A Computational Search for Cubic-Like Bent Functions

Jakob Snorrason

Lilya Budaghyan & Samuele Andreoli

Master's Thesis

Department of Informatics
University of Bergen

October 2, 2023

Abstract

Boolean functions are a central topic in computer science. A subset of Boolean functions, Bent Boolean functions, provide optimal resistance to various cryptographical attack vectors, making them an interesting subject for cryptography, as well as many other branches of mathematics and computer science. In this work, we search for cubic Bent Boolean functions using a novel characterization presented by Carlet & Villa in [CV23]. We implement a tool for the search of Bent Boolean functions and cubic-like Bent Boolean functions, allowing for constraints to be set on the form of the ANF of Boolean functions generated by the tool; reducing the search space required for an exhaustive search. The tool guarantees efficient traversal of the search space without redundancies. We use this tool to perform an exhaustive search for cubic-like Bent Boolean functions in dimension 6. This search proves unfeasible for dimension 8 and higher. We further attempt to find novel instances of Bent functions that are not Maioarana-McFarland in dimension 10 but fail to find any interesting results. We conclude that the proposed characterization does not yield a significant enough reduction of the search space to make the classification of cubic Bent Boolean functions of dimensions 8 or higher viable; nor could we use it to produce new instances of cubic Bent Boolean functions in 10 variables.

Contents

1 Introduction **1**

2 Preliminaries **4**

2.1 Definitions 4

2.1.1 Boolean Functions 4

2.1.2 Nonlinearity 7

2.1.3 Bent Boolean Functions 8

2.1.4 Equivalence Relations 8

2.1.5 A Characterization of Cubic-like Bent Functions 9

2.2 Survey 10

2.2.1 Classification of Boolean Functions of Dimension 6 10

2.2.2 Classification of Boolean Functions of Dimension 8 10

2.2.3 Boolean Functions in the Completed Maiorana-MacFarland Class 11

3 Method and Implementation **12**

3.1 Representation 12

3.2 Boolean Function Generation 12

3.2.1 ANF Mask 13

3.3 Tests 15

3.3.1 Nonlinearity Test 16

3.3.2 Constant Derivative Test 16

3.3.3 Further Improvements for Cubic Bent Functions 17

3.4 Notes on Practical Implementation 18

4 Results **21**

4.1 Classification Attempts 22

4.1.1 Cubic Bent Functions of 6 Variables 22

4.1.2 Cubic Bent Functions of 8 Variables 23

4.1.3 Bentable Boolean Functions of Dimension 8 23

4.2 Attempts to Find New Bent Boolean Functions 24
4.2.1 Cubic Bent Functions of 10 Variables 24
4.2.2 Quartic Cubic-Like Bent Functions of 8 Variables 25
4.3 Performance Analysis 25

5 Conclusion **27**

References **29**

A Testing CCZ-Equivalence **31**

B ANF Masks for 10-variables search **32**

Chapter 1

Introduction

Boolean functions, or $(n, 1)$-functions, are functions that take n bits of input and yield a single bit of output.

Boolean functions have a wide range of applications, for instance, in complexity theory, electronic circuits, and, quite notable, in secure and reliable communication. In this thesis, we focus on Boolean functions for cryptographic applications. Indeed, in cryptography, Boolean functions as well as vectorial Boolean functions are the foundation for symmetric ciphers and pseudo-random generators. Vectorial Boolean functions are used as S-Boxes in SPN block ciphers, while Boolean functions are used as filter or mask functions in pseudo-random number generators and stream ciphers [Car21, Chapter 3].

For instance, stream ciphers that use Linear Feedback Shift Registers (LFSR) to generate their keystream can use Boolean functions to increase the keystream's linear complexity. The *Combiner model* relies on an n-variable Boolean function taking input from n LFSRs to generate a keystream with higher resistance to linear attacks than a single LFSR on its own. The *filter model* has an n-variable Boolean function taking n bits from an LFSR state. As with the combiner model, this gives an increase in the linear complexity of the keystream as compared to the LFSR on its own [Car21, Section 1.3.1].

For a Boolean function to effectively increase the linear complexity of a generator, it must have properties suitable for the job. High nonlinearity is an important aspect when selecting a Boolean function to increase the security of a cipher.

A notable attack vector against ciphers is *Linear Cryptanalysis* [Can11]: A study of the linear relations between a cipher's input plaintext, its keystream and its resulting ciphertext. By approximating linear relations between the input and output of a cipher, parts of the key can be recovered. A cipher with high nonlinearity is resistant to this attack, as it cannot be easily approximated using linear functions.

Thus, using functions with optimal nonlinearity is desirable to achieve the best possible re-

sistance against linear cryptanalysis. We call Boolean functions with optimal nonlinearity *Bent* Boolean functions. These functions are ideal in resisting linear cryptanalysis, as the correlation between plaintext and ciphertext decreases.

Unfortunately, Bent Boolean functions are sparse, and their behaviour is hard to predict. An exhaustive search over all Boolean functions is often the only way to find new Bent functions. The number of Boolean functions one must search through increases exponentially in relation to the number of input variables n, at a rate of 2^{2^n}; there are 2^{256} functions of dimension 8. This makes an exhaustive search over every Boolean function – even at a relatively low number of input variables – unfeasible [Car21, Section 1.3].

Thus, a good direction in the search for Bent functions is to try reducing this search space by finding characteristics unique to Bent functions that somehow constrain the form a Bent function might have. One such method is the use of *Equivalence Relations*: Classifying Boolean functions into discrete subclasses that have a common invariant, in this case, bentness. This means that all Boolean functions in the class are guaranteed to be bent, allowing one to generate new Bent functions based on the representative function.

Equivalence relations are useful for characterizing many Boolean functions with a single representative, but we still have to exhaustively search for Boolean functions to classify them into equivalence classes. Reducing the search space of *all* Boolean functions when exhaustively searching through them is what we are interested in.

In [CV23, Proposition 4], we find a proposition about cubic-like Bent Boolean functions that potentially fits this purpose. In this proposition, it is proven that any Bent function $f(x_1, \ldots, x_n)$ can be rewritten as

$$f(x_1, \ldots, x_n) \overset{EA}{\sim} x_1x_2 + x_3x_4 + h(x_1, \ldots, x_n)$$

up to EA-equivalence (which is the most general equivalence relation preserving bentness) . This means that we can limit the search space of h to n-variable Boolean functions not containing multiples of x_1x_2 or x_3x_4.

In this thesis, we implement a tool for finding cubic-like Bent Boolean functions using Carlet & Villa's proposition, and we try to use it for efficient search and classification of Bent Boolean functions.

The thesis is organized as follows: In chapter 2, we introduce definitions of relevant subjects and tools, as well as a survey of fully classified Bent Boolean functions of dimensions 6 and 8, as well as partial classifications of Boolean functions of dimensions 10 and higher. In chapter 3, we describe the implementation of our tool for the search of Bent Boolean functions using Carlet & Villa's proposition and our methodology both for the classification of cubic Bent Boolean functions, and the search for new Boolean functions. In chapter 4, we discuss the computational searches we performed with our tool. First, we give a quick outline of the exhaustive searches of 6 and 8 variables cubic Bent Boolean functions, and an attempt to classify cubic Bent Boolean functions by first searching for all "bentable" Boolean functions. However, we observe that the characterisation

presented in the proposition does not yield a large enough reduction of the search space to effectively search for - and classify - Boolean functions of dimension 8 and higher , even when the search space is further refined by searching for bentable purely cubic Boolean functions first . Finally, we try to generate new instances of quartic cubic-like Bent Boolean functions in 8 variables, and new cubic Bent Boolean functions in 10 variables, with the aim of producing new instances of interesting Bent functions, such as previously unknown Bent functions outside the completed Maiorana-McFarland class. Unfortunately, the exhaustive search using the characterisation in [CV23, Proposition 4] proves unsuitable for this purpose.

Chapter 2

Preliminaries

In this chapter, we introduce all preliminary information relevant to this thesis, defining Boolean functions, Nonlinearity and Bentness, and Equivalence Relationships. We also provide a brief survey of previous findings relevant to the subject of this thesis.

2.1 Definitions

2.1.1 Boolean Functions

Let $\mathbb{F}_2$ denote the finite field with two elements $\{0, 1\}$, and $\mathbb{F}_2^n$ the n-dimensional vector space over $\mathbb{F}_2$. A function f taking n binary inputs and producing a single binary output ($f : \mathbb{F}_2^n \to \mathbb{F}_2$) is called a *Boolean function*. A function $F : \mathbb{F}_2^n \to \mathbb{F}_2$ is called a *vectorial Boolean function* or (n, m)*-function*.

The simplest method for representing a Boolean function is with the use of a Truth Table. A truth table is a listing of all possible inputs of f and their respective outputs [Car21, Section 2.2.1].

Example 2.1.1. *The truth table for a function* $f : \mathbb{F}_2^3 \to \mathbb{F}_2$.

x_1	x_2	x_3	$f(x_1, x_2, x_3)$
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

It is common to agree on an order for the input of the truth table and omit it, allowing one to encode the truth table into a binary string.

Example 2.1.2. *The truth table in Example 2.1.1, with lexicographic order, would be* 10000111.

We also introduce the Hamming weight $\mathrm{w}(f)$ of a Boolean function f as the sum of its positive outputs. Evaluating the Hamming weight of a Boolean function represented as a truth table is trivial, as all outputs are explicitly displayed.

While truth tables are simple, they do not indicate many useful properties of the Boolean function they represent. Their size is also tied to the dimension n of the function, growing exponentially in size relative to n. A more versatile representation is Algebraic Normal Form representation.

2.1.1.1 Algebraic Normal Form

Algebraic Normal Form (ANF) – also known as multivariate representation – is the most common representation used in the context of cryptography and coding [Car21, Section 2.2.1].

The ANF of an n-variable Boolean function $f(x_1, \ldots, x_n)$ is a multivariate polynomial defined as

$$f(x_1, \ldots, x_n) = \bigoplus_{I \subseteq \{1,\ldots,n\}} a_I \left(\prod_{i \in I} x_i \right) = \bigoplus_{I \subseteq \{1,\ldots,n\}} a_I x^I \in \mathbb{F}_2 \left[x_1, \ldots, x_n \right] / \left(x_1^2 \oplus x_1, \ldots, x_n^2 \oplus x_n \right), \tag{2.1}$$

with coefficients $a_I \in \{0, 1\}$.

Example 2.1.3. *Consider the Boolean function f from Example 2.1.1.*
Its ANF would be $f(x_1, x_2, x_3) = 1 + x_1 x_2 + x_3$.

ANF representations, like truth table representation, can be encoded into a binary string by listing the coefficients of each term in the ANF. The ordering of the terms in the binary string encoding must be agreed upon beforehand. For our use-case, the order is reversed; the *last* digit of the binary string represents the first term in $\mathbb{F}_2^n$: the constant term 1. Although this encoding scheme is no more space-efficient than the one used for truth table representation, it makes it easier to implement ANFs and switch between the two representations. It sees extensive use in Chapter 3.

Example 2.1.4. *The ANF in Example 2.1.3 can be encoded by parsing its coefficients to a binary string:*

Index	*Term*	*Coefficient*
1	1	1
2	x_1	0
3	x_2	0
4	x_1x_2	1
5	x_3	1
6	x_1x_3	0
7	x_2x_3	0
8	$x_1x_2x_3$	0

The Boolean function f's ANF would be encoded as 10011000 *using the lexicographic order.*

Converting from ANF to Truth Table representation is trivial. Evaluating the polynomial for each element of $\mathbb{F}_2^n$ for an n-variable Boolean function gives a complete Truth Table. Converting from Truth Table to ANF representation is more complex; the binary Möbius transform, described in [Car21, p. 50] can be used for this purpose.

2.1.1.2 Algebraic Degree

Using ANF representation, the algebraic degree of a Boolean function can be defined as the largest number of variables in any term with a non-zero coefficient [Car21, Section 2.2.1]:

$$d(f) = \max\{|I|;\ a_I \neq 0\}, \tag{2.2}$$

where $|I|$ denotes the size of I.

Example 2.1.5. *The Boolean function $f(x_1, x_2, x_3) = x_1x_2 + x_3 + 1$ has two terms with non-zero coefficients: x_1x_2 and x_3, as well as a constant term* 1*. The term x_1x_2 has a degree of* 2*, x_3 has a degree of* 1*; and the constant term has a degree of* 0*, as it contains no variables. Then, the algebraic degree of the whole function is* 2*.*

The Boolean function f's degree $d(f) = 2$.

The algebraic degree is an important property of a Boolean function, especially when considering implementation. Functions of high algebraic degree are usually harder to implement because more AND gates are necessary to perform the multiplications in the high-order monomials. Moreover, we can define some particular classes of Boolean functions based on their degree, which will be useful throughout this work.

We call *affine*, *quadratic* and *cubic* Boolean functions, all functions that have an algebraic degree equal to or lower than 1, 2 and 3 respectively. Moreover, if an affine function f is such that $f(0) = 0$, we say that f is *linear* [Car21, Section 2.2.1]. We say that a Boolean function is *homogeneous* if all the monomials in its ANF have the same algebraic degree.

2.1.2 Nonlinearity

The bias of a Boolean function is a measure of its output distribution over $\{0,1\}$ [Car21, Section 3.1]:

$$\mathcal{E}(f) = \sum_{x \in \mathbb{F}_2^n} (-1)^{f(x)} = 2^n - 2\mathrm{w}(f). \tag{2.3}$$

A Boolean function is considered *balanced* if its bias $\mathcal{E}(f) = 0$.

The bias of a Boolean function can be evaluated from its truth table by subtracting the number of outputs equal to 1 from the number of outputs equal to 0.

Bias is useful in studying a function's relations with linear and affine functions. Given an n-variable Boolean function f and some affine function ϕ_a, one can study the bias of $f + \phi_a$, that is, the amount of times f and ϕ_a differ from one another – the distance from f to ϕ_a. The bias of $f + \phi_a$ can be expressed by the *Walsh transform* $W_f(a)$ of f, defined as

$$W_f(a) = \mathcal{E}(f + \phi_a) = \sum_{x \in \mathbb{F}_2^n} (-1)^{f(x)+a \cdot x}, \tag{2.4}$$

where $a \in \mathbb{F}_2^n$ [Car21, Definition 16].

With the Walsh transform, we can evaluate the *nonlinearity* of Boolean functions, the minimum Hamming distance between a function and any affine function. The nonlinearity $\mathrm{nl}(f)$ of a Boolean function f is defined as [Car21, Section 3.1.3]

$$\mathrm{nl}(f) = 2^{n-1} - \frac{1}{2} \max_{a \in \mathbb{F}_2^n} \{W_f(a)\}. \tag{2.5}$$

A Boolean function f is perfectly linear if $\mathrm{nl}(f) = 0$.

2.1.2.1 Cryptographic Meaning of Nonlinearity

The nonlinearity of a Boolean function is an important property in regard to cryptographical security. Linear cryptanalysis exploits biased linear relations between the keystream and key in a stream cipher [Can11], meaning that linear functions offer little resistance against such an attack.

A linear function f can be split into component parts $f(x + y) = f(x) + f(y)$, allowing one to study the relation between the two components $f(x)$ and $f(y)$, and correlate the input and output of the function.

Boolean functions with low nonlinearity are similarly vulnerable to linear cryptanalysis, as they can be easily approximated by linear functions. Boolean functions with high nonlinearity are difficult to approximate and are therefore desirable in the construction of secure ciphers.

2.1.3 Bent Boolean Functions

The maximum nonlinearity of any Boolean function is upper bounded [Car21, Theorem 3]. Indeed, for any n-variable Boolean function f, we have

$$\mathrm{nl}(f) \leq 2^{n-1} - 2^{\frac{n}{2}-1}. \tag{2.6}$$

For the bound to be tight, the term $2^{\frac{n}{2}-1}$ must have an integer value. Therefore, only n-variable Boolean functions where n is even can have maximal nonlinearity. These Boolean functions are called *Bent* functions [Car21, Section 3.1.3], and clearly have nonlinearity $\mathrm{nl}(f) = 2^{n-1} - 2^{\frac{n}{2}-1}$. Moreover, it is possible to express Bentness in terms of the Walsh transform of a Boolean function f. Indeed, f is Bent if and only if $W_f(a) = \pm 2^{\frac{n}{2}}$ for every $a \in \mathbb{F}_2^n$ [Car21, Section 5.1.1]. One more interesting result on Bent Boolean functions is that their algebraic degree is upper bounded, that is, for any n-variable Bent Boolean function f, $d(f) \leq \frac{n}{2}$ [Car21, Theorem 13].

Since Bent functions are optimal in the sense of nonlinearity, their search and construction has been investigated a lot. A very popular approach for constructing Bent Boolean functions is the *Maiorana-McFarland Construction*: Let n and r be any positive integers such that $r \leq n$. We call Maiorana-McFarland's function any n-variable Boolean function of the form

$$f(x, y) = x \cdot \phi(y) \oplus g(y); \quad x \in \mathbb{F}_2^r,\ y \in \mathbb{F}_2^{n-r}, \tag{2.7}$$

where ϕ is a permutation from $\mathbb{F}_2^{n-r}$ to $\mathbb{F}_2^r$ and g is a $(n-r)$-variable Boolean function [Car21, Definition 46]. We denote by MM_r the corresponding class. A function is part of the more general MM if it is MM_r for some r. We note that Maiorana-McFarland functions make up the majority of known Bent Boolean functions.

Another class of Bent Boolean functions that we are interested in is called *cubic-like Bent* Boolean functions. First, we need to define the *derivative* of a Boolean function. Let a be a non-zero element in $\mathbb{F}_2^n$, we call the derivative of f in direction a, the Boolean function $D_a f(x) = f(x+a)+f(x)$. We say that an n-variable Boolean function f is cubic-like Bent if, for any non-zero $a \in \mathbb{F}_2^n$, there exists $b \in \mathbb{F}_2^n$ such that the second-order derivative $D_a D_b f = D_b(D_a f)$ is equal to the constant function 1 [CV23, Proposition 1]. If a cubic Boolean function is bent, then it is cubic-like bent [CV23, Proposition 1].

2.1.4 Equivalence Relations

The amount of Boolean functions grows exponentially as the dimension increases, it can therefore be useful to organize them into classes based on equivalence relations that respect interesting cryptographic properties. These equivalence relations allow partitioning the space of Boolean functions in classes. We can then choose a representative function from each class and study the interesting cryptographic properties on the representative alone. We are then ensured that all functions in the class share the same property. We now introduce some equivalence relations that are useful in the study of Boolean functions.

The simplest equivalence relations we define are *linear* and *affine* equivalence. We say that two Boolean functions f and g in n variables are linear equivalent if there exists a linear permutation L over $\mathbb{F}_{2^n}$ such that $g = f \circ L$. If instead of a linear permutation L, we have an affine permutation A such that $g = f \circ A + c$, where $c \in \mathbb{F}_2$ is a constant, then we say that f and g are affine equivalent. Note that if f and g are linear equivalent, they are also affine equivalent [Car21, Definition 5].

The second equivalence we introduce is the *Extended Affine* (EA) equivalence. We say that two Boolean functions f and g are EA-equivalent if there is an affine permutation A over $\mathbb{F}_{2^n}$, and an affine n-variable Boolean function a, such that $g = f \circ A + a$.

The final equivalence relation we introduce is called CCZ-equivalence [Car21, Definition 5]. CCZ-equivalence differs from the previous equivalences as it does not directly work on the functions being classified, but rather on their graphs. The graph of a n-variable Boolean function is defined as the set

$$\mathcal{G}_f = \{(x, y) \in \mathbb{F}_{2^n} \times \mathbb{F}_2 | y = f(x)\}.$$

We say that two Boolean functions f and g are CCZ-equivalent, if there exists an affine permutation $\mathscr{L}$ over $\mathbb{F}_{2^n} \times \mathbb{F}_2$ such that $\mathcal{G}_g = \mathscr{L}(\mathcal{G}_f)$.

Finally, we note that in the case of Boolean functions, two functions are EA equivalent if and only if they are CCZ equivalent [Car21, Section 2.1.1]. This fact is particularly useful to us because it is easy to check two functions for CCZ equivalence using the algorithm presented in [EP09], so we can test two functions for EA equivalence using the fast algorithm that exists to check for CCZ equivalence.

2.1.5 A Characterization of Cubic-like Bent Functions

In [CV23], a characterization of cubic-like Bent Boolean functions up to Extended Affine equivalence is proposed. We report the statement of the proposition here:

Proposition 2.1.1 ([CV23, Proposition 4])**.** *Let $f : \mathbb{F}_2^n \to \mathbb{F}_2$ be a cubic-like bent function, then*

$$f(x_1, \ldots, x_n) \overset{EA}{\sim} x_1x_2 + x_3x_4 + h(x_1, \ldots, x_n), \tag{2.8}$$

where none of the terms of h are a multiple of x_1x_2 or x_3x_4.

The proposition states that the functions f and $x_1x_2 + x_3x_4 + h$ are EA-equivalent. This means that while the two sides of Equivalence 2.8 might not be the same function, they are part of the same EA-equivalence class. Bentness is invariant under EA-equivalence [Car21, Section 6.1.1] , that is, if two functions f and g are EA-equivalent then they are either both bent or they are both not bent. Thus, this characterization can be used to express all EA-equivalence classes that have the cubic-like Bent property.

2.2 Survey

In this section, we present a brief survey of preliminary findings relevant to our thesis. We focus mainly on previously classified cubic Bent Boolean functions in the dimensions we are interested in, namely dimensions 6 and 8.

A complete classification up to EA-equivalence for cubic Bent Boolean functions of 6 variables was given in [Bra+05]. Some preliminary results on the classification of cubic Bent Boolean functions of 8 variables were presented in [Agi05]; completed by Langevin in 2012 [Lan13], although these results were not published outside of Langevin's website as far as the authors can tell. For 10 variables and higher, the classification of n-variable cubic Bent Boolean functions is still an open problem.

2.2.1 Classification of Boolean Functions of Dimension 6

A complete classification up to EA-equivalence of Boolean functions of 6 variables and degree up to 3 was presented in [Bra+05]. In particular, this includes a complete classification of cubic Bent Boolean functions of 6 variables. We report the following excerpt, showing the cubic representatives, using the following notation: $f_n(x_1, \ldots, x_6) = x_1x_2x_3 + x_4$ is denoted by $f_n = 123 + 4$.

f	Cubic terms	Quadratic terms
f_2	123	$16 + 25 + 34$
f_3	$123 + 245$	$13 + 15 + 26 + 34$
f_5	$123 + 245 + 346$	$35 + 26 + 25 + 12$

Table 2.1: EA classes of cubic bent functions of 6 variables [Bra+05, Table 9].

In Table 2.1 the cubic and quadratic terms of the class representatives are separated to reflect the original table in [Bra+05, Table 9]. In order to obtain the complete ANF of the class representative, one only needs to sum the cubic and quadratic terms.

2.2.2 Classification of Boolean Functions of Dimension 8

Boolean functions of 8 variables and degree of up to 3 were completely classified in [Lan13]. As for the case of 6 variables, this also includes a complete classification of cubic Bent Boolean functions of 8 variables. We organize the results from the classification presented in [Lan13] in Table 2.2 following the notation used in Table 2.1, dividing cubic and quadratic terms of the class representatives.

While this separation was of little interest in Table 2.1, it is worth pointing out that for cubic Bent Boolean functions of 8 variables this highlights that some EA classes only differ in their quadratic terms. This is a product of the techniques used in [Bra+05] and [Lan13] for classification. These efforts first consider only Boolean functions with terms of degree three, and find classes of

functions that can be completed to a Bent function using terms of degree 2. Then, they find suitable quadratic terms so that the sum of quadratic and cubic terms is a Bent function and classify the resulting functions.

Class	Cubic terms	Quadratic terms
f_1	124	$45+36+27+18$
f_2	$257+678$	$45+26+37+18$
		$12+14+45+26+37+18$
f_3	$123+124+234+125+235+245+246+$ $346+256+356+127+357+457+367$	$13+34+15+16+38+48+58+78$
		$12+13+34+15+16+38+48+58+78$
f_4	$125+235+145+127+157$	$24+45+36+27+18$
		$34+26+56+27+18$
		$13+14+24+45+27+68$
f_5	$347+267+367+467+578$	$45+36+17+28$

Table 2.2: EA classes of cubic bent functions of 8 variables [Lan13].

2.2.3 Boolean Functions in the Completed Maiorana-MacFarland Class

As mentioned in Section 2.1.3, the Maiorana-McFarland Construction is a method of creating Bent Boolean functions. Until recently, it was not known whether cubic Bent functions outside the completed Maiorana-McFarland class $\mathcal{M}$ existed. However, in [PP20], it was shown that all cubic bent Boolean functions of dimension $n \in \{6,\ 8\}$ can be found in $\mathcal{M}$, while also showing that for cubic bent Boolean functions of dimension $n \geq 10$ this is not the case. In particular, they produce examples of Bent Boolean functions of dimension $n \geq 10$ outside $\mathcal{M}$. Moreover, in a 2023 paper [Pas+23] many more constructions and infinite families of Bent functions that are not in the completed Maiorana-McFarland class are produced.

Chapter 3

Method and Implementation

In this chapter, we describe the implementation of our tool for finding new cubic-like Bent Boolean functions. We describe the implementation of our function generation algorithm, as well as the two Bentness testing approaches implemented.

3.1 Representation

In our implementation, Boolean functions are represented using both ANF and truth table representations, encoded as binary strings or unsigned integers, depending on the situation.

Algebraic Normal Form representation is used mainly in the function generation step of the program. The characterization in [CV23] is expressed in terms of the ANF, so we need to generate Boolean functions in this form. It is also easier to limit the degree of generated functions when represented in ANF form, rather than truth table form.

When testing candidates for Bentness, it is more efficient if the functions are in truth table form, as it makes it easier to compute the function's Walsh transform, or evaluate its derivatives. Both tests implemented in this work require truth table representation to be used.

The two representations can be efficiently converted between using the Fast Möbius transform, shown in Algorithm 1.

3.2 Boolean Function Generation

The implementation splits the search space into n equally sized parts, which are then iterated through in parallel to find Boolean functions that match the given criteria; the amount of parts n depends on the multithreading capabilities of the hardware used. As the program is generally run with certain invariants in mind, i.e., the [CV23, Proposition 4], it would be inefficient to iterate

Input: $(a[i],\ 0 \leq i \leq n^2)$
Output: $b = \mathcal{M}_n(a)$

```
for i ← 0 to 2^n − 1 do
    b[i] ← a[i];
end
for k ← 0 to n do
    for i ← 0 to 2^{n−k} do
        // Compute the image of the i-th 2^k-bit block under M_k. for j ← 0 to
        2^{k−1} − 1 do
            b[2^k i + 2^{k−1} + j] ← b[2^k i + j] + b[2^k i + 2^{k−1} + j]  mod 2;
        end
    end
end
return b;
```

Algorithm 1: Fast Möbius transform

over *all* functions in the search space. To efficiently reduce the search space to only functions that match the initial limits, an *ANF Mask* is used to iterate over the search space.

3.2.1 ANF Mask

The ANF Mask is responsible for iterating through the ANFs for all Boolean functions generated by the implementation. The mask guarantees that every function generated matches parameters specified by the user.

The ANF Mask is primarily composed of two distinct bit-masks: an *enabled* and *disabled* mask.

The enabled mask contains terms that should be enabled for every Boolean function generated by the implementation. For the purposes of this thesis, these terms are usually x_1x_2 and x_3x_4, as specified by the characterization from proposition 2.1.1.

The enabled mask for the terms x_1x_2 and x_3x_4 in bit-string form - leading zeros removed - is `1000000001000`. Note that this representation uses reverse lexicographical ordering.

	x_3x_4	$x_1x_2x_4$	x_2x_4	x_1x_4	x_4	$x_1x_2x_3$	x_2x_3	x_1x_3	x_3	x_1x_2	x_2	x_1	1
$\cdots$	1	0	0	0	0	0	0	0	0	1	0	0	0

Figure 3.1: Enabled mask for terms x_1x_2 and x_3x_4.

For conciseness, we use hexadecimal representation when presenting ANF masks, where each

hexadecimal digit represents four bits, prepending 0's as required:

```
0001 0000 0000 1000 → 0x1008
```

Figure 3.2: Enabled mask from Figure 3.1 in hexadecimal notation

The disabled mask is a combination of two masks. The first mask is a per-term degree limiting mask. Generally, we are not interested in terms of degree higher than 3, as we are searching for cubic functions. We are not interested in linear terms either, as EA-equivalence depends on adding linear terms to functions as part of classification.

	1	x_1	x_2	x_1x_2
1	1	x_1	x_2	x_1x_2
x_3	x_3	x_1x_3	x_2x_3	$x_1x_2x_3$
x_4	x_4	x_1x_4	x_2x_4	$x_1x_2x_4$
x_3x_4	x_3x_4	$x_1x_3x_4$	$x_2x_3x_4$	$x_1x_2x_3x_4$

→

Base 2				Base 16
1	1	1	0	e
1	0	0	0	8
1	0	0	0	8
0	0	0	1	1

→ 0xe881

Table 3.1: Per-term degree limit selection (`min=2, max=3`) for dimension $\mathbb{F}_2^4$.

The second part of the disabled mask is the disabling of multiples of enabled terms. This is useful for the characterization from proposition 2.1.1.

	1	x_1	x_2	x_1x_2
1	1	x_1	x_2	x_1x_2
x_3	x_3	x_1x_3	x_2x_3	$x_1x_2x_3$
x_4	x_4	x_1x_4	x_2x_4	$x_1x_2x_4$
x_3x_4	x_3x_4	$x_1x_3x_4$	$x_2x_3x_4$	$x_1x_2x_3x_4$

→

$\mathbb{Z}_2$				$\mathbb{Z}_{16}$
0	0	0	0	0
0	0	0	1	1
0	0	0	1	1
0	1	1	1	7

→ 0x0117

Table 3.2: Disabled mask selection for multiples of x_1x_2 and x_3x_4 for dimension $\mathbb{F}_2^4$.

Combining the masks from tables 3.1 and 3.2

```
  1110 1000 1000 0001   ( 0xe881 )
| 0000 0001 0001 0111   ( 0x0117 )
= 1110 1001 1001 0111   ( 0xe997 )
```

and *reversing the order* gives us the final disabled mask, which in hexadecimal is `0xe997`.

The last notable part of the ANF Mask is the *minimum function degree mask*. This mask is responsible for ensuring that any generated function has a set minimum degree, while still allowing component terms to be of lower degree. The mask enabled all terms of degree higher or equal to the limit set by the user.

For efficient iteration through the search space, the ANF mask uses the algorithm defined in Algorithm 2. This algorithm guarantees that all generated Boolean functions will contain the fixed

terms set by the enabled mask, and none of the terms disabled by the disabled mask, skipping all invalid ANFs.

Input: The current ANF c, The enabled mask E, The disabled mask D, the minimum function degree mask M
Output: The next valid ANF n

```
do
    n ← c ∨ D;
    n ← n + 1;
    n ← n ∧ ¬D;
    n ← n ∨ E;
while M ≠ 0 and n ∨ M = 0;
return n;
```

Algorithm 2: ANF mask iteration algorithm

We give some insight into the mechanism of Algorithm 2, on how it iterates to the next valid ANF, given the enabled mask E and disabled mask D. First, the algorithm recombines the current ANF n with the disabled mask using an OR operation and increases the obtained value by 1, iterating to the next valid ANF. The disabled mask is then removed from the next ANF, using a logical AND with the negated disabled mask, ensuring that the disabled terms do not appear in the next valid ANF. By incrementing the ANF after combining it with the disabled mask, and then removing the disabled mask, we ensure that the new ANF skips over all invalid increments that may be between the current and the next valid ANF. Finally, since the terms in the enabled mask E should always be a part of the current ANF, we add E using a logical OR operation, since the increment performed in the first step of the algorithm might have flipped some bits related to the enabled terms.

If the minimum degree mask M is set, the algorithm loops until the condition $n \vee M \neq 0$, that is, until the next ANF n contains at least one term enabled in M. Otherwise, the algorithm only iterates once.

3.3 Tests

The implementation contains two methods of testing Boolean functions for Bentness. The first approach is naively computing the nonlinearity of the Boolean function and checking if it satisfies the bound. The second approach involves checking whether the second derivatives of the Boolean function are constantly equal to 1. Note that the second method only works for finding cubic-like bent Boolean functions, and may not hold when searching for generic Boolean functions.

3.3.1 Nonlinearity Test

The first method of testing uses the Walsh transform mentioned in Definition 2.4 to compute the nonlinearity of a generated Boolean function.

With a naive implementation, i.e., directly implementing the algorithm defined in Definition 2.4, the Walsh Transform has a worst-case complexity of $\mathcal{O}(2^{2n})$. We implement the Fast Walsh-Hadamard Transform algorithm as defined in Algorithm 3, which has a worst-case complexity of $\mathcal{O}(2^n n)$. This optimized implementation uses the Butterfly construction, a divide-and-conquer tactic that divides the input into smaller parts that can be computed in parallel. Figure 3.3 shows an example of this construction.

Input: $(a[i],\ 0 \leq i \leq n^2)$
Output: $b = \mathcal{W}_n(a)$

```
for i ← 0 to 2^n − 1 do
    b[i] ← a[i];
end
for k ← 1 to n + 1 do
    for i ← 0 to 2^{n−k} do
        for j ← 0 to 2^{k−1} do
            b[2^k i + j] ← b[2^k i + j] + b[2^k i + 2^{k−1} + j];
            b[2^k i + 2^{k−1} + j] ← b[2^k i + j] − 2b[2^k i + j + 2^{k−1}];
        end
    end
end
return b;
```

Algorithm 3: Fast Walsh-Hadamard Transform

3.3.2 Constant Derivative Test

The second approach is specific to cubic-like Bent Boolean functions. This test relies on checking the condition mentioned in section 2.1.3:

$$\forall a\ \exists b:\ D_a D_b f = 1, \quad a, b \in \mathbb{F}_2^n, \tag{3.1}$$

where f is a Boolean function.

This test has a worst-case complexity of $\mathcal{O}(2^{2n})$, which is worse than Algorithm 3 used in the previous section. It is, however, expected that this test will fail early often, as it fails as soon as

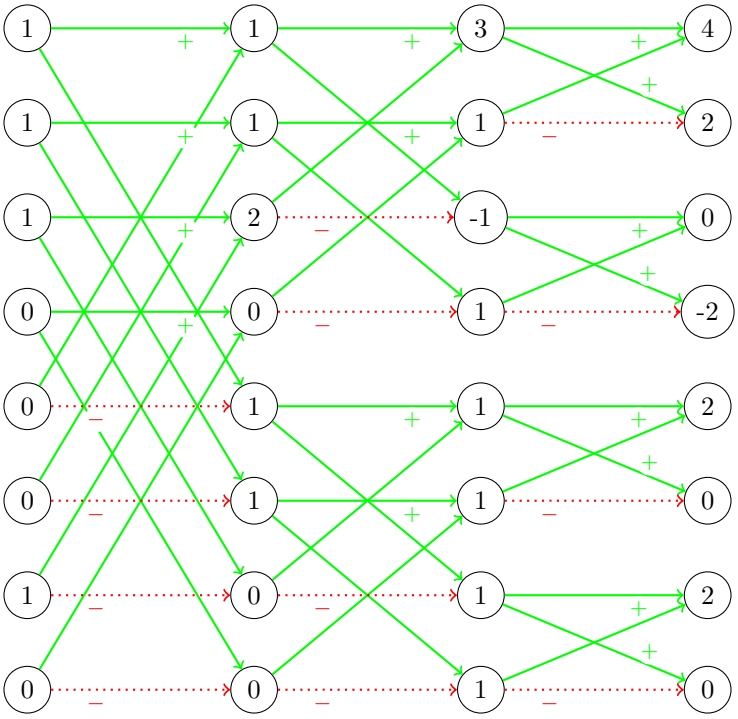

Figure 3.3: Butterfly construction working on input `11100010`

the condition does not hold for any value of a. The complexity of the test can therefore be as little as checking the 2^n values of b for each a.

In particular, up to half of the values of b may be valid for an arbitrary a [CV23, Proposition 6]. Thus, the constant derivative test is often more efficient than testing the nonlinearity of a function, even when utilizing a more efficient implementation of the Walsh Transform.

3.3.3 Further Improvements for Cubic Bent Functions

If we restrict ourselves to searching only for cubic Boolean functions, it is possible to more efficiently traverse the search space. We note that we can divide a function f into cubic and quadratic parts $f = C + Q$, where C are terms of degree 3 and Q terms of degree up to 2. In [LL11, Lemma 1] it is proved that the ANF coefficient of a bent function in 8 variables must respect a number of conditions. In the case of cubic bent functions, the conditions massively simplify to just one. In particular, for any $W \subset \{1, \ldots, 8\}, |W| = 6$, we have that

$$\bigoplus_{|U|=3} a_U a_{W \setminus U} = 0, \tag{3.2}$$

where a_U are coefficients of the ANF as defined in Equation 2.1. We note that the above condition does not depend on the quadratic terms, but only on the cubic terms. Then, we say that a homogeneous cubic function is *bentable* if the condition in Equation 3.2 holds for its ANF.

We are now ready to give a description of our method. The first step of our search involves iterating only over the cubic terms C, excluding terms divisible by x_1x_2 and x_3x_4. Then, if the cubic part is *bentable*, the ANF of the function is stored for the second step of the search. Once all possible cubic C have been examined, we move to the next step. The second step iterates over the bentable candidates we stored in the first step, setting their ANF as the new enabled ANF Mask, and searching for quadratic terms to complete h such that f satisfies Equation (3.1).

3.4 Notes on Practical Implementation

This section briefly outlines some interesting elements of the practical implementation of the tool.

The tool was implemented in the Rust[1] programming language.

The implementation utilizes parallelisation by dividing the search space into n equally sized partitions, where n is the amount of available - or specified - threads on the hardware.

A rudimentary save-load system was implemented, the reasoning being that searches can take several hours. It is convenient to be able to stop a search and be able to resume it at a later time. The save format is simply all parameters used, then the indices of all threads; in the format `(start, end, progress)`.

The implementation can write both ANFs and Truth tables in a variety of formats. It also has a mode for displaying information about specific functions, as shown in listings 3.1, with a demonstration in 3.2.

```
ANF printout mode

Usage: bfsearch.exe anf [OPTIONS] <DIMENSION>

Arguments:
  <DIMENSION>  Dimension of ANF

Options:
  -f, --fixed-terms <ANF>              Fixed terms
  -i, --input-file <FILE PATH>         The format of the ANFs provided in the given file
  -F, --fixed-terms-format <FORMAT>    Format of input fixed terms [default: bitstring] [
                                       possible values: digit, array, bitstring, subscript]
      --ignore-multiples               Do not disable terms that are multiples of fixed
                                       terms
  -t, --input-truth-table              Parse input as a truth table rather than an ANF
  -h, --help                           Print help (see more with '--help')
```

Listing 3.1: ANF display mode parameters

[1] `https://www.rust-lang.org/`

```
> .\bfsearch.exe anf 4 -f "12 + 34" -F digit
x1x2 + x3x4 (
        Dimension              : 4
        Degree                 : 2
        Bias                   : 4
        Algebraic Normal Form  : 0001000000001000
        Truth Table            : 0111100010001000
        Walsh Transform        : [4, 4, 4, -4, 4, 4, 4, -4, 4, 4, 4, -4, -4, -4, -4, 4]
        Walsh Spectrum         : {4: 10, -4: 6}
        Is bent                : true
        Constant derivative    : 1
)
```

Listing 3.2: ANF printout mode showing Boolean function $x_1x_2 + x_3x_4$

Listing 3.3 shows the help screen printed when running `bfsearch.exe search -h`, the exhaustive search mode using Walsh-transform nonlinearity testing to identify Bent Boolean functions (if flag `-b` is set).

```
General search mode

Usage: bfsearch.exe search [OPTIONS] <DIMENSION>

Arguments:
  <DIMENSION>  Dimension to search

Options:
  -m, --min <N>                          Minimum allowed degree
  -M, --max <N>                          Maximum allowed degree
  -l, --no-linear-terms                  Remove linear terms
  -b, --bent_only                        Filter bent
  -f, --fixed-terms <ANF>                Fixed terms
  -i, --input-file <FILE PATH>           The format of the ANFs provided in the given file
  -F, --fixed-terms-format <FORMAT>      Format of input fixed terms [default: bitstring] [
                                         possible values: digit, array, bitstring, subscript]
      --ignore-multiples                 Do not disable terms that are multiples of fixed
                                         terms
  -t, --input-truth-table                Parse input as a truth table rather than an ANF
  -o, --output-path <FILE PATH>          File to output to. Standard output is used by
                                         default
  -O, --output-format <FORMAT>           Format of output [default: bitstring] [possible
                                         values: digit, array, bitstring, subscript]
  -r, --representation <REPR.>           Boolean function representation to output [default:
                                         anf] [possible values: anf, truth-table]
  -T, --output-truth-table               Output truth tables rather than an ANF
  -c, --threadcount <N>                  Amount of threads to allocate for multithreaded
                                         function generation
  -h, --help                             Print help (see more with '--help')
```

Listing 3.3: Search mode parameters

Listing 3.4 shows the help screen printed when running `bfsearch.exe cubic -h`, the exhaustive search mode using the method described in 3.3.3 to find cubic bentable functions. Note that this mode does not allow the user to set allowed degrees, as this mode only works on candidate functions of degree 3.

```
Search for constant cubic bent derivatives

Usage: bfsearch.exe cubic [OPTIONS] <DIMENSION>

Arguments:
  <DIMENSION>  Dimension to search

Options:
  -o, --output-path <FILE PATH>  File to output to. Standard output is used by default
  -O, --output-format <FORMAT>   Format of output [default: bitstring] [possible values:
                                 digit, array, bitstring, subscript]
  -r, --representation <REPR.>   Boolean function representation to output [default: anf]
                                 [possible values: anf, truth-table]
  -T, --output-truth-table       Output truth tables rather than an ANF
  -c, --thread-count <N>         Amount of threads to allocate for multithreaded
                                 function generation
  -h, --help                     Print help (see more with '--help')
```

Chapter 4

Results

In this chapter, we summarize the results of the computational searches we carried out with our tool. All tests were conducted on a machine running Windows 10. Full system specifications are available in Table 4.1.

Motherboard	MS-7C90
CPU	AMD Ryzen 6 5800x 8-Core Processor
GPU	NVIDIA GeForce RTX 4080 Ti
Memory	64 GB DDR4 2300MHz

Table 4.1: Hardware used for computation

We use our tool to iterate over the possible candidates and compile a list of functions with the desired properties. The ANF of functions found using the tool are then stored in CSV files, ready for further processing. For classification attempts, this means compiling a list of functions that are EA-inequivalent. For attempts to generate new instances of cubic or cubic-like Bent Boolean functions, this means checking that a candidate is inequivalent to known cubic or cubic-like bent functions. In both cases, we need a way to test for EA-equivalence. We do so using the algorithm presented in [EP09], remembering that for Boolean functions, EA-equivalence and CCZ-equivalence coincide. For this task, we implement the algorithm in Magma [BCP97]. The code used for the equivalence is available in Appendix A. We run the Magma code on Kepler, a server owned by the computer science department at the University of Bergen. Specifications of the Kepler server are reported in Table 4.2.

Finally, we are ready to see our results in detail in the next few sections.

Motherboard	Dell Inc. Poweredge C4130
CPU	Intel Xeon CPU E5-2690 v4 @ 2.60GHz
GPU	NVIDIA Tesla K80
Memory	512 GB DDR4 2300MHz

Table 4.2: Hardware of the Kepler server

4.1 Classification Attempts

We first attempt to classify cubic Bent Boolean functions to verify that our implementation returns correct data, and potentially find new classification results. We remember that if a Bent function is cubic, then it is also a cubic-like Bent function. So, in our searches, we can test nonlinearity using the constant derivative check from Section 3.3.2 since this is on average quicker than the check using the Walsh transform.

We performed two searches, a search of functions of dimension 6, to verify that our implementation yielded data that matched our survey, and a search of dimension 8, to test whether this approach was efficient enough to search through higher dimensions. We did not attempt a classification of dimension 10, as the exhaustive search of dimension 8 did not return any conclusive results.

4.1.1 Cubic Bent Functions of 6 Variables

We start with an exhaustive search of cubic Boolean functions of dimension 6. Our motivation for this search is to verify that our results match the findings given in Table 2.1.

For this search, we limit the degree of generated functions to 3 as we are only interested in cubic Bent functions. However, we note that for dimension 6 this is not restrictive, as 6-variable Bent functions cannot have degree higher than 3, as mentioned in Section 2.1.3. Moreover, we set a lower per-term limit of 2, as we are not interested in linear terms; through EA-Equivalence classification, all Boolean functions f such that $f = C + Q + L$ are in the same class as $f' = C + Q$.

We fix the terms from the characterization (x_1x_2 and x_3x_4) and disable all of their multiples.

Enabled	`0000000000001008`
Disabled	`fee8f889f889e997`

Table 4.3: ANF Mask used for the search of cubic Boolean functions of dimension 6.

The obtained search space has size 2^{25}. We note that some functions are not going to be tested for bentness since we verify that the degree of the proposed ANF is exactly 3 before checking. The exhaustive search took 41 seconds to complete, and yielded 178,560 functions. We then classified

the functions up to EA-equivalence and verified we obtained the three equivalence classes identified in Table 2.1. The classification task took 5 minutes and 17 seconds on Kepler.

4.1.2 Cubic Bent Functions of 8 Variables

The natural next step was to perform a similar search for Boolean functions of dimension 8. This was done to further verify our results against the survey, but also as a test of the implementation's performance, as the search space for dimension 8 is orders of magnitude larger than the search over functions of dimension 6.

As in the previous test, we fix the terms from the [CV23] characterization in the enabled mask, and set all multiples in the disabled mask.

We also limit the degree of generated functions to a maximum of 3, and a minimum of 2.

Enabled	00000000000000000000000000000000000000000000000000000000000001008
Disabled	ffffffffefffefee8fffefee8fee8f889fffefee8fee8f889fee8f889f889e997

Table 4.4: ANF Mask used for the search of cubic 8 variables Boolean functions.

The obtained search space has size 2^{70}. We remind once again that some functions are going to be excluded immediately at runtime. Unfortunately, this search proved to be unfeasible, as after 11 hours we examined approximately 24 billion functions. We use this fact to estimate that testing one function takes about 1.67 µs on average. Thus, traversing the whole search space of 2^{70} functions would require approximately 62 million years.

As we are most interested in the partially classified dimensions 10 and higher, the speed of our nonlinearity testing implementation is unfeasible. We require a more efficient approach.

4.1.3 Bentable Boolean Functions of Dimension 8

As the naive nonlinearity testing approach failed at higher dimensions, we attempted a two-part search using the method described in Section 3.3.3. The first step involves searching for homogeneous cubic bentable Boolean functions, so we begin with an exhaustive search of all functions composed only of cubic monomials not divisible by x_1x_2 and x_3x_4, to respect the conditions of Proposition 2.1.1. The enabled mask is left empty, as we do not wish to fix any terms in our generated functions. The disabled mask consists of terms that are not of degree 3, and terms of degree 3 that are divisible by x_1x_2 and x_3x_4.

Enabled	00000000000000000000000000000000000000000000000000000000000000000
Disabled	ffffffffefffefee9fffefee9fee9e997fffefee9fee9e997fee9e997e997977f

Table 4.5: ANF mask for the search of 8 variable cubic bentable functions.

Letting the search run for approximately 1 hour yields $3,206,370$ candidate cubic parts. In that time, $2,628,172,637$ functions were evaluated, giving a ratio of 0.122% candidate functions for each evaluated function. Extrapolating the data we have, it would take about 279 days for this search to be completed, yielding an estimated number of $21,462,466,974$ candidate functions. We note, however, that the rate was still slightly dropping when running the experiment. Thus, the actual number of bentable candidates might be lower.

The second step involves using each result from the first step as the enabled mask, in addition to the monomials x_1x_2 and x_3x_4 to respect the conditions of Proposition 2.1.1. Using these enabled masks, we run an exhaustive search using the constant derivative test, as in sections 4.1.1 and 4.1.2, disabling linear terms and terms of too high degree as in the previous searches.

Displayed below is an example ANF Mask using one cubic bentable function acquired in the last search, where $C(x_1, \ldots, x_8) = x_1x_3x_5$ (`200000` in hexadecimal notation).

Enabled	000000000000000000000000000000000000000000000000000000000000000000000000201008
Disabled	`ffffffefffefee8fffefee8fee8e881fffefee8fee8e881fee8e881e8818117`

Table 4.6: ANF mask for the search of 8 variables cubic Bent functions, fixing the cubic terms.

This search, on only one cubic bentable function $x_1x_3x_5$, found $3,555,328$ candidate Bent functions in 2 minutes and 36 seconds.

Extrapolating from this example search, as well as the estimated search time of all candidate bentable cubics above, it would take approximately $106,098$ years to exhaustively search through the bentable functions found in our limited run for the first step, with a search space of 2^{26} for each of the $21,462,466,974$ cubic bentable functions, for a total search space size of about 2^{60}.

4.2 Attempts to Find New Bent Boolean Functions

As the classification of Bent Boolean functions of higher dimensions proved unfeasible, we at least attempt to search for new instances of Bent Boolean functions. In particular, we try to generate new Quartic cubic-like bent Boolean functions in dimension 8, and new cubic bent functions in dimension 10. These searches can be interesting, as they can produce Bent functions outside the completed Maiorana-McFarland class $\mathcal{M}$. As we saw in Section 2.2, the search of Bent functions outside $\mathcal{M}$ is an interesting problem being researched currently.

4.2.1 Cubic Bent Functions of 10 Variables

While a full classification of Bent Boolean functions of 10 variables is not possible with our tool, we are interested in seeing if we can generate any Bent Boolean functions at all. The search parameters are virtually unchanged from the ones we use for the classification searches seen above. We have

a disabled mask removing terms with degree strictly higher than 3, linear terms, and multiples of the terms from the [CV23] characterization. Moreover, the enabled mask only includes the terms x_1x_2 and x_3x_4. We report the full ANF masks in Table B.1 in the Appendix, as they become quite large.

The search space for Boolean functions of dimension 10 contains 2^{147} candidates. After letting this search run for 1 hour, 2 minutes and 21 seconds , we did not get any meaningful results. The search space is simply too large, even after applying the characterisation. Indeed, extrapolating our results, this search would take more than 10^{30} years to explore the whole search space, highlighting how small of a fraction of the search space we can realistically explore.

4.2.2 Quartic Cubic-Like Bent Functions of 8 Variables

In Section 2.2 we mentioned that cubic Bent functions of 8 variables were completely classified in 2012. On the other hand, quartic cubic-like bent functions of 8 variables have not been completely classified. Unfortunately, the search space for quartic cubic-like Bent functions is even larger than the search space for cubic Bent functions. Since we did not manage to classify even cubic Bent functions of 8 variables, a complete classification is beyond our reach even in this scenario. However, as for the previous section, it is still interesting to try generating novel quartic cubic-like Bent functions.

For this search, we exclude terms of degree 5 or higher, and all terms that are divisible by the monomials x_1x_2 and x_3x_4 to respect the condition in Proposition 2.1.1, while imposing that the terms x_1x_2 and x_3x_4 are enabled. We also exclude linear terms for the reasons already explained in Section 4.1.1. The complete enabled and disabled masks are in Table 4.7. Moreover, we fix the minimum function degree to 4, since we are only interested in functions that are at least quartic.

Enabled	`00000000000000000000000000000000000000000000000000000000000001008`
Disabled	`fffefee8fee8f888fee8f888f888f889fee8f888f888f889f888f889f889e997`

Table 4.7: ANF masks for the search of 8 variables quartic cubic-like Bent functions

The obtained search space has size 2^{111}. Once more, we remind that some functions are not going to be tested, since we exclude from our search functions of degree strictly less than 4. We let the search run for 1 hour, 6 minutes and 3 seconds , but we did not get any meaningful results. Extrapolating the estimated total time of this search, we get approximately $3{\cdot}10^{29}$ years to complete, suggesting that the search space is still too large even after applying our characterisation.

4.3 Performance Analysis

In Table 4.8 we summarize the runtime of our tool using different parameters, and the estimated cost of the most expensive operation during a search, namely the check for bentness. Moreover,

we try running slight variants of the classification for dimension 6. First, we try using the Walsh transform to check for bentness as described in Section 3.3.1. Although this check is asymptotically faster, we measure no benefit in the average running time. Moreover, we try to run the test without imposing that the generated functions have a minimum degree. In this case, the running time is faster than the previous tests. However, running the search with these parameters also produces quadratic bent functions, that need to be filtered out after the search.

Dimension	Description	Search time	Search space	Average per function
6	Cubic Bent, Walsh Transform Test	41s	2^{25}	~ 1.22 µs
	Cubic Bent, Constant derivative (CD) Test (Section 4.1.1)	41s	2^{25}	~ 1.22 µs
	Cubic Bent, CD, no minimum function degree	37s	2^{25}	~ 1.12 µs
8	Cubic Bent, CD* (Section 4.1.2)	11h12m26s	2^{70}	~ 1.67 µs
	Cubic bentable* (Section 4.1.3)	1h2m29s	2^{44}	~ 1.37 µs
	Cubic $x_1x_3x_5$ + Q, CD	2m36s	2^{26}	~ 2.50 µs
	Cubic + Quadratic search*	$106,098$y	$\sim 2^{44} + 2^{60}$	N/A
	Quartic cubic-like bent, CD*	1h6m3s	2^{111}	~ 1.22 µs
10	Cubic bent, CD*	1h2m21s	2^{147}	~ 2.00 µs

*Total time estimated in Sections 4.1.2, 4.1.3, 4.2.1, and 4.2.2

Table 4.8: Performance Metrics

Chapter 5

Conclusion

In this thesis, we implemented a tool for exhaustively searching Bent Boolean functions and cubic-like Bent Boolean functions. The tool allows us to set constraints on the ANF of all searched Boolean functions and efficiently evaluate the cryptographic properties of each Boolean function searched.

We decided to test our tool using the characterization proposed in [CV23, Proposition 4] to search for cubic-like Bent Boolean functions, and potentially try to classify cubic Bent Boolean function in 6 or more variables. We confirmed the functionality of the tool by replicating the classification of cubic Bent Boolean functions in dimension 6 presented by [Bra+05, Table 9]. We attempted to replicate the classification of cubic-like Bent Boolean functions presented by [Lan13], but the reduction of the search space yielded by [CV23, Proposition 4], proved to be too insignificant; at least in combination with our implementation and the computational tools at our disposal. Trying to divide the search by considering purely cubic bentable functions first proved to also be insufficient to achieve a classification of 8 variable cubic Bent Boolean functions.

Finally, we tried to use the characterization to generate new instances of cubic Boolean functions in dimension 10, and cubic-like quartic Boolean functions in dimension 8. However, these searches did not generate any interesting results, and we failed to find new instances of Bent Boolean functions in dimensions 8 and 10.

New characterizations that constrain further the ANF of cubic-like Bent Boolean functions may be necessary for this approach to be viable. Finding such constraints is an interesting open problem for the future. Studying the derivatives of cubic bentable Boolean functions might also be investigated as a way to achieve a classification of cubic Boolean functions in 8 or more variables.

References

[Agi05] Sergey Agievich. *On the affine classification of cubic bent functions*. Cryptology ePrint Archive, Paper 2005/044. 2005. URL: https://eprint.iacr.org/2005/044.

[BCP97] Wieb Bosma, John Cannon, and Catherine Playoust. "The Magma algebra system. I. The user language". In: *J. Symbolic Comput.* 24.3-4 (1997). Computational algebra and number theory (London, 1993), pp. 235–265. ISSN: 0747-7171. DOI: 10.1006/jsco.1996.0125. URL: http://dx.doi.org/10.1006/jsco.1996.0125.

[Bra+05] An Braeken et al. "Classification of Boolean Functions of 6 Variables or Less with Respect to Some Cryptographic Properties". In: *Automata, Languages and Programming, 32nd International Colloquium, ICALP 2005, Lisbon, Portugal, July 11-15, 2005, Proceedings*. Ed. by Luís Caires et al. Vol. 3580. Lecture Notes in Computer Science. Springer, 2005, pp. 324–334. DOI: 10.1007/11523468_27. URL: https://doi.org/10.1007/11523468_27.

[Can11] Anne Canteaut. "Linear Cryptanalysis for Stream Ciphers". In: *Encyclopedia of Cryptography and Security*. Ed. by Henk C. A. van Tilborg and Sushil Jajodia. Boston, MA: Springer US, 2011, pp. 725–726. ISBN: 978-1-4419-5906-5. DOI: 10.1007/978-1-4419-5906-5_356. URL: https://doi.org/10.1007/978-1-4419-5906-5_356.

[Car21] Claude Carlet. *Boolean Functions for Cryptography and Coding Theory*. Cambridge University Press, 2021. DOI: 10.1017/9781108606806.

[CV23] Claude Carlet and Irene Villa. "On cubic-like bent Boolean functions". In: *IACR Cryptol. ePrint Arch.* (2023), p. 879. URL: https://eprint.iacr.org/2023/879.

[EP09] Yves Edel and Alexander Pott. "On the Equivalence of Nonlinear Functions". In: *Enhancing Cryptographic Primitives with Techniques from Error Correcting Codes*. Ed. by Bart Preneel et al. Vol. 23. NATO Science for Peace and Security Series - D: Information and Communication Security. IOS Press, 2009, pp. 87–103. DOI: 10.3233/978-1-60750-002-5-87. URL: https://doi.org/10.3233/978-1-60750-002-5-87.

[Lan13] Philippe Langevin. *Classification of Bent Cubics in 8 variables*. 2013. URL: https://langevin.univ-tln.fr/project/bent/bent.html.

[LL11] Philippe Langevin and Gregor Leander. "Counting all bent functions in dimension eight 99270589265934370305785861242880". In: *Des. Codes Cryptogr.* 59.1-3 (2011), pp. 193–205. DOI: 10.1007/s10623-010-9455-z. URL: https://doi.org/10.1007/s10623-010-9455-z.

[Pas+23] Enes Pasalic et al. "Explicit infinite families of bent functions outside the completed Maiorana-McFarland class". In: *Des. Codes Cryptogr.* 91.7 (2023), pp. 2365–2393. DOI: 10.1007/s10623-023-01204-w. URL: https://doi.org/10.1007/s10623-023-01204-w.

[PP20] Alexandr A. Polujan and Alexander Pott. "Cubic bent functions outside the completed Maiorana-McFarland class". In: *Designs, Codes and Cryptography* 88.9 (Sept. 2020), pp. 1701–1722. ISSN: 1573-7586. DOI: 10.1007/s10623-019-00712-y. URL: https://doi.org/10.1007/s10623-019-00712-y.

Appendix A

Testing CCZ-Equivalence

```
// Compute the fixed part for the linear code generator.
// F: (n+1 X 2^n) Matrix over F2. Columns are (1,x) for x in F2^n (lex order)
function CCZ_fixed_code(n)
    F2 := FiniteField(2);

    O := Matrix(F2, 1, 2^n, [1: i in [1..2^n]]);

    // Use VectorSpace to keep the same ordering of variables of C/Rust
    F := Matrix(F2, 2^n, n, [Eltseq(f): f in VectorSpace(F2, n)]);
    TF := Transpose(F);

    return VerticalJoin(O, TF);
end function;

// Check if T1 and T2 are CCZ equivalent
// C is the fixed part of the linear code precomputed with CCZ_fixed_code
//
// Returns true if T1 and T2 are CCZ equivalence, false otherwise
//
function CCZeq(C, T1, T2)
    G1 := VerticalJoin(C, Matrix(FiniteField(2), 1, Ncols(C), T1));
    G2 := VerticalJoin(C, Matrix(FiniteField(2), 1, Ncols(C), T2));
    L1 := LinearCode(G1);
    L2 := LinearCode(G2);

    r := IsIsomorphic(L1, L2);

    return r;
end function;

// Example usage
n := 3;
T1 := [0,1,0,1,0,1,0,1];
T2 := [1,0,1,0,1,0,1,0];

C = CCZ_fixed_code(n);
are_T1_T2_equivalent := CCZeq(C, T1, T2);
```

Appendix B

ANF Masks for 10-variables search

We report here the full enabled and disabled masks used for the search of 10-variables cubic-like Bent functions.

Enabled	...00000000000000000000000000000000000000000000001008
Disabled	5989fb5839895ab51077a96787cc23e9ebbe0ae4a5d7615c5 f8934cf5379ad61d7a3bd2d3a4b8b6a5a0cdd09410612c356 036507cdf6fcad94385e72ded7f2034f562f22988f60b8573 86b068cc793624725f9d8b8e523f34321ff99a76c42ce36f9 d6176b54090a31fc9c08ecd728339b42ffaa7cf28e3b2e24e cf61b3a3c3b710094b444c37d9c52643d5f004fb5a5690078 c54cfea8f8b2629d5ecdc3fae7aa09cd7393490d1d354faaf bfd644004befdd7992fa515b9f44a39b5a25fe86c6767dc08 cd15b62164687afd11f2c0890b3f97bfe33f2d4c88460e43e fb3bd079d09b736306bcceb73dd6209aa4dabd9a1db1006a4 e1d34abff8ff6e31d9fde36b2a94ea3df0057c2bcb42b59f9 c51af7004a3b3ef51292dc1965c59aad0c3d050f0ba8962a8 9fc114e4e3b673c6e078f48e0a261d85369a9873b5ee67abb 2cef11f6b28741da1524560ff42db6452fb8724fe2931f807 70a98837851699035f43e24f86e3d5e78d1b460a650eb70b2 d607f80bd21f4bb9f173b0ef9cc8dbc6c5af9bf6e7843ac37 cafa9e5e4a86564a2cea0e9149db72e075451f7ea46e0c010 49fa97e459ae1e6ff

Table B.1: ANF Mask used for search of cubic Boolean functions of dimension 10.