHAPTER SUMMARY

In this chapter, a new simplified algorithm of fast generation of all 1st order and some of 2nd order Reed-Muller codes is analyzed and a fast construction of a linear subcode of 2nd order Reed-Muller code with good properties is presented and analyzed. A hardware implementation of this algorithm is also presented for $n=3$. In addition, the “reverse” of the algorithm is introduced, showing at the same time, the process of decoding. In Chapter VI, we summarize the conclusions of this thesis and future work is proposed.
VI. CONCLUSIONS AND FUTURE WORK

This thesis points out the difficulty of completely estimating a critical property of error-correcting codes, namely the covering radius of a code. This covering radius problem plays a critical role, along with minimum Hamming distance and decoding complexity, to our decision of choosing the most efficient error-correcting code. Nevertheless, even though it is a well-defined property in coding theory, in the majority of the codes, it can only be bounded and not exactly calculated.

Further, in this thesis, a new method of fast generation of $1^\text{st}$ order $R(1,n)$ Reed-Muller codes was introduced. This method seems highly memory efficient and fast, since we generate all $1^\text{st}$ order Reed-Muller codes using just the $T_1$ set of 4-bit words and entire $F_2^{n-2}$ field. For example, to generate $2^{1+\binom{7}{1}} = 2^8 = 256$ codewords of $2^7 = 128$ bits length, thus $R(1,7)$ code, we just need the whole set of $T_1$ (32 bits), as well as the entire $F_2^{n-2} = F_2^{7-2} = F_2^5 = 32$ bits. Furthermore, the fast construction of $2^\text{nd}$ order Reed-Muller codes using both $T_1,T_2$ sets of 4-bits, is another method that can efficiently use memory. In addition, this algorithm allows the use of a compact way to represent low order Reed-Muller codes.

In this thesis, the hardware implementation of the “expansion” algorithm for each codeword is presented (for $n=3$). The complexity of this logic circuit is analyzed and we show that the number of two-input exclusive OR gates that are needed for the implementation of our algorithm is $2^n - 4$, where $n$ is the number of variables used. Although exponential in $n$, this is close to minimal mostly because $2^n$ outputs are needed, four of which are driven directly by four inputs and thus, require no gate. Without doubt, this implementation is faster than any common software running on conventional computers.

In addition, it is obvious that this compact representation of $1^\text{st}$ order Reed-Muller codewords is highly memory efficient because it can actually stores a large amount of information in a small word. For example, using a “guide” word of 8 bits, we can
expand, and thus compress a codeword of 1024 bits to a string of 12 bits. The compression ratio in this example is 85:1, and the memory saving is 98.83%.

Reversing the algorithm, we show that not only can we highly compress any codeword of 1st order Reed-Muller codes, and of a new construction of subcodes of 2nd order Reed-Muller codes, but we can also reconstruct a damaged codeword, correcting some errors occurring during the transmission or storage. This new correction method can correct \( n-2 \) errors, or similarly the number of “guide” word bits. It is obvious that there are many cases where our “reverse” algorithm can correct more than \( n-2 \) errors, but we cannot generalize on these cases.

The compressed string includes all the information of expanded codeword. We show that the general compression ratio of this “reverse” algorithm is \( \frac{2^n}{n+1} :1 \), and the memory saving is \( \left(1-\frac{n+1}{2^n}\right)\% \). It is obvious that, for high \( n \), the compression ratio is extremely high. For example, we can imagine the memory saving in a realistic case for \( n=15 \), where we can compress \( 2^{15} = 32768 \) bits to only 16 bits, without losing any information of the codeword. The memory saving in this particular case is 99.94\%. This estimation highlights the importance of this algorithm in environments where memory efficiency is critical.

One of the main contributions of this thesis is the fast generation of a new 2nd order Reed-Muller subcodes of good properties. Even though, this is a non-linear category of subcodes, their low distance \( d \), and some other good properties make them worthy of investigation. Their performance in communication-oriented environments can be simulated, and further investigated in a future work. The coding gain of these subcodes must be simulated. It is also recommended to implement these subcodes in devices used for data storage. The usefulness of these 2nd order Reed-Muller subcodes, might be the minimization of memory errors.

In this thesis, it is proven that the format of Algebraic Normal Form, for our fast construction of subcodes of 2nd order Reed-Muller codes is \( affine + x_{n-1}x_n \). Therefore, the sum of any two codewords is \( affine \) function, and the sum of any three codewords is another codeword.
The applicability of both algorithms can be tested in a variety of environments. For example in digital repeaters, where store and forward process takes place, and where *Automatic Response Request* (ARQ) processes are needed, instead of Forward Error Correction (FEC), codewords can be stored in compact form, and transmitted in full representation. The storage of the compact form can last until the transmitter receives an acknowledgement. On the other hand, in storage processes, both algorithms’ usefulness is indisputable.

In addition, one communication-oriented application that can take advantage of these algorithms, is when, through a process, signal conditions can be measured, and automatically changes the error-correction coding to match current link quality.

In any case, logical extension of this thesis would include a computer simulation on the performance of proposed algorithms, in various operational environments.
LIST OF REFERENCES


INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Professor Jeffrey B. Knorr, Chairman
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, California

4. Professor Dan C. Boger, Chairman
   Department of Applied Mathematics
   Naval Postgraduate School
   Monterey, California

5. Professor Pantelimon Stanica
   Department of Applied Mathematics
   Naval Postgraduate School
   Monterey, California

6. Professor Jon T. Butler
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, California

7. Embassy of Greece
   Office of Army Attaché
   Washington, District of Columbia

8. Cpt Alexopoulos Argyrios
   Hellenic Army General Staff
   Athens, Greece