Thickness Distribution of Boolean Functions in 4 and 5 Variables

Mathias Hopp

Master's thesis in Software Engineering at

Department of Computing, Mathematics and Physics,
Bergen University College

Department of Informatics,
University of Bergen

May 2020

Abstract

This thesis explores the distribution of algebraic thickness of Boolean functions in four and five variables, that is, the minimum number of terms in the ANF of the functions in the orbit of a Boolean function, through all affine transformations. The calculation is completed computationally, and the designed programs are explained thoroughly, and listed as appendices in full. A class of Boolean functions is defined, the *rigid functions*, that is relevant to algebraic thickness, and – as will be shown – is very useful in revealing the algebraic thickness distribution. From rigid functions within the same orbit, the minimum function is chosen as a *representative*, and the method of this choice is presented. Additionally, a complete analysis of some complexity properties (e.g., nonlinearity) of all relevant orbits of Boolean functions is calculated and listed, with comparisons to a lower number of variables. Some properties of these rigid functions are also presented, and proven.

Acknowledgements

First of all, I want to thank my supervisors, Constanza S. Riera and Pål Ellingsen, for their guidance, support, and help in understanding and solving this problem, and in writing this thesis. Next, I want to give a special thank you to Pantelimon Stănică, for taking an interest in my project, for all the time and energy he spent on aiding me, and for fruitful discussions. Furthermore, thank you to all my friends and family for allowing me to rant about Boolean functions and their importance, for pushing me when I needed it, and for supporting me through this project. Specifically, I would like to thank Stine, Anne-Marit, Mats, Ane, Annar, and Sigurd, for listening to me and for all your advice. Thank you to Yngve, Erlend, Hannah, Oliver, and Thomas, for sharing your experiences, for our lively discussions, and for your friendship. Thank you, Gunhild, for convincing me to start this project and for believing that I would succeed.

And finally, thank you to my father, Ole Kristian, who did not get to see me finish, but who I know would be proud, and joyful for me, and who I will always carry with me.

Thank you.

Contents

1 Introduction **1**
- 1.1 Problem Description 4
- 1.2 Thesis Outline 4

2 Theoretical Background **6**
- 2.1 Mathematical Foundation 6
  - 2.1.1 Set Theory 6
  - 2.1.2 Combinatorics 9
  - 2.1.3 Abstract Algebra 10
  - 2.1.4 Linear Algebra 11
  - 2.1.5 Boolean Algebra 13
  - 2.1.6 Cryptography 14
- 2.2 Boolean Functions 16
- 2.3 Related Work 23
  - 2.3.1 Algebraic Thickness 23
  - 2.3.2 Known Bounds on Algebraic Thickness 24
- 2.4 Methodology 25

3 Calculating thickness distribution for $n = 4$ **26**
- 3.1 SageMath and computational strategy 26
- 3.2 Program: Brute-force implementation for $n = 2,3,4$ 28
  - 3.2.1 Constructing all Boolean functions 28
  - 3.2.2 Generating invertible matrices and vectors 29
  - 3.2.3 Calculating Algebraic Thickness, $n = 2,3,4$ 32

4 Calculating thickness distribution for $n = 5$ **35**
- 4.1 Rigid functions 35
  - 4.1.1 Examples of rigid functions 38
- 4.2 Representative functions 39
  - 4.2.1 Examples of choosing representative functions 40
- 4.3 Focusing on monomial counts 41
- 4.4 Program: Finding representatives in $n = 5$ 44
  - 4.4.1 Constructing relevant Boolean functions 45
  - 4.4.2 Generating invertible matrices and vectors 45
  - 4.4.3 Searching for representatives in $n = 5$ 46
  - 4.4.4 Execution time 49

5 Analysis and Assessment **50**
5.1 Results and analysis for $n = 4$ 53
5.1.1 Property distribution in $n = 4$ 56
5.1.2 Bent functions in $n = 4$ 59
5.1.3 Balanced functions in $n = 4$ 60
5.2 Results and analysis for $n = 5$ 61
5.2.1 Property distribution in $n = 5$ 61
5.2.2 Semi-Bent functions in $n = 5$ 66
5.2.3 Balanced functions in $n = 5$ 67
5.2.4 Functions with maximum thickness in $n = 5$ 69
5.2.5 Details of orbit lengths in $n = 5$ 70
5.3 General Results and Analysis 72
5.3.1 Symmetric Property of Thickness Distribution 72
5.4 Validity of shown programs and results 74

6 Conclusions **75**
6.1 Further Work 77
6.1.1 Rotation Symmetric functions 77
6.1.2 Thickness Sequences 78
6.1.3 Orbit lengths and function pairs 79
6.1.4 Generating representative functions 80
6.1.5 Similar implementation for $n > 5$ 81
6.1.6 Conjectures 82

A Data: $n = 2$ Raw Data **86**

B Program: Brute-force calculation, $n = 4$ **87**

C Program: Representatives collection for $n = 5$ **90**

D List: $x_1x_2x_3$-form of representatives **94**

E List: Representatives in $n = 4$ **96**

F List: Representatives in $n = 5$ **98**

List of Figures

2.1 Representation of an $m \times n$ matrix 12
2.2 Identity matrix I_3 13
2.3 Example: a simple LFSR 14

3.1 Code: Variable declaration of programs 28
3.2 Code: Generation of Boolean functions 29
3.3 Code: Dividing number of functions into iterations, $n \leq 4$ 29
3.4 Code: Matrix maps and vector generation in $n \leq 4$ 31
3.5 Code: Calculating thickness distribution for $n = 4$ 34

4.1 Code: Mapping a function into its lowest indexed form 45
4.2 Code: Matrix maps and vector generation in $n = 5$ 46
4.3 Code: Method for calculating thickness in $n = 5$ 48

5.1 Upper- and lower bound of maximum $\mathcal{T}$ vs. Fibonacci 51
5.2 Distribution of representatives in $n = 2, 3, 4$ 72
5.3 Distribution of representatives in $n = 5$ 73

List of Tables

2.1 Examples of named Boolean functions . . . 13
2.2 Distribution of Boolean functions in n variables . . . 14
2.3 Basic operators for Boolean functions . . . 16
2.4 Number of affine transformations in n variables . . . 19

3.1 *Brute-force:* number of mapped functions in n variables . . . 32

4.1 Number of rigid functions in $n \leq 5$ variables . . . 37
4.2 Number of equivalence classes in the affine group . . . 39
4.3 Distribution of monomials in $n \leq 5$ variables . . . 43
4.4 Distribution of variable-permutation unique functions in $\mathcal{B}_5$ . . . 44
4.5 Execution time of iterations . . . 49

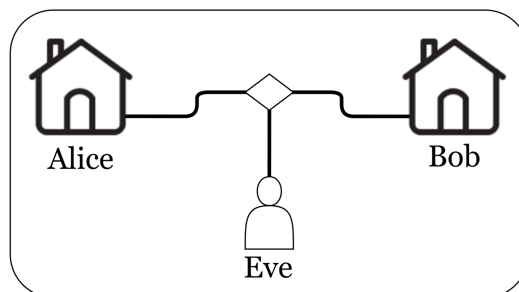
5.1 Distribution of representatives, $n \leq 5$ . . . 50
5.2 Distribution of number of $f \in \mathcal{B}_n$ with given $\mathcal{N}$-value, $n \leq 5$ . . . 52
5.3 Distribution of number of orbits with given $\mathcal{N}$-value, $n \leq 5$ . . . 53
5.4 All representatives in $n \leq 4$ . . . 55
5.5 Property distribution: $\mathcal{T} = 1$, $n = 4$ . . . 56
5.6 Property distribution: $\mathcal{T} = 2$, $n = 4$ . . . 57
5.7 Property distribution: $\mathcal{T} = 3$, $n = 4$ . . . 57
5.8 Property distribution: $\mathcal{T} = 4$, $n = 4$ . . . 57
5.9 Property distribution: $\mathcal{T} = 5$, $n = 4$ . . . 58
5.10 Property distribution: Summary, $n = 4$ . . . 58
5.11 Property distribution: $\mathcal{T} = 1$, $n = 5$ . . . 61
5.12 Property distribution: $\mathcal{T} = 2$, $n = 5$ . . . 62
5.13 Property distribution: $\mathcal{T} = 3$, $n = 5$ . . . 62
5.14 Property distribution: $\mathcal{T} = 4$, $n = 5$ . . . 63
5.15 Property distribution: $\mathcal{T} = 5$, $n = 5$ . . . 63
5.16 Property distribution: $\mathcal{T} = 6$, $n = 5$ . . . 64
5.17 Property distribution: $\mathcal{T} = 7$, $n = 5$ . . . 64
5.18 Property distribution: $\mathcal{T} = 8$, $n = 5$ . . . 65
5.19 Property distribution: Summary, $n = 5$ . . . 65
5.20 All semi-bent representatives in $n = 5$. . . . 66
5.21 All balanced orbits in $n = 5$. . . . 68
5.22 All orbits with maximum orbit length . . . 70
5.23 The 52 unique orbit lengths in $n = 5$ . . . 71

6.1 Relisting of Table 5.1 . . . 75

Chapter 1

Introduction

Let us say that Alice and Bob want to send each other messages, and they do not want anyone else to know the content of them. To do this, they put a cable between their houses and send the messages through it. Say, then, that there is a third person – Eve – who is very nosy, and who sees the cable connecting the two houses. Eve knows cables, and manages to join a new cable with the existing one, such that anything sent through the cable also comes out on Eve's side, meaning Eve now can see any message sent.



In order for Alice and Bob to avoid Eve being able to read their messages, they can *encrypt* the information, by following a very specific set of rules that they themselves agreed upon, before transmission. Upon receiving a message, they can then *decrypt* it, by (for instance, depending on what rules they chose) following this rule set in the reverse order, revealing the original information. If the encryption is efficient, Eve may have a hard time in discovering the secret message.

If, however, the rules that Alice and Bob uses are too simple, Eve can deploy an *attack*, which could expose the message.

The concepts introduced here are the main motivators for a field of mathematics called *Cryptology*, which can be divided into *Cryptanalysis* and *Cryptography*, and envelops, among other subjects, *secure communication* through unsecure channels – this may be over the internet, as letters, or any other sort of indirect communication form. This thesis is centered around some important concepts within Cryptography.

Throughout history, there has been a need for secure communication between various parties, and the methods used to achieve this have been improved upon continuously – from 2000 B.C., when the Egyptians used non-standard "secret" hieroglyphs, to Roman times using the famous *Caesar cipher* , up to today – and they are always evolving [21]. In modern society, secure channels for communications are used multiple times each day, by most people – be it over phone lines or the internet; on smart phones, or computers.

A message without any encryption is referred to as the *plaintext*, which is readable to anyone who knows the language it is written in. After encryption, the resulting encrypted message is called the *ciphertext*, and may consist of any symbols, depending on the encryption method used. When the intended recipient receives the ciphertext, they need to decrypt it, and to do this, a *key* is often used. This key will contain, in some form, the correct configuration of the encryption method, making the translation from ciphertext back to plaintext simple.

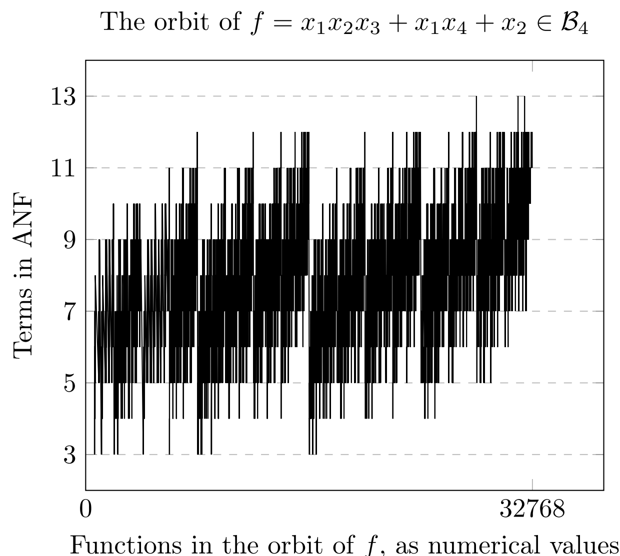
If someone is *eavesdropping*, as Eve did in the text above, they may be able to collect the ciphertext and attempt to decrypt it without the original senders awareness. If the methods of encryption are not complex enough, or if the number of possible solutions is too small, a *brute-force attack* can be attempted, which means that all possible configurations of the methods used are applied to the ciphertext. One of the principles of modern cryptography is to guarantee that using such a brute-force attack would take too long, effectively making it impossible to attempt.

The techniques we use today for achieving secure communication are, in general, all following the fundamental concepts coined by Claude Shannon: *confusion* and *diffusion* (cf. [10, 21]). The aim of confusion is to conceal the structure of the system, attempting to avoid letting parts of the system be exploitable by attackers. Diffusion ensures that a small change in the plaintext or in the key spreads out over the whole ciphertext.

The study of *Boolean functions* in relation to Cryptography has been of interest since they were introduced for use in combination with linear feedback shift registers (LFSRs). They are applicable and central in several cryptographic algorithms today, both in stream- and block ciphers – represented by combinations of LFSRs and Substitution boxes (S-boxes, cf. AES, DES, and more). Furthermore, it has been shown that they provide both confusion and diffusion in cryptographic systems [10]. Confusion is achieved by the *complexity* of the related functions, which can be described by several properties, but this thesis is related to two main concepts: the *algebraic degree* and *nonlinearity* of said functions. These criteria describe, in their own way, the difference between any given function and the *affine* functions – that is, linear functions with or without a constant – as Claude Carlet phrased it, in [9]. Affine functions are considered ineffective for cryptographic purposes, and should be avoided as much as possible – in fact, Carlet states that *all cryptographic functions must have high algebraic degree and high nonlinearity.*

As an addition to these known complexity criteria, Carlet defined – in [8] – the concept of *algebraic thickness*, referring to work by W. Meier and O. Staffelbach in [19]. In noting that some functions with a high number of terms in

their corresponding algebraic normal form (ANF)[1] behave similarly to functions with few terms, Carlet states that the number of terms of a function is not itself a satisfactory complexity criteria. Rather, the function with the lowest amount of terms, in its orbit through affine transformations, is of interest, and has implications for all other functions in its orbit.



The continued application of all affine transformations in n variables to a given function $f \in \mathcal{B}_n$ (*where* $\mathcal{B}_n$ *is the set of all Boolean functions in* n *variables*), reveals the orbit of f. Given in the figure above is an illustration of the number of terms in the ANF of *all functions* in the orbit of $f = x_1x_2x_3 + x_1x_4 + x_2$ – the largest orbit of $\mathcal{B}_4$. There are 10 080 functions plotted in this graph, with a wide range of terms (lowest is three, highest is thirteen) – but the functions of interest here are at the bottom of the graph. These are the functions in the orbit of f with the lowest number of terms, named in this thesis as the *rigid functions*, which determine the algebraic thickness of the orbit, and – as will be explained – can all be *represented* by f.

The concept of algebraic thickness of Boolean functions is the main subject of this thesis, and will be explored thoroughly for *all* Boolean functions in four and five variables, with implications for further study in larger numbers of variables. In the following text, the definitions and concepts used are first defined, then used in practice to design two programs in Python. After collection of the results of the first program, a connection to work done by Harrison in the 1960s (cf. [16]) was found, and is used to further discuss all orbits of Boolean functions in the relevant variables.

[1] This concept, as well as the other technical terms in this section, are defined in Section 2.2.

1.1 Problem Description

As mentioned, and which will be more rigorously defined (in Section 2.2), this thesis appertains to the concept of algebraic thickness – i.e. the minimum number of terms of all Boolean functions in the orbit of any one Boolean function, through all affine transformations – and aims to reveal the distribution of algebraic thickness of all Boolean functions in four and five variables.

Specifically, when the project was first started, the main goal was to iterate through *every* Boolean function $f(x_1, \ldots, x_n)$ in $n = 4, 5$ variables, and check the number of terms in the ANF of $f(A(x_1, \ldots, x_n)+\mathbf{b})$, for every $n \times n$ invertible matrix A and every vector $\mathbf{b}$ – in other words, checking every invertible affine transformation of f.

The minimum number of terms in the transformations of f is the algebraic thickness $\mathcal{T}(f)$ of f, as defined by Carlet in [8, 4], given here in Definition 2.10. With an interest for determining patterns that could relate to a higher number of variables (than $n = 4, 5$), the specific A and $\mathbf{b}$ that maps f to a function with the minimum number of terms should be stored as well. A complete distribution for every f would then be the outcome of the thesis.

As will be discussed in the coming chapters, by using an exhaustive search, the calculation of this distribution for $n \leq 4$ variables is at best a trivial, and at worst a lengthy – but manageable – assignment. There are 2^{2^n} Boolean functions in n variables, which means, for $n = 4$, this number is 65 536. This is not a large number in terms of computer science, but, given that there are 322 560 different affine transformations needed to be checked for *each* Boolean function, the calculation of algebraic thickness for all Boolean functions in four variables – when visiting all functions – is a time consuming task. The result of this iteration would then be a set of 2^{2^4} data points to be analysed for patterns and validity.

However, in moving from four to five variables, this number grows more than exponentially. The total number of unique Boolean functions is 4 294 967 296, and the number of different affine transformations is 319 979 520. One of the sub-goals of the thesis was to find an efficient method able to handle the magnitude of the product of these two large numbers, and another was to effectively handle and analyse the resulting data set of $n = 5$.

The concepts and numbers mentioned here, and in the introduction above, will be further explained, defined, and discussed (mainly) in Chapter 2. Additionally, throughout the thesis, when discussing functions $n \leq 5$, we omit the trivial cases $n = 0, 1$, unless specified.

1.2 Thesis Outline

In the interest of this thesis being self-contained, Chapter 2 serves as a compilation of the definitions relevant to – and used in – this thesis, based on the definitions and explanations of the cited books. Section 2.1 introduces basic concepts within a variety of fields of mathematics (a similar foundation can be seen in [2]), which builds the foundation for the definitions given in Section 2.2, as well as other sections throughout the thesis.

Section 2.3 mentions the related works and known results from scientific literature, and in Section 2.4 the methodology of relevance to the thesis is discussed briefly.

Chapter 3 contains a description and discussion of the brute-force program used to calculate the full distribution of algebraic thickness in $n = 4$ variables, which includes the various methods of the program used to accomplish this. The methods of importance are explained in detail, and the full program is given in Appendix B.

Next, in Chapter 4, the results from Chapter 3 are discussed relative to extension from four to five variables. From pattern analysis of these, two concepts are defined: *rigid functions* and *representative functions*. The proven properties of these concepts are used in the design of a program (in Section 4.4) able to calculate the full distribution of algebraic thickness in $n = 5$ variables, in a reasonable time. After the program has been discussed in full (i.e., the important methods, the full program is given in Appendix C), the execution time of the program is listed, and a comparison to the brute-force method is made.

The results of the listed programs, and analysis thereof, are gathered in Chapter 5, including a property distribution overview sorted by algebraic thickness, for Boolean functions in both $n = 4$ (Section 5.1) and $n = 5$ (Section 5.2) variables.

Finally, in Chapter 6, a summary of all findings is given, with a concluding report. Further work that may be of interest is given in Section 6.1, including concepts that could be expanded upon, or comments on further implementation. At the end, in Section 6.1.6, a discussion about possible properties related to algebraic thickness that we could not prove at this time, is presented as conjectures.

The appendices of this thesis contain information deemed of interest. In Appendix A, the full data set of $n = 2$ is given, calculated by the brute-force program described in Chapter 3 – included to show how the results of the program were stored. The same data sets for $n = 3, 4$ are not included, because of their respective size. (The data set for functions in 3 variables could arguably be listed, as there are only 254 lines – however, we believe the important data has been summarized adequately in the given property analysis, in Chapter 5.)

As mentioned above, Appendices B and C contain the full programs described in Chapters 3 and 4, respectively. These programs are exactly as they were when they were run for data collection purposes, except for the removal of some method calls for printing and saving of position data, and time elapsed. This is further specified in the respective chapters.

Appendices D, E and F contain full lists of the representative functions, as defined in Section 4.2, which are discussed in Section 5.1 and 5.2. The representative functions are of vital importance to this thesis, and allow discussion of the distribution of algebraic thickness.

Chapter 2

Theoretical Background

2.1 Mathematical Foundation

2.1.1 Set Theory: the basis for modern mathematics

As a general basis for this thesis, set theory – and its concepts – will be used, as it is considered to be the best approach to a bedrock of modern mathematics, and is therefore widely used (other approaches exist, e.g. category theory). Technically, a distinction is made between *axiomatic* set theory (introduced by Zermelo and Fraenkel, also called 'contemporary set theory') and *naïve* set theory, wherein the former avoids some paradoxes (cf. the Barber Paradox, proposed by Bertrand Russel) which the latter fails to accommodate. However, for the intents and purposes of this thesis, the naïve approach gives an intuitive understanding of sets, which will suffice in this case – for more on the axiomatic approach, refer to [11]. The following section is a brief explanation of the concepts that will be used further in the text. All the definitions are taken from [15], which is recommended for a more complete introduction to set theory and its applications.

A *set* – in simple terms – is a collection of objects of any kind, which are referred to as the *elements* of the set. These elements are in no particular order, and each element in the set is *unique*, meaning there is only one of each element within the set. Although a set can contain anything, most sets used in this thesis will be comprised of numbers, or other mathematical concepts. A set is usually denoted by a capital letter for reference purposes, and listing all elements in a given set is accomplished by separating each element by commas, and containing them within two braces $\{\ \}$; e.g. $A = \{1, 2, 3\}$, which is the set containing the numbers 1, 2, and 3. We say *the* set, because any *other* set $B = \{1, 2, 3\}$ that contains all and only the same elements as in A, will be equal (i.e. $A = B$), and therefore they are the same set.

Claiming an object is contained within a set is one of the fundamental statements in set theory, and is represented by $\in$, e.g. for a set $A = \{a\}$ where a and b are objects, and $a \neq b$: $a \in A$ is true, but $b \in A$ is not true (the latter case is denoted $b \notin A$). Notation for the generation of sets, where the elements in the set have certain properties, is commonly written as $\{\text{property of } x \mid \text{set containing } x\}$, or similar – as should be understood through context. E.g. $\{x < 3 \mid x \in \{1, 2, 3, 4\}\} = \{1, 2\}$.

Example 2.1. Some commonly used sets in mathematics are

$$
\begin{aligned}
\text{the } \textit{natural numbers } \mathbb{N} &= \{1,2,3,\dots\}, \\
\text{the } \textit{integers } \mathbb{Z} &= \{\dots,-2,-1,0,1,2,\dots\}, \\
\text{the } \textit{rational numbers } \mathbb{Q} &= \left\{\frac{a}{b} \mid a,b \in \mathbb{Z}\right\}, \text{ and} \\
\text{the } \textit{real} \text{ numbers } \mathbb{R}&,
\end{aligned}
$$

where the latter includes both the rational and *irrational* numbers (e.g. π, $\sqrt{2}$, etc.), that is, $\mathbb{R}$ contains all decimals. Note that some texts may include 0 in $\mathbb{N}$.

Two sets can be combined to form a third, either by use of the union $\cup$ operation – the new set will contain all elements in the first set and all elements in the second – or by the intersection operation $\cap$ – the new set will contain all elements present in both sets.

A set can also be a *subset* of another set, written $A \subseteq B$, if all elements in A are in B. If $A \subseteq B$ and $B \subseteq A$, then $A = B$. In the case that $A \subseteq B$ and it is known that $A \neq B$, this may be denoted by $A \subset B$, i.e. A is a *strict* subset of B. The notation $A \subsetneq B$ can also be used for this, in the case that it is important to mark that $A \neq B$.

The *Cartesian product* of sets, denoted by $A \times B$, forms a new set of what is called *tuples* (a, b) where $a \in A, b \in B$, such that each element of A is in a tuple with each element of B. Tuples are denoted by the use of parentheses, and show a *relation* between elements in sets, a concept used in several fields of mathematics. The Cartesian product $A \times A$ is often referred to as A^2, and in general $A \times \cdots \times A = A^n$ when referring to the n-th Cartesian product of A.

Example 2.2. Let a, b, c be objects, and let $A = \{a, b\}$, $B = \{b, c\}$, and $C = \{c\}$ be sets. Then,

$$
\begin{aligned}
A \cup B &= \{a, b, c\}, \\
A \cap B &= \{b\}, \\
A \times C &= \{(a, c), (b, c)\}.
\end{aligned}
$$

Also, $C \subseteq B$ and $C \subseteq A \cup B$, but $C \nsubseteq A \cap B$.

The size of a set A is often written $|A|$ (called the *cardinality* of A), and is equal to the number of objects within the set. Some sets are of infinite size (both countable and uncountable), but this thesis will (in most cases) work with sets of finite size. The cardinalities of the resulting sets after application of the operations discussed above are, for $|A| = n$, $|B| = m$, where $n, m \in \mathbb{N} \cup \{0\}$:

- $\max(n, m) \leq |A \cup B| \leq n + m$,
- $0 \leq |A \cap B| \leq \max(n, m)$, and
- $|A \times B| = nm$,

where *max* takes any set of comparable objects and outputs the maximum object.

When mathematical operators (e.g. $\cup$ and $\cap$, but also addition and multiplication, with more) are defined on a set, there are a number of properties these

operators may have, of which three will be listed here, as they are relevant to definitions further down in the text. Let $+, \star$ be two operators defined on a set S, and let $a, b, c \in S$.

1. If $+$ is *associative*, then $a + (b + c) = (a + b) + c$.
2. If $+$ is commutative, then $a + b = b + a$.
3. Finally, the *left* and *right distributive laws* hold if

   $a \star (b + c) = (a \star b) + (a \star c)$ and
   $(b + c) \star a = (b \star a) + (c \star a)$, respectively.

A specific subset of the relations mentioned above is used in much of mathematics, with some restrictions imposed on the tuples that can occur. In the definition below, a *binary relation*, as used here, refers to the fact that there are only *two* elements in the relation.

Definition 2.1. (Function) [15]
A *function* from a set A to a set B is a binary relation in which *every* element of A is associated with a *uniquely specified* element of B. In other words, for each $a \in A$, there is precisely one pair of the form (a, b).

In mathematical notation, a function f from a set A to a set B may be expressed as

$$f : A \to B,$$

where A, in this case, is called the *domain* of the function f, and B is called the *co-domain*. The *range* of f is the set of images of all the elements of A under f, where the image of an element of A under f is an element of B – defined $f(A) = \{f(x) \mid x \in A\}$.

There are some properties of functions that are important to note. For a function $f : A \to B$, where $a_1, a_2 \in A$, f is called *injective* (or *one-to-one*) when $f(a_1) = f(a_2)$ implies $a_1 = a_2$. If the range of f is equal to the co-domain of f, then f is called *surjective* (or *onto*). If f is both injective and surjective, f is called *bijective*, and then f is *invertible* – i.e. there exists a function $f^{-1} : B \to A$ such that $f^{-1}(f(a)) = a$ and $f(f^{-1}(b)) = b$. The function f^{-1} is called the *inverse* of f.

Given two functions $f : A \to B$ and $g : B \to C$, the composite relation $g \circ f$ between A and C consists of pairs of the form (a, c), where $a \in A, c \in C$, such that, for some $b \in B$, $(a, b) \in f$ and $(b, c) \in g$. Since both of f, g are functions, $b = f(a)$ is uniquely determined by a, and $c = g(b)$ is uniquely determined by b. Thus, $c = g(f(a))$ is uniquely determined by a, and therefore the composition of f and g is also a function, denoted by $g \circ f : A \to C$, such that $(g \circ f)(x) = g(f(x))$ [15].

When discussing functions and their outputs, the term *map* or *mapping* may be used. By this it is meant that when $f(a) = b$, then f maps a to b, which is denoted $a \mapsto b$ by f. The term *mapping* may also be used instead of *substitution*, as the concepts are related. This should be understood from context.

2.1.2 Combinatorics

Combinatorics is a field of mathematics concerned with *counting*, and is based on set theory. One of the building stones of this field is the *multiplication principle*, which states:

> If an activity consists of k stages, and the ith stage can be carried out in α_i different ways, irrespective of how the other stages are carried out, then the whole activity can be carried out in $\alpha_1\alpha_2\ldots\alpha_k$ ways.

More on this principle – and the rest of this subsection – can be found in [1].

Binary numbers are numbers in the base-2 numeral system, using the set $\{0, 1\}$ as a base, where the binary numbers '0', '1', '10', '11', and '100' are, respectively, 0, 1, 2, 3, and 4, in the decimal system. The multiplication principle stated above can be used to find, for instance, the number of different binary numbers of length k (where '1' is represented by '01' when forced to be length 2), as there are two *choices* for each entry of the number (so $\alpha_1 = \alpha_2 = \cdots = \alpha_k = 2$). The number of different binary numbers is, then, by the multiplication principle,

$$2 * 2 * \cdots * 2 = 2^k,$$

i.e. 2 multiplied with itself k times. E.g. for length 2, there are $2^2 = 4$ different binary numbers: '00', '01', '10', '11'. In numbers such as these, repetition of objects is allowed ('00'), and there is an order of the objects in the list (that is, '10' $\neq$ '01').

If, however, we cannot repeat objects – i.e. choosing one means we cannot choose it again at a later point – it means the number of choices decreases for each choice we make. Say there are k choices in total of some arbitrary list of objects to be put in an order. Then we have k choices among the objects for the first entry of the order, and $(k - 1)$ choices for the second entry, and so on, until at the very last entry there is only one choice left: the object that has not been chosen yet. Mathematically, this is named *factorial*, and is denoted by an exclamation point, '!'. It can be defined as

$$n! = n * (n - 1) * (n - 2) \cdots 2 * 1. \tag{2.1}$$

The number of different *permutations* (i.e., a re-ordering of elements) of a list is equal to $n!$.

Removing the restriction of there being an order to the objects, but maintaining the restriction of no repetitions being allowed, we have what we call a *combination*, a *selection*, or more formally: the *binomial coefficient*. It is defined

$$\binom{n}{k} = \frac{n!}{k!(n - k)!} \tag{2.2}$$

and can be read as "from n choose k".

On another note, but within combinatorics, is the sequence of numbers discovered by Fibonacci when he supposedly was investigating the growth of rabbit

populations. It is a sequence that shows up in many areas – not only mathematics – and is defined *recursively*, i.e. in terms of itself:

$$F_n = F_{n-1} + F_{n-2}, \tag{2.3}$$

where $F_0 = 0$ and $F_1 = 1$. The first few numbers of the sequence are

$$0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$$

The summation (using '+', i.e. regular addition) of multiple terms defined iteratively from i to n, where $i, n \in \mathbb{Z} : i \leq n$, is often written with Σ (capital sigma), e.g.:

$$\sum_{i=0}^{n} 2^i = 2^0 + 2^1 + \cdots + 2^{n-1} + 2^n,$$

and multiplication is usually denoted by Π (capital pi), e.g.:

$$\prod_{i=1}^{n} i = 1 * 2 * \cdots * (n-1) * n = n!$$

2.1.3 Abstract Algebra

The following basic concepts of abstract algebra are written in [13], which serves as a good introduction for this subject. Further explanations and examples can be found there.

The *closure* of an operation defined on a set means that the application of the operator on the elements of the set always yields an element which is also in the set. In abstract algebra, a *group* $\langle G, + \rangle$ is a set G which is *closed* under a binary operation +, that satisfy the following axioms:

1. + is associative.
2. there is an *identity element* $0 \in G$ such that, for any $x \in G$:

$$0 + x = x + 0 = x$$

3. For any $a \in G$ there is a corresponding *inverse element* $a^{-1} \in G$, where:

$$a + a^{-1} = a^{-1} + a = 0$$

A group is *abelian* if the binary operation is commutative.

Furthermore, a *ring* $\langle R, +, * \rangle$ consists of a set of elements R and two operations defined on that set, called addition (+) and multiplication (∗), such that the following axioms hold:

1. $\langle R, + \rangle$ is an abelian group (i.e. + is commutative).
2. Multiplication is associative.
3. Both the left- and right distributive laws apply.

If a ring has *unity*, there is a *multiplicative identity element* (denoted by $1 \in R$), where, for some $a, a^{-1} \in R$, if $a*a^{-1} = a^{-1}*a = 1$, then a^{-1} is the multiplicative inverse of a.

Building on this, a *field* is a ring with unity where also multiplication is commutative, and all *non-zero* elements in R has a multiplicative inverse.

In many cases the set of elements is infinite – if it is not, the field is commonly called a *finite field*, and the operations are defined with *modulo n*, where n is the number of elements in the set. Modulo (*mod*, for short) is simply the remainder after division by n, e.g. $16 \equiv 2 \bmod 7$, or $(4+6) \equiv 3 \bmod 7$.

The two-element (finite) field $\mathbb{F}_2$ is the set of integers $\{0, 1\}$ together with multiplication and addition modulo 2. For $a, b \in \mathbb{F}_2$, where $a \neq b$, addition is represented by (the XOR-operation) $\oplus$, where $a \oplus a = 0$ and $a \oplus b = 1$, and multiplication is represented by (the AND-operation) $*$, defined $a * a = a$ and $a * b = 0$ [10, 13]. In some cases, when the operation is obvious from the context, the symbol $*$ may be omitted, such that $a * b := ab$.

2.1.4 Linear Algebra

The following concepts refer to [17], and further examples and information can be found there. Linear algebra is the study of *linear equations* and *systems of linear equations*, and the field introduces basic concepts such as *vectors* and *matrices*, that are highly important for modern Computer Science.

A *vector* is, simply put, an *ordered list of numbers*. They are contained within the Cartesian product of a set of numbers, e.g. $\mathbb{R}^3 = \{(x, y, z) \mid x, y, z \in \mathbb{R}\}$. For any set of numbers A, a vector $\mathbf{v} \in A^n$ has length n, and can be written as an n-tuple $\mathbf{v} = (v_1, v_2, \ldots, v_n)$, where each $v_i \in A$ (for $i = 1, \ldots, n$). A *scalar* is a number $c \in A$. Multiplication by scalars for vectors is defined: for $\mathbf{v} \in A^n : c\mathbf{v} = (cv_1, cv_2, \ldots, cv_n)$.

For two vectors $\mathbf{v} = (v_1, v_2, \ldots, v_n)$ and $\mathbf{w} = (w_1, w_2, \ldots, w_n)$, addition is defined as $\mathbf{v} + \mathbf{w} = (v_1 + w_1, v_2 + w_2, \ldots, v_n + w_n)$, or, representing the vectors as *column vectors:*

$$\begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} + \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix} = \begin{bmatrix} v_1 + w_1 \\ v_2 + w_2 \\ \vdots \\ v_n + w_n \end{bmatrix}$$

The *dot product* (or sometimes *inner product* or *scalar product*) of two vectors is similar to addition of vectors, but with multiplication instead. It is represented by $(\cdot)$ and is defined $\mathbf{v} \cdot \mathbf{w} = v_1 w_1 + v_2 w_2 + \cdots + v_n w_n \in A$, for $\mathbf{v}, \mathbf{w} \in A^n$, that is, the result is a scalar.

A *vector space* $\mathbb{V}$ is a non-empty set of vectors, along with two operations defined on the set, called *addition* and *multiplication by scalars*, on which each of the following axioms hold: There is *closure* of the operations, addition is commutative, both addition and multiplication is associative, and both the left and right distributive laws apply. Also, for $\mathbf{u}, \mathbf{v}, \mathbf{w} \in \mathbb{V}$, there exists a zero vector $\mathbf{0}$ such that $\mathbf{u} + \mathbf{0} = \mathbf{u}$ and for each $\mathbf{u} \in \mathbb{V}$ there exists $-\mathbf{u}$ such that $\mathbf{u} + (-\mathbf{u}) = \mathbf{0}$. Finally, multiplication with the scalar 1 is the identity operation ($1\mathbf{u} = \mathbf{u}$). In short, a vector space is defined over a field, as defined in Section 2.1.3.

$$\begin{bmatrix} a_{1,1} & \cdots & a_{1,j} & \cdots & a_{1,n} \\ \vdots & & \vdots & & \vdots \\ a_{i,1} & \cdots & a_{i,j} & \cdots & a_{i,n} \\ \vdots & & \vdots & & \vdots \\ a_{m,1} & \cdots & a_{m,j} & \cdots & a_{m,n} \end{bmatrix}$$

Figure 2.1: A representation of an $m \times n$ matrix, with indices shown

As shown in Section 2.1.2, summation over multiple terms can be denoted by Σ when using $+$. In $\mathbb{F}_2$, with multiplication and addition modulo 2 ($\oplus$) defined as in Section 2.1.3, we can substitute Σ with $\oplus$ and sum modulo 2 over many elements. For instance, if there is a need to sum all vectors in a set $V = \{(0,1,0),(1,0,0),(1,0,1)\} \subset \mathbb{F}_2^3$, this would be denoted:

$$\bigoplus_{\mathbf{v} \in V} \mathbf{v} = (0,1,0) \oplus (1,0,0) \oplus (1,0,1) = (0,1,1),$$

wherein the iterator $\mathbf{v} \in V$ determines which objects to sum over, but says nothing about the order of which they are summed. Since $\oplus$ (as well as common addition, and multiplication) is commutative, this is of no consequence.

A *matrix* of size $m \times n$ may be represented as an array of entries, and consists of m rows (horizontal) and n columns (vertical), where each row of the matrix is a list of numbers, sometimes called row-vectors. See Figure 2.1 for an overview of a general matrix of size $m \times n$.

Addition of two matrices A and B can be defined as addition of each row-vector of the corresponding matrices, that is,

$$\begin{bmatrix} a_1 & b_1 \\ c_1 & d_1 \end{bmatrix} + \begin{bmatrix} a_2 & b_2 \\ c_2 & d_2 \end{bmatrix} = \begin{bmatrix} a_1 + a_2 & b_1 + b_2 \\ c_1 + c_2 & d_1 + d_2 \end{bmatrix}$$

and multiplying a matrix with a scalar is merely multiplying each entry of the matrix with said scalar.

Multiplying an $m \times n$ matrix A with a vector $\mathbf{b} = (b_1, \ldots, b_n)$, is done by scalar multiplying each of the n scalars in $\mathbf{b}$ with each *column* $\mathbf{a}_1, \ldots, \mathbf{a}_n$ in A:

$$A\mathbf{b} = b_1\mathbf{a}_1 + \cdots + b_n\mathbf{a}_n,$$

which equals a vector of length m.

If B is an $m \times n$ matrix, and A is an $n \times p$ matrix with columns $\mathbf{a}_1, \ldots, \mathbf{a}_p$, the product BA (*matrix multiplication*) is the resulting $m \times p$ matrix whose columns are $B\mathbf{a}_1, \ldots, B\mathbf{a}_p$ [17]. Note that matrix multiplication is *not* commutative, and such $AB \neq BA$, that is, the two are not necessarily equal.

If the number of rows equals the number of columns of a matrix, it is referred to as an $n \times n$ matrix, or a *square* matrix. The *identity matrix*, often denoted as I_n (where the subscript n refers to the number of rows and columns), is a matrix of which the *diagonal entries* $i_{j,j}$ for $j \in \{1, \ldots, n\}$ are all 1, and all

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

Figure 2.2: Identity matrix I_3

other entries are 0 (see Figure 2.2). The main property of the identity matrix is that $I_n\mathbf{x} = \mathbf{x}$ for all vectors of length n, and $I_mA = A = AI_n$, for an $m \times n$ matrix A.

A square ($n \times n$) matrix A is said to be *invertible* if there exists an $n \times n$ matrix C such that

$$CA = I_n \text{ and } AC = I_n.$$

This C is in fact unique and is determined by A, and is therefore called the *inverse* of A, denoted by A^{-1}. A matrix that is not invertible is called a *singular matrix*, and sometimes an invertible matrix may therefore be called *non-singular*.

2.1.5 Boolean Algebra

P	Q	True	False	$\neg P$	$P \wedge Q$	$P \oplus Q$	$P \vee Q$
0	0	1	0	1	0	0	0
0	1	1	0	1	0	1	1
1	0	1	0	0	0	1	1
1	1	1	0	0	1	0	1

Table 2.1: Examples of named Boolean functions with zero, one, or two variables

An important concept in Cryptography, which will be discussed in great detail further in this thesis, are *Boolean functions* – named after George Boole (1815-1864), who laid the foundation for what is now called *Boolean Algebra* [10], and are usable in a range of fields (not only in mathematics, cf. *logic gates* in Electrical Engineering). They are based on the concept of a preposition being *true* or *false*, also denoted 1 or 0, respectively, in Computer Science [15]. Two preposition variables P, Q can be used together with a logic operator to form a logical sentence, which makes a statement in the relevant context. E.g., if we let P = "it is raining", Q = "it is wet", then the sentence "not(P) or Q" states "it is not raining, or it is wet (or both)" – using the logic operators *not* and *or*, first of which takes one variable, second takes two. These logical operators can be represented as Boolean functions, that map values from the domain $\mathbb{F}_2^n$ to co-domain $\mathbb{F}_2$, i.e. $\mathbb{F}_2^n \to \mathbb{F}_2$, where n is the number of variables of the functions. (As mentioned below, this will be further discussed and defined in Section 2.2.)

The Boolean functions in $n = 0, 1, 2$ variables are the building blocks of larger functions, and as such they all have been named: the 0-variable constant functions *True* and *False*, the one-variable negation function *not*, and the two-variable functions *and, exclusive or*, and *inclusive or*, etc. The value mappings

n	Boolean Functions
0	2
1	4
2	16
3	256
4	65 536
5	4 294 967 296
6	18 446 744 073 709 551 616

Table 2.2: The number of different Boolean functions in n variables

of the functions mentioned here are shown respectively in Table 2.1, and each column is a *truth table* of the respective functions, which can be represented by a vector of length 2^n. Any two Boolean functions with the same truth table are said to be equal. Since there are 2^k different vectors of length k, (cf. binary numbers, Section 2.1.2), and the truth tables are of length $k = 2^n$, it should be clear that the number of different Boolean functions in n variables is equal to 2^{2^n}. An overview of how many functions there are in $n \leq 6$ variables, following this formula, is given in Table 2.2.

A more specific definition of Boolean functions in the context of cryptographic Boolean functions and this thesis, is given in Section 2.2 (i.e. Definition 2.2).

2.1.6 Cryptography

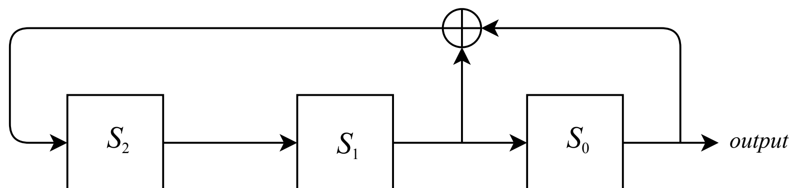


Figure 2.3: Example: a simple LFSR, from [21]

In Computer Science, a *bit* is one digit of information, usually represented by a 0 or 1. The plaintext of any message can be transformed to bits, which can then be encrypted and decrypted, for secrecy. Encrypting each bit in a given plaintext *individually* is called a *stream cipher*, while encrypting an entire block of plaintext bits at a time is called a *block cipher*. In practice, the length of a block can vary in size, but is often at a length of 128 bits (AES) or 64 bits (DES, 3DES) [21].

A *linear feedback shift register* (LFSR) is a type of sequential logic circuit consisting of clocked storage elements and a feedback path, whose input bit is the output of a linear function of (two or more of) its previous states [10, 21]. Provided an initial input string, an LFSR generates a sequence of bits which can be used in a variety of cryptographic systems, although, since there are finite possible states, the sequence must eventually be periodic. However, nonlinearity

of the system can be attained using combinations of LFSRs.

Any LFSR can be represented by a linear Boolean function. The sequence of the example given in Figure 2.3 (listed in [21]) is determined by the function $S_2 = S_1 \oplus S_0$, and starts to repeat after clock cycle 6. LFSRs are often used in stream ciphers.

No matter the implementation and chosen method of security, strong cryptographic systems must obtain the principles of confusion and diffusion, as described by Shannon, where confusion is the principle of obscuring the relationship between key and ciphertext, and diffusion spreads the influence of one plaintext symbol over many ciphertext symbols, the goal being to hide statistical properties of the plaintext [21].

The confusion principle can be achieved with the use of *S-boxes* (Substitution boxes), where an $m \times n$ S-box can be defined as a function $S : \mathbb{F}_2^m \to \mathbb{F}_2^n$, i.e. an n-tuple of Boolean functions on $\mathbb{F}_2^m$ [10]. S-boxes satisfy various cryptographic properties (e.g. highly nonlinear, correlation immunity, etc.), which stems from the Boolean functions within. S-boxes are especially used in block ciphers, for instance in the AES algorithm (also known as Rijndael), or DES [21]. Although they relate mainly to the principle of confusion, choosing Boolean functions that satisfy the *Strict Avalanche Criterion* (SAC) may improve the diffusion of the cryptosystem [6]. A Boolean function $f(\mathbf{x})$ in n variables is called a SAC function if changing any one of the n bits in the input $\mathbf{x}$ results in the output of the function being changed for exactly half of the 2^{n-1} vectors $\mathbf{x}$ with the changed input bit [10].

The term *brute-force* is applied to problem solving wherein all possible configurations of a problem is checked. A *brute-force attack* is an attempt at breaking[1] cryptographic systems by iteration through all possible states, such that the encrypted information is deciphered. In theory, brute-force can be applied to any system, and the only guard against such an attack, in practice, is time. Although power of computation in modern computers is growing at a high rate, it is a rather simple task to ensure the solution space of a cryptographic system is of such a size where it would take years to check every possibility.

As an answer to this, there is a high variety of other attacks that can be deployed on a cryptographic system. These (non-brute-force) attacks take advantage of any weakness that can be found, ranging from mathematical properties (e.g. linearity of the system) to social engineering (e.g. abusing the trust of end-users of the system). For this thesis, the former type of attacks are relevant, revolving around analysis of properties of Boolean functions that can help in avoiding algebraic attacks, correlation attacks, higher-order differential attacks, with more [4].

These attacks attempt to exploit or break parts of the cryptographic systems by various means and methods. To avoid such attacks, the designers of the algorithms must choose good cryptographic Boolean functions which show high complexity, determined by, for instance, the nonlinearity and algebraic degree (defined in Section 2.2) – among others – of the functions. Another feature of Boolean functions is also the *number of terms* in each function, but, as Carlet points to in [4], the complexity of functions with a high number of terms can be equal to the complexity of some function with a lower number of terms.

[1] In the sense of searching for and finding a solution.

x_1	x_2	1	0	$x_1 \oplus 1$	x_1x_2	$x_1 \oplus x_2$
0	0	1	0	1	0	0
0	1	1	0	1	0	1
1	0	1	0	0	0	1
1	1	1	0	0	1	0

Table 2.3: Basic operators for Boolean functions

2.2 Boolean Functions

In this thesis there are a number of concepts used that build on the concepts defined in the previous section. Many of these definitions are explained in more detail in [10], a recommendation for the interested reader new to this subject. As the subject of this thesis revolves around the complexity of Boolean functions as cryptographic tools, the following section serves to define such functions – and a number of properties of them – in precise and rigorous terms. Therefore, the following definition is given for Boolean functions (in more detail than Section 2.1.5). Recall that $\mathbb{F}_2$ is the finite field defined over the set $\{0, 1\}$, with multiplication and addition modulo 2.

Definition 2.2. (Boolean Function) [4, 10]
A Boolean function f in n variables, where n is any positive integer, is a function from the vector space $\mathbb{F}_2^n$ to the finite field $\mathbb{F}_2$, i.e. $f : \mathbb{F}_2^n \rightarrow \mathbb{F}_2$. The set of all Boolean functions in n variables is denoted by $\mathcal{B}_n$, and the symbol $\oplus$ denotes addition modulo 2, in $\mathbb{F}_2$, $\mathbb{F}_2^n$, and $\mathcal{B}_n$.

The variables of Boolean functions in this thesis will be denoted by a vector $\mathbf{x} = (x_1, \ldots, x_n)$, where n is the number of variables. The operators used on these variables are $\oplus$ for addition and $(*)$ for multiplication (when explicitly needed, in most cases the variables are concatenated, i.e. put together), where the order of operations is as usual: first multiplication, then addition – e.g., for $ab \oplus c$, ab is first multiplied, then added to c. Value mappings of the basic operators used are given in Table 2.3 (cf. Table 2.1, excluding the right-most column, and substituting P with x_1 and Q with x_2).

Any Boolean function in $\mathcal{B}_n$ can be expressed as a polynomial in

$$\mathbb{F}_2[x_1, \ldots, x_n]/(x_1^2 - x_1, \ldots, x_n^2 - x_n)$$

(see [10, 7]), consisting of one or more terms (except the constant 0 function, defined to contain zero terms) summed over $\oplus$. These terms are the product of one or more variables in $\{x_1, \ldots, x_n\}$, that is

$$\prod_{i=1}^{n} x_i^{u_i} = x_1^{u_1} * \cdots * x_n^{u_n}$$

where u is a vector in $\mathbb{F}_n^2$, and where the zero-vector yields the constant term, i.e. $x_1^0 \cdots x_n^0 = 1$. E.g. let $n = 3$ such that the variables are $\{x_1, x_2, x_3\}$ and let $u = (0, 1, 1)$. Then

$$\prod_{i=1}^{n} x_i^{u_i} = x_1^0 * x_2^1 * x_3^1 = 1 * x_2 * x_3 = x_2x_3.$$

These terms are also referred to as *monomials* (*mono* means *one* and *poly* means *many* in Greek, thus a *poly*nomial consists of *many* terms, where the monomial is only *one* term), and the polynomial representation of a Boolean function is described in the following definition:

Definition 2.3. (Algebraic Normal Form) [4, 10]
Every Boolean function f has a unique representation called its *algebraic normal form* (ANF) as a polynomial over $\mathbb{F}_2$ in n variables, whose degree relative to each variable x_i is at most 1:

$$f(\mathbf{x}) = \bigoplus_{\mathbf{u}\in\mathbb{F}_2^n} c_{\mathbf{u}} \left(\prod_{i=1}^{n} x_i^{u_i}\right) = \bigoplus_{\mathbf{u}\in\mathbb{F}_2^n} c_{\mathbf{u}} \mathbf{x}^{\mathbf{u}},$$

where each $c_{\mathbf{u}} \in \mathbb{F}_2$, $\mathbf{u} = (u_1, \dots, u_n)$ and $\mathbf{x} = (x_1, \dots, x_n)$.

The existence and uniqueness of the ANF of every Boolean function is further discussed in [6], using an equivalent definition. Each Boolean function discussed in this thesis will be presented in its corresponding ANF. The number of variables in the highest-order monomial with nonzero coefficient (i.e., that is present in the ANF) is called the *algebraic degree* of a function. If all monomials of a function each have the same individual number of variables, the function is called *homogeneous* [10].

An *affine function* $\ell_{\mathbf{u},c}$ is a function with algebraic degree at most 1, which takes the form

$$\ell_{\mathbf{u},c}(\mathbf{x}) = \mathbf{u} \cdot \mathbf{x} \oplus c = u_1 x_1 \oplus \cdots \oplus u_n x_n \oplus c, \tag{2.4}$$

where $\mathbf{u} = (u_1, \dots, u_n) \in \mathbb{F}_2^n$ and $c \in \mathbb{F}_2$. If $c = 0$, such that $\ell_{\mathbf{u},0}$ only consists of monomials of algebraic degree 1, and no constant, then it is a *linear* function [10].

Definition 2.4. (Affine Transformation) [17]
An *affine transformation* $T : \mathbb{F}_2^n \to \mathbb{F}_2^n$ is a transformation of the form $T(\mathbf{x}) = A\mathbf{x} + \mathbf{b}$, with A an $n \times n$ matrix, and $\mathbf{b}$ in $\mathbb{F}_2^n$.

By matrix multiplication, the affine transformations map each x_i in $\mathbf{x} = (x_1, \dots, x_n)$ to an affine function given by $x_j = \sum a_{i,j} x_i + b_j$, for each $i, j \leq n$, where $a_{i,j}$ is the entry of A in column i, row j.

The matrices considered are invertible, such that the affine transformations are invertible, meaning no information is lost in the transformation. (E.g., mapping all variables x_j to 0 is a non-invertible affine transformation, since all information is lost). In the remainder of the text, when discussing affine transformations, only those that are invertible are included. This thesis does not concern itself with non-invertible affine transformations, and when referring to *all affine transformations*, it is implied that these are invertible.

In practice, an affine transformation on a Boolean function f transforms f into another Boolean function g, via a map $T(\mathbf{x}) = A\mathbf{x}+\mathbf{b} = (x_1 \mapsto t_1, \dots, x_n \mapsto t_n)$, such that $f(t_1, \dots, t_n) = g(x_1, \dots, x_n)$. This transformation $A\mathbf{x} + \mathbf{b} = (t_1, \dots, t_n) = \mathbf{t}$ is a mapping of each individual variable x_i in f to an affine function

$$t_i = a_{i,1} x_1 \oplus \cdots \oplus a_{i,n} x_n \oplus b_i,$$

where $a_{i,j}$ is the entry at row i, column j of A, and the same i indicates the index in $\mathbf{b}$, and $\mathbf{t}$ itself. This g, that f can be mapped to through such an affine transformation, is said to be equivalent to f through affine transformations. An example of affine transformations is given towards the end of this section (see Example 2.3).

Iteration through all affine transformations on a specific Boolean function, and storing all functions found, yields what we call the *orbit* of said Boolean function. This orbit is disjoint from any other orbit in $\mathcal{B}_n$, meaning no orbits share any functions. This topic will be further discussed in Chapter 4, where it is related to the work [16] of Harrison on equivalence classes.

Remark. A *variable permutation* of a Boolean function f is an affine transformation $A\mathbf{x} + \mathbf{b}$ where each row in the matrix A only contains one entry of 1, with 0 in the remaining entries (i.e., A is a permutation of I_n, the identity matrix), and $\mathbf{b} = \mathbf{0}$, i.e. the zero-vector. This means we are only exchanging the positions of the variables of f. E.g., for a function $f(x_1, x_2, x_3) = x_1x_3 + x_2$, we permute the variables as such:

$$
\begin{aligned}
x_1 &\mapsto x_2 \\
x_2 &\mapsto x_1 \\
x_3 &\mapsto x_3
\end{aligned}
$$

and the resulting function is $g(x_1, x_2, x_3) = x_2x_3 + x_1 = f(x_2, x_1, x_3)$, which is equivalent to f over affine transformations. The number of variable permutation transformations is $n!$ (see Section 2.1.2, specifically (2.1)).

The general affine group of transformations is defined over the field $\mathbb{F}_2$, and is applied as a transformation group to Boolean functions [16]. The elements of this group are the affine transformations as explained above. The complete number of all (invertible) affine transformations in n variables is the number of $n \times n$ invertible matrices multiplied with the number of vectors in $\mathbb{F}_2^n$. As proven in [12], the number of $n \times n$ invertible matrices over $\mathbb{F}_2$ is:

$$
\prod_{i=0}^{n-1} (2^n - 2^i), \tag{2.5}
$$

which is mentioned also in [16] (for more information on the general affine group, refer to this article). The number of vectors in $\mathbb{F}_2^n$ is known to be 2^n (cf. binary numbers, described in Section 2.1.2). A complete overview of the number of invertible matrices and vectors in $n \leq 5$ variables is given in Table 2.4, together with the number of possible affine transformations. Note that the application of two different affine transformations on a function f may result in the same function, e.g. $f(x_1, x_2) = x_1$, and $f(x_1, x_1 + x_2) = x_1 = f(x_1, x_2 + 1)$.

n	Invertible matrices	Vectors	Affine Transformations
2	6	4	24
3	168	8	1344
4	20 160	16	322 560
5	9 999 360	32	319 979 520

Table 2.4: Overview of number of elements in the listed sets, in n variables

Definition 2.5. (Hamming Weight and Distance) [10]

1. The *Hamming weight* of a vector $\mathbf{x} \in \mathbb{F}_2^n$ is denoted by $wt(\mathbf{x})$ and is equal to the number of 1's in the vector $\mathbf{x}$.

2. For a Boolean function f on $\mathbb{F}_2^n$, let $\Omega_f = \{\mathbf{x} \in \mathbb{F}_2^n \mid f(\mathbf{x}) = 1\}$ be the *support* of f. The Hamming weight of f is then $|\Omega_f|$, or equivalently, the weight of the vector of its truth table.

3. The *Hamming distance* between two functions $f, g : \mathbb{F}_2^n \to \mathbb{F}_2$, denoted by $d(f, g)$, is defined as

$$d(f, g) = wt(f \oplus g)$$

Noted in Section 2.1.5, there are 2^n entries (or, rows in its truth table) in a Boolean function f in n variables. If $wt(f) = 2^{n-1}$, this means there are 2^{n-1} vectors in $\mathbb{F}_2^n$ mapped by f to 1, and therefore 2^{n-1} mapped to 0. In this case (where these two values are equal), we say that f is *balanced*, which is an important property for cryptographic Boolean functions [10, 5].

Another property of Boolean functions is the number of monomials in the ANF of the function. The minimum number of monomials a function can have is 0, and this is only true for the constant 0 function, $f(\mathbf{x}) = 0$. Any other function in $\mathcal{B}_n$ has a number of monomials in the range of $\{1, \ldots, 2^n\}$ in their ANF. There is only one function in n variables with 2^n monomials, and this function is equal to $\prod_{i=1}^{n}(x_i + 1)$.

The term *monomial count* will, in this thesis, be a shorthand for *the number of monomials* of a function in its ANF. That is, from Definition 2.3, for each f, the monomial count is the sum of each $c_{\mathbf{u}} \in \mathbb{F}_2$ that determines the ANF of f. In short, it is the number of nonzero terms in the ANF of the function. Since the ANF of a function f is unique, the monomial count of f is well-defined. More on the monomial counts of Boolean functions is discussed in Section 4.3, including a distribution of how many functions have a certain number of monomials in their corresponding ANF (see Table 4.3).

Each Boolean function $f : \mathbb{F}_2^n \to \mathbb{F}_2$ is associated with a *sign function*, denoted by $\hat{f}(\mathbf{x}) : \mathbb{F}_2^n \to \mathbb{R}^*$, i.e. the sign function of f, defined by

$$\hat{f}(\mathbf{x}) = (-1)^{f(\mathbf{x})} \tag{2.6}$$

such that, for a specific $\mathbf{x}$,

$$\text{if } f(\mathbf{x}) = 0 \text{ then } \hat{f}(\mathbf{x}) = (-1)^0 = 1, \text{ and}$$
$$\text{if } f(\mathbf{x}) = 1 \text{ then } \hat{f}(\mathbf{x}) = (-1)^1 = -1.$$

This function is related to the *Walsh transform*, given in the next definition [10].

Definition 2.6. (Walsh Transform) [10]
The *Walsh transform* of a function f on $\mathbb{F}_2^n$ is the map $W(f) : \mathbb{F}_2^n \to \mathbb{R}$, defined by

$$W(f)(\mathbf{w}) = \sum_{\mathbf{x} \in \mathbb{F}_2^n} f(\mathbf{x})(-1)^{\mathbf{w} \cdot \mathbf{x}}$$

where $\mathbf{w} \in \mathbb{F}_2^n$, and $\mathbf{w} \cdot \mathbf{x}$ is the dot product, as defined in Section 2.1.4.

Applying the Walsh transform to $\hat{f}$ is often referred to as Walsh-Hadamard transform, or sometimes just Hadamard transform. By application of Definition 2.6 and (2.6):

$$W(\hat{f})(\mathbf{w}) = \sum_{\mathbf{x} \in \mathbb{F}_2^n} \hat{f}(\mathbf{x})(-1)^{\mathbf{w} \cdot \mathbf{x}} = \sum_{\mathbf{x} \in \mathbb{F}_2^n} (-1)^{f(\mathbf{x}) \oplus \mathbf{w} \cdot \mathbf{x}}$$

The list of numbers attained by iteration over all $\mathbf{w} \in \mathbb{F}_2^n$ and collecting $W(\hat{f})(\mathbf{w})$ is called the *Walsh-Hadamard spectrum.* If the absolute value of $W(\hat{f})(\mathbf{w})$ for all $\mathbf{w}$ is either equal to 0 or $2^{\frac{n+s}{2}}$, for a fixed integer s, f is called *s-plateaued* [23].

If an affine function (see (2.4)) is used in the encryption of data, there are known attacks that can take advantage of the linearity and break the encryption faster than brute-force [4]. If a function is not affine, it is called *nonlinear*, and the lowest Hamming distance from a Boolean function f to any affine function ϕ is called the *nonlinearity* of f.

Definition 2.7. (Nonlinearity) [10]
The *nonlinearity* of a function f, denoted by $\mathcal{N}_f$, is defined as

$$\mathcal{N}_f = \min_{\phi \in \mathcal{A}_n} d(f, \phi)$$

where $\mathcal{A}_n$ is the class of all affine functions on $\mathbb{F}_2^n$, and min is the minimum-function, outputting the minimum element in a set of comparable objects.

Note that when speaking of nonlinearity in general, and not for a specific f, only the symbol $\mathcal{N}$ will be used.

An equivalent definition of nonlinearity, based on the interpretation that $W(\hat{f})(\mathbf{w})$ is equal to the number of 0's minus the number of 1's in the binary vector $f \oplus \ell_{\mathbf{w}}$, is given in [10], and states that

$$\mathcal{N}_f = 2^{n-1} - \frac{1}{2} \max_{\mathbf{w} \in \mathbb{F}_2^n} |W(\hat{f})(\mathbf{w})| \tag{2.7}$$

This definition is used in some programming language libraries, as it is clearly more efficient than checking the distance to all affine functions – for instance, this is what is used in SageMath ([25], see source code for *boolean_function*), which will be expanded upon in Chapter 3. A full distribution of the nonlinearity of functions in $n \leq 5$ variables is given in Chapter 5, see Table 5.2.

The nonlinearity $\mathcal{N}_f$ of any Boolean function f in n variables satisfies

$$\mathcal{N}_f \leq 2^{n-1} - 2^{\frac{n}{2}-1} \tag{2.8}$$

[4, 10]. The functions that achieve equality to the right-hand side of (2.8) are functions with the highest nonlinearity possible, and denote a special class of functions, called bent functions. These functions have many interesting properties – not only for cryptographic purposes – on which much literature has been written.

Definition 2.8. (Bent Boolean functions) [10]
A Boolean function f in n variables is called *bent* if and only if the Walsh transform coefficients of $\hat{f}$ are all $\pm 2^{\frac{n}{2}}$, that is, $W(\hat{f})^2$ is constant.

From this definition, it is easy to see that bent functions *only* exist for even dimensions, i.e. $n = 2k$, for some integer k, since $2^{\frac{n}{2}}$ is not an integer for odd n; and that all bent functions are plateaued (with $s = 0$).

For odd n, where $n = 2k + 1$, the question regarding how close one can get to this value is not completely answered to this day. A class of functions related to this, that always achieve *high* nonlinearity in odd n, is the class of *semi-bent* functions [10, 19, 9]. The following definition is used for this thesis, as it is the most general case of this type of function. Below the definition is a remark about a more restricted class, which is defined in [10].

Definition 2.9. (Semi-Bent Boolean functions) [23]
In odd n, if a Boolean function $f \in \mathcal{B}_n$ is s-plateaued with $s = 1$, such that $|W(\hat{f})(\mathbf{w})| \in \{0, 2^{\frac{n+1}{2}}\}$ for all $\mathbf{w}$ in $\mathbb{F}_2^n$, then f is a *semi-bent* function.

Letting $s = 2$, this class of functions can be found in even n as well, but the definition as it stands suits the purposes of this thesis, and the focus is rather put on bent functions for even n.

Remark. The mentioned stricter class of semi-bent functions given in [10] is concerned with semi-bent functions of a special form: for an integer k, such a function in $2k+1$ variables is the concatenation of f_0 and f_1, where f_0 is a bent function in $n = 2k$ variables and $f_1 = f_0(A\mathbf{x} \oplus \mathbf{b}) \oplus 1$ (for a nonsingular $n \times n$ matrix A over $\mathbb{F}_2$ and a vector $\mathbf{b} \in \mathbb{F}_2^n$). In this form, the semi-bent functions are balanced and have $\mathcal{N} = 2^{2k} - 2^k$. In this thesis, this restricted form will not be explored further, but is mentioned here because these two differing definitions have been the cause of some confusion. Note that by using Definition 2.9, not all semi-bent functions are balanced, as will be shown in Chapter 5.

The maximum algebraic degree for a bent function is $\frac{n}{2}$, and for a semi-bent function (as in Definition 2.9) it is $\frac{n+1}{2}$ [14].

Carlet defines in [4] a property related to the monomial count of Boolean functions, which he claims plays an important role – on level with nonlinearity and algebraic degree – in resisting cryptographic attacks against both stream ciphers and block ciphers. He introduces this property by stating that the number of monomials of a function's ANF is itself not satisfactory, as functions with high monomial count sometimes behave similarly with some functions with low monomial count. This definition is one of the main definitions for this thesis, and is what the following programs are designed to calculate.

Definition 2.10. (Algebraic Thickness) [4]
The *algebraic thickness* $\mathcal{T}(f)$ of a Boolean function f is the minimum number of monomials with non-zero coefficients in the ANF of the functions $f \circ \mathcal{A}$, where $\mathcal{A}$ ranges over the general affine group, i.e. all affine transformations.

In other words, for a Boolean function f, the algebraic thickness $\mathcal{T}(f)$ is equal to the minimum monomial count of the functions that f can be mapped to through all possible affine transformations. As this thesis will discuss functions in a different number of variables – and in the interest of being unambiguous when confusion may occur – the sometimes added subscript $\mathcal{T}_n$ will refer to (algebraic) thickness in n variables.

Example 2.3. In $n = 3$ variables, there are 168 3×3 invertible matrices, and $2^3 = 8$ vectors; therefore $168 * 8 = 1344$ affine transformations. Let

$$f(x_1, x_2, x_3) = x_1x_2x_3 + x_1x_2 + x_1 + x_2x_3 + x_3 + 1 \in \mathcal{B}_3,$$

a function with 6 monomials. We want to calculate $\mathcal{T}(f)$.

To do this, we iterate through all affine transformations of f, mapping the function by the affine transformations represented by the matrices and vectors. For each resulting function after transformation of f, we check the number of monomials in its ANF. Through this search, one of the transformations found is:

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

which maps the variables of f as such:

$$\begin{aligned} x_1 &\mapsto x_1 + x_3, \\ x_2 &\mapsto x_2, \\ x_3 &\mapsto x_2 + x_3. \end{aligned}$$

Thus, we can easily calculate what f is mapped to through this transformation (with algebra in $\mathcal{B}_3$):

$$\begin{aligned} f(x_1 + x_3, x_2, x_2 + x_3) &= (x_1 + x_3)x_2(x_2 + x_3) + (x_1 + x_3)x_2 + x_1 + \\ &\quad x_3 + x_2(x_2 + x_3) + x_2 + x_3 + 1 \\ &= x_1x_2x_3 + x_1x_2 + x_1x_2 + x_2x_3 + x_2x_3 + x_2x_3 + \\ &\quad x_2x_3 + x_1 + x_2 + x_2 + x_3 + x_3 + 1 \\ &= x_1x_2x_3 + x_1 + 1, \end{aligned}$$

a function which ANF has three monomials. In our iteration through affine transformations, we find no other function in the orbit of f with fewer monomials, thus $\mathcal{T}(f) = 3$.

If a property of a Boolean function is preserved through all affine transformations, then this property is an *affine invariant*, or invariant under affine transformation [6]. In relation to algebraic thickness, the number of monomials in the ANF of a Boolean function is *not* an affine invariant, arguing that a high monomial count is not a satisfactory complexity criteria for use in cryptography, and why it is interesting to find the minimum number in the orbit of a function. Properties of Boolean functions of relevance to this thesis that *are* invariant under affine transformation are the well-known:

1. Algebraic degree
2. Nonlinearity
3. Balancedness
4. Algebraic thickness

Surely, (1) can be seen to be true by observing that the algebraic degree of any nonzero affine function is – by definition – equal to 1, and the same is true for any one variable. Thus, mapping the variables of a function to (nonzero) affine functions does not change the degree of any monomial in the ANF of the function, and therefore the function itself preserves its degree. This is mentioned briefly in [4, 7].

For (2), it has been shown that the Walsh-Hadamard spectrum is invariant under affine transformation, and therefore all properties related to it, e.g. nonlinearity, bentness and semi-bentness. See [26, 19] for more on this. This means that if a function is bent (or semi-bent), all affine transformations of this functions will also be bent (or semi-bent) functions. In Chapter 5, the number of orbits that are bent for $n = 4$ (and semi-bent for $n = 5$) are listed, and are referred to as *bent orbits* (respectively, *semi-bent orbits*).

Next, (3) can be shown to be invariant under affine transformation by the observation that the transformation is merely a permutation of the vectors in $\mathbb{F}_2^n$ – this results in the truth table of the function to be permuted, but not otherwise altered, therefore the Hamming weight of the truth table is preserved. As with bent functions, the number of *balanced orbits* for $n = 4, 5$ is given in Chapter 5.

Finally, (4) is invariant by its definition, as algebraic thickness relates directly to the orbit of a function through affine transformations.

2.3 Related Work

2.3.1 Algebraic Thickness

The work of Claude Carlet in [8] and [4] is the genesis of algebraic thickness, and these articles are therefore crucial to this project.

The former article defines algebraic thickness, and discusses lower and upper bounds on its maximum value (see Section 2.3.2). It also includes further discussion on the relation that algebraic thickness has with other complexity criteria (e.g. nonlinearity, algebraic degree, etc.). The latter article improves some of the work done in the former, and further expands on the properties of algebraic thickness.

The algebraic thickness of affine functions (i.e. functions with algebraic degree 1 or lower) is at most 1 [4, 3]. The quadratic functions (i.e. functions

with algebraic degree 2) are also well understood, due to a theorem by Dickson, summarized by Boyar and Find in [3]:

Theorem 2.1. ***(Dickson)*** *Let $f : \mathbb{F}_2^n \to \mathbb{F}_2$ be quadratic. Then there exist an invertible $n \times n$ matrix A, a vector $\mathbf{b} \in \mathbb{F}_2^n$, $t \leq \frac{n}{2}$, and $c \in \mathbb{F}_2$ such that for $\mathbf{y} = A\mathbf{x} + \mathbf{b}$ one of the following two equations holds:*

$$f(x) = y_1y_2 + y_3y_4 + \cdots + y_{t-1}y_t + c, \text{ or}$$
$$f(x) = y_1y_2 + y_3y_4 + \cdots + y_{t-1}y_t + y_{t+1}.$$

Furthermore A, $\mathbf{b}$, and c can be found efficiently.

Also of relevance to this thesis is the work [26] of Sertkaya and Doğanaksoy, wherein they write about affine equivalence and preservation of nonlinearity of Boolean functions through bijective mappings. Knowing this proved useful for the analysis of nonlinearity in this thesis, as described at the end of Section 2.2. Also, Table 5.2 (calculated independently) shows the distribution of nonlinearity of all functions in $n = 2, 3, 4, 5$ variables, which is confirmed by their work in [26], as they list the very same table.

2.3.2 Known Bounds on Algebraic Thickness

For Boolean functions in n variables, it is of interest to determine the maximum value possible for $\mathcal{T}_n$, for any $f \in \mathcal{B}_n$, i.e. $max_{f \in \mathcal{B}_n}(\mathcal{T}(f))$ – and specifically, the growth of this value. As the maximum number of terms in the ANF of any f in n variables is smaller or equal to 2^n, $max_{f \in \mathcal{B}_n}(\mathcal{T}(f)) \leq 2^n$ serves as the trivial upper bound. Determining whether the maximum $\mathcal{T}$ is polynomial or exponential in n can have indications for weaknesses of ciphers using Boolean functions [8]. Studies on both lower and upper bounds on this value have been conducted, and are summarized here.

For the lower bound of $max_{f \in \mathcal{B}_n}(\mathcal{T}(f))$, Carlet showed in [8] that, for every number $\lambda < \frac{1}{2}$ and positive n, the density in $\mathcal{B}_n$ of the subset

$$\{f \in \mathcal{B}_n \mid \mathcal{T}(f) \geq \lambda 2^n\}$$

is greater than $1 - 2^{2^n H_2(\lambda) - 2^n + n^2 + n}$, where $H_2(x) = -x\log_2(x) - (1-x)\log_2(1-x)$ is the *entropy function*. In the words of Carlet, they deduced that *almost* all Boolean functions have algebraic thickness greater than $\lambda 2^n$. (See [8] for more on this.) Furthermore, in [4], they improved upon this result, and showed that *almost* all Boolean functions have algebraic thickness greater than

$$2^{n-1} - n2^{\frac{n-1}{2}},$$

and a theorem describing this – and proof thereof – can be found in said paper.

As of now, the best upper bound on algebraic thickness is proven by induction by Carlet in [8], stating that, for every Boolean function f in $\mathcal{B}_n$,

$$\mathcal{T}(f) \leq \frac{2}{3}2^n,$$

a bound which Carlet says could be improved on, listing it as an open problem.

2.4 Methodology

This project is a quantitative simulation case study, and concerns analysis and calculations of mathematical concepts. The validity of a mathematical simulation is grounded in the validity of the mathematical concepts used (given in Sections 2.1 and 2.2), as the simulation explores mathematics quantitatively. More on simulation studies can be found in [2].

In Chapters 3 and 4, the programs used in the calculation of the algebraic thickness distribution for $n \leq 5$ are listed, and a discussion of the efficiency and execution time of the methods and programs is given when relevant. The validity of the results of these programs is discussed in Section 5.4.

Continuous testing of the programs – both in results and in logic flow – was key to developing programs that were efficient and correct in their execution. The tests written were based on the mathematical concepts defined in Sections 2.1 and 2.2, using the built-in packages of the tools used (cf. SageMath [25], further discussed in Section 3.1).

The SOLID-principles – coined by Robert C. Martin and described in [18] – were applied throughout the development process, when applicable. For instance, *the Single-Responsibility Principle* is reflected here in that each method is focused on *one* task (e.g. matrix and vector generation, function generation, etc.). This further supports *the Open-Closed Principle* (i.e., "open for expansion, closed for modification"), in that each method is programmed once, but re-used for multiple testing- and analysis-programs, without modification. Additionally, each method is written for any n, leaving room for further expansion from the lower number of variables to the higher, and re-using the source code. The methods are therefore explained in detail individually, as they were used in combination for several different testing- and analysis-programs, both before and after the distribution was completed.

The compilation of the methods when the intention of the constructed program was to compute the final algebraic thickness distribution, is given in Appendices B and C – this is mentioned in more detail in the chapters, stated above. The various programs used to conduct the analysis (which is listed and discussed in Chapter 5) of the results, are not given in full. Those programs are merely different combinations of the listed methods, with some smaller (and rather trivial) extensions, that would not be of great interest to the point being made (i.e., how the distribution was calculated).

Chapter 3

Calculating thickness distribution for $n = 4$

3.1 SageMath and computational strategy

SageMath [25] is an open-source mathematics software system, which builds on several existing open-source packages, and runs on Python. It includes software for calculations in calculus, linear algebra, abstract algebra, and much more. Source code written in SageMath is parsed into Python code before being compiled and run as a Python program. Updates are rolled out fairly often, as during the writing of this thesis, there have been updates about every 2-3 months, starting with 8.6 in January 2019, 8.7 March 2019, 8.8 June 2019, 8.9 September 2019, and 9.0 January 2020. The versions before 9.0 runs on Python 2, while 9.0 runs on Python 3. The programs and methods discussed in this thesis should run on all versions before 9.0, but were written and used with 8.8 and 8.9.

SageMath was chosen for this project because of the existing interfaces for interaction with Boolean rings and Boolean functions, and the built-in matrix manipulation. SageMath contains a package for Boolean functions – *sage.crypto.boolean_function* – but early in the process it was found that this package did not suit the requirements this project needed, as a memory leak occurred anytime more than approximately 900 Boolean functions were turned into ANF. By checking the source code of this package, it became apparent that the error stemmed from the method of converting Boolean functions (from *boolean_function*) into Boolean polynomials (in the *pbori*-package), which is used for ANF. Because of this, the computational implementation of Boolean functions in this thesis are constructed directly in ANF as Boolean polynomials using the *pbori*-package, and the *boolean_function*-package is used only for its nonlinearity calculation method. Chapter 5 – where the analysis of the results from the programs to be explained is given – contains data and results that have been collected and analysed by use of the methods from these packages.

To our knowledge, there has not been any previous work conducted on calculation of the distribution of algebraic thickness, in a similar manner as this thesis will describe. As such, when this project was first undertaken, a foundation of data was needed for an initial analysis of patterns. In this first data collec-

tion, the brute-force method of iterating through every Boolean function and calculating their respective thickness would be applied, in the hope that there would be implications for use with the higher n (specifically, in $n = 5$). This brute-force program would calculate the distribution of thickness in $n = 2, 3, 4$, and is described and discussed in Section 3.2. From the patterns found by analysis of the collected data, the definitions given in Sections 4.1 and 4.2 were defined, and these were used for calculating the thickness distribution in $n = 5$ variables. The program implementing these are given in Chapter 4. For the interested reader, the full programs implementing the thickness calculation are attached to the thesis as appendices (see Appendices B (brute-force) and C (using the to-be-discussed strategy)).

The algebraic thickness calculation programs for $n = 4, 5$ are both created to be run in iterations. This is because of the sizes of both functions sets, even after optimization for $n = 5$ (which will be discussed later). Several of these program iterations were run simultaneously on each computer available, where one iteration processed one part of the complete function set, with no functions in common with the other sets. The limiting factors for this approach were the amount of processors (CPUs) and the amount of memory (RAM) in each available computer. Attempts were made to utilize *parallel processing* in Python, but as of SageMath 8.9 this is implemented poorly – at least for the purposes needed in this project. Testing was done for $n = 2, 3$ with a method that managed to implement parallel processing, but the time it took far exceeded the time it took to run an equal non-parallel method on the same data set. Upgrading the following programs with a working parallel processing method would be an improvement, but it could not be done at this time, due to the implementation of parallel processing in SageMath 8.8 and 8.9. *(Note: In January 2020, SageMath 9.0 was released, which uses Python 3 instead of Python 2. By this time, the final data collection of this thesis was ongoing, and therefore no changes were made nor attempts to implement this. In the release log (found in [25]), there are notes on changes to the Parallel processing module, which may be of interest in the continuation of the work done in this thesis.)*

Remark. All the methods given have been cleaned up in some parts, i.e. removal of various "quality-of-life"[1] method calls that print to the terminal. E.g., printing which function is being processed, and the time it took to process it; printing the number of matrices and vectors generated (for program validity assurance), the time elapsed, etc. The removal of these method calls is done for the readability of the programs, such that only the logic and flow of the programs can remain.

[1]This term is used here in its colloquial manner, and refers to aspects of the design that are not necessary for the calculations, but indeed necessary for estimating the completion of the programs as they are run.

3.2 Program: Brute-force implementation for $n = 2,3,4$

```
1 n = int(raw_input("n = "))                  # variable integer
2 ring = BooleanPolynomialRing(n, "x")        # F_2
3 x_vars = ring.gens()                        # List of x1, x2, ...
4 m_space = MatrixSpace(GF(2), n)             # Matrix Space
5 x_vec = vector(ring, x_vars)                # Variable vector
```

Figure 3.1: Code: Variable declaration of programs

In the programs listed in the following, there is a standard setup in the main method, consisting of choosing an integer n for number of variables, setting up the Boolean polynomial ring, matrix spaces, etc. Figure 3.1 shows this setup, using the available built-in concepts of SageMath. In the full programs, after this setup, there are method calls using these variables, i.e. calls for the methods described below.

3.2.1 Constructing all Boolean functions

Generation of the algebraic normal form (ANF, see Definition 2.3) of all possible Boolean functions in n variables $\mathcal{B}_n$, can be efficiently generated by construction from the set containing all possible Boolean functions in $(n-1)$ variables $\mathcal{B}_{n-1}$, and the new variable for $\mathcal{B}_n$: x_n; together with multiplication $(*)$ and addition $(\oplus)$ in $\mathbb{F}_2$. Then,

$$\mathcal{B}_n = \{g * x_n \oplus h \mid g, h \in \mathcal{B}_{n-1}\}. \tag{3.1}$$

where $\mathcal{B}_0 = \{0, 1\}$. This holds, because every variable in an arbitrary function $f \in \mathcal{B}_n$ appears with exponent 1, and so, one can gather all monomials in the ANF of f that involve x_n, and factor it out, creating a function g that can only depend on $x_1, \ldots, x_{n-1}$. The monomials that does not involve x_n obviously does not depend on x_n either, and create another function h. Thus, $g, h \in \mathcal{B}_{n-1}$, and we obtain $f(x_1, \ldots, x_n) = g(x_1, \ldots, x_{n-1}) * x_n \oplus h(x_1, \ldots, x_{n-1})$.

In SageMath, by using *BooleanPolynomialRing* and the corresponding variables in it, this can be accomplished as shown in Figure 3.2 – where the input for the method on display is the list of variables declared in Figure 3.1, line 3.

Remark. An alternative function generation was first used, where binary numbers of length n were used to first combine all variables into all possible monomials (including 0, 1), and then a new set of binary numbers of length 2^n were used for summing all combinations of monomials into all possible Boolean functions. Testing revealed that this method of generation was far more inefficient than the algorithm in (3.1) – by a whole *two days* in $n = 5$ variables. Because of this inefficiency, it will not be listed here.

As mentioned in the start of this chapter, both the program for $n \leq 4$ and the program for $n = 5$ are designed to work with several iterations, to save time. For the brute-force program, this was solved by – in each iteration – first

```
def generate_functions(variables):
    generated_funcs = [0, 1]  # n = 0

    # iterate through variables {x_1,.., x_n}
    for new_x in variables:
        temp_funcs = []
        for f1 in generated_funcs:
            for f2 in generated_funcs:
                temp_funcs.append( f1*new_x + f2 )
        generated_funcs = temp_funcs

    return generated_funcs
```

Figure 3.2: Code: Generation of Boolean functions

generating all Boolean functions, then dividing the list up into segments, with a given *start* and an *end*, where the function at index *end* is not included. A method implementing a formula for the division of the number of functions per iteration is shown in Figure 3.3. Since generation of all functions in ≤ 4 variables takes seconds, this solution was chosen – but as it is inefficient compared to other solutions, it was changed for $n = 5$, discussed in Section 4.4.1. Line 12 in Figure 3.3 refers to code not relevant to this concept, and can be viewed in full in Appendix B.

```
## Division of functions per iteration
division = 16  # because this was what was available
current_session = input("Iteration (0..%d)?: " % (division-1))
function_iterations = range(0, 2^2^n, 2^2^n/division)

start = function_iterations[current_session]
if (current_session == division-1):
    end = 2^(2^n)
else:
    end = function_iterations[current_session+1]

# [...]

## SETUP: Construction of Boolean Polynomials
bool_pols = generate_functions(x)[start:end]
```

Figure 3.3: Code: Dividing number of functions into iterations, $n \leq 4$

3.2.2 Generating invertible matrices and vectors

Any $n \times n$ matrix over $\mathbb{F}_2$ can be represented by a n^2-length binary number by concatenation of the rows of the matrix, e.g. for $n = 3$:

$$\begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 \end{bmatrix} = [a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9,],$$

where $a_1, \ldots, a_9 \in \mathbb{F}_2$. In SageMath, this is one of the approaches for the construction of a matrix, and was chosen for its simplicity. As there are 2^k

binary numbers of length k (see Section 2.1.2), for $k = n^2$ there are then 2^{n^2} possible $n \times n$ matrices over $\mathbb{F}_2$. A method to generate all *invertible* $n \times n$ matrices over $\mathbb{F}_2$ was needed, and two main algorithms were analysed:

1. **Computational (brute-force):**
   Generate all 2^{n^2} possible matrices and use the built-in *is_invertible()*-method in SageMath as a filter.

2. **Mathematical:**
   Generate all independent vector combinations and constructing the matrices from said vectors.

In both cases, the correct number of invertible matrices was generated (see Table 2.4), but the methods were inefficient, especially for $n > 4$ – and a method that could be used efficiently for both $n = 4, 5$ was preferred. Therefore, a hybrid method based on the two approaches was designed, using the filtering method from the computational approach, and the vector combination from the mathematical approach. In this method, which is given in Figure 3.4, the independency criteria of the mathematical approach is disregarded, and instead, all vector permutations are combined. Then, using the built-in method to filter out those combinations that are dependent, the invertible matrices are found. This reduces the generation time down (for $n = 5$) from a few hours to approximately 35 minutes, on the resources that were available at the time. A better solution to matrix generation may exist, but the solution presented here is satisfactory for its use related to this thesis.

Generation of all vectors in $\mathbb{F}_2^n$ is a trivial task, and is merely the generation of all binary numbers in the inclusive range $[0,\ 2^n - 1]$.

For the brute-force implementation of thickness calculation, the main requirement was to store each function f and the minimal function f_{min} that f was mapped to. An additional requirement was to store each matrix A and vector $\mathbf{b}$ which provide the affine transformation that maps f into f_{min}. At first, the n^2-length binary number of the matrix, and the n-length binary number of the vector was stored, but tests showed that this made each output-file very large, and both slowed down the full program and the analysis programs. Instead, a method called *matrix_value* was programmed, using the integers in base-10 that generates each *vector* – when converted to a binary number – to calculate the integer in base-10 that would generate each *matrix*, also when converted to a binary number.

In addition, to avoid multiplying each matrix with the vector containing all variables (i.e. $\mathbf{x} = (x_1, \ldots, x_n)$) every time a function would be analysed, the calculation of $A\mathbf{x}$ was stored instead of the matrix itself, saving up to

$$\left(\prod_{i=0}^{n-1} (2^n - 2^i) \right) * (2^{2^n} - 1)$$

operations, where the left term here is the number of invertible matrices in n variables, from (2.5) – in short, the given formula describes that the matrix maps are calculated once instead of once for each function.

However, the operation of adding the matrix maps with the vectors $\mathbf{b}$ that combine into the affine transformations required, *is* done once per function. This choice was made because of the resulting size of the list containing the

```
def generate_invertible_matrices_and_vectors(n_var, M, x_vector):
    binary_list = []
    base = ['0'] * n_var
    vectors = [(0, vector(GF(2), base))]

    # Vector generation
    for index in range(2^n_var)[1:]:
        binary = list(bin(index))[2:]
        vec = base[len(binary):] + binary
        binary_list.append((index, vec))
        vectors.append((index, vector(GF(2), vec)))

    # Vector permutations
    permutations = Arrangements(range((2^n_var)-1), n_var).list()

    # Invertible matrix generation
    matrices = []
    for permutation in permutations:
        accumulator = []
        value_list = []
        for index in permutation:
            value_list.append(binary_list[index][0])
            accumulator += binary_list[index][1]
        inv_matrix = M.matrix(accumulator)
        if inv_matrix.is_invertible():
            generating_int = matrix_value(n_var, value_list)
            matrix_map = inv_matrix*x_vector    # matrix maps
            matrices.append((generating_int, matrix_map))
    return matrices, vectors

# Calculate generating integer for given matrix
def matrix_value(n_var, value_list):
    return sum( [ (value_list[i] * 2^(n_var*(n_var-i-1)))
                for i in range(len(value_list)) ] )
```

Figure 3.4: Code: Matrix maps and vector generation in $n \leq 4$

affine transformations (see Table 2.4 for the numbers of such) – when tests were run to attempt to store all of them, the computers available at the time crashed, because of memory errors.

The method for matrix and vector generation, and the method for matrix value calculation is given in Figure 3.4. In the following $n = 5$ program, this method was changed slightly, as the requirement of storing which matrix and vector that created the map was abandoned, as they were deemed not of interest.

In Figure 3.4, vector generation is accomplished in Lines 3–11, appending a tuple consisting of *(generating integer, vector)* to a list, which – when the process is completed – contains all vectors. Line 10 contains the vectors used for matrix generation further below.

The *permutations* created in Line 14 is using SageMath's built-in method *Arrangements()*, and consists of all possible permutations of indices $0..(2^n - 1)$ of length n. This is used in the following for-loop to check all combinations of vectors of whether they combine into an invertible matrix or not. If they do, the generating base-10 integer of the matrix is calculated, and a map is created by multiplying the matrix with the variable-vector. Each matrix map is stored as a

tuple: *(generating integer, matrix map)*. Note that there is a known issue with the Permutation module in SageMath (discussed in the documentation of its package, found in [25]), but rigorous testing confirmed this issue did not affect its use with this method.

3.2.3 Calculating Algebraic Thickness, n = 2,3,4

n	Number of mapped functions
2	384
3	344 064
4	21 139 292 160
5	1 374 301 573 789 777 920

Table 3.1: *Brute-force*: number of transformations to check in n variables

Having all invertible matrix maps, all vectors, and all Boolean polynomials generated, the calculation of algebraic thickness of each function consists of creating an affine transformation map for each vector with all matrix maps, then applying said map to each function – and finally checking the number of monomials in each resulting mapped function, storing the minimal monomial count.

Given in Table 2.4 is the complete number of affine transformations needed to be iterated through for each Boolean function in $n \leq 5$ variables. The total number of functions to check the monomial count of, for n variables, is given in Table 3.1, using the formula

$$\text{Number of mapped functions} = 2^{2^n} * 2^n * \left(\prod_{i=0}^{n-1} 2^n - 2^i \right)$$

where the first term is the number of Boolean functions, the second is the number of vectors, and the third is the number of invertible matrices. Note that these resulting functions are not necessarily unique, in fact most of them are duplicates. This is mentioned briefly following Definition 2.4, i.e. for a function $f(x_1, x_2) = x_1$, there are many instances where there is no change under affine transformation, e.g. $f(x_1, x_1 + x_2) = x_1 = f(x_1, x_2 + 1)$. Nevertheless, the resulting functions must all be checked in relation to the function being mapped.

Iteration through all of these functions is a trivial task for $n = 2, 3$, and a rather lengthy – but still manageable – task for $n = 4$. Initial tests approximated the execution time to 32 days, which is why the iteration approach was used. The corresponding time frame for $n = 5$ variables was calculated by timing the brute-force program when applying all affine transformations to 10 Boolean functions of various degrees and monomial counts in $\mathcal{B}_5$, and then averaging the time spent. The average time found was approximately 5 hours, which resulted in a time estimate of 2 450 000 years – when the brute-force program is run on one iteration. While this estimate is not at all precise, it served its purpose to show that a better method of calculating thickness was needed, discussed in Chapter 4.

Programmed method of thickness calculation

The method given in Figure 3.5 is the method incorporating the algebraic thickness calculation for each function in $n = 4$ variables – and also $n = 2, 3$, trivially, although Line 24 must then be modified. The input variables of the method are (1) all Boolean polynomials sorted by algebraic degree, (2) all invertible matrix maps, and (3) all vectors; as generated by the methods described above. The functions are sorted by degree in this program, because – at the time – it was suspected that this could be of use for $n = 5$, inspired by Dickson's theorem (as mentioned in Section 2.3, showing that quadratic functions are well-understood). Missing from this program are the previously mentioned quality-of-life sections, not relevant to the logic of the program.

The method iterates through each degree from $1 \ldots n$, storing information about the current minimal number of monomials in each mapped function, which matrix and vector was used, and the mapped function itself. When the lists of matrices and vectors have been exhausted (or the function was mapped to the absolute minimal number of terms possible for a nonzero function, i.e. 1), nonlinearity is calculated for the function, using the previously mentioned *boolean_function*-package, at Line 42. Then each result is printed to a text file, which is saved and closed – in case of power failure, or other unexpected problems, during this time consuming process.

Line 25 in Figure 3.5 uses the built-in method *.monomials()*, which creates a list containing all the monomials in the function. Counting the size of this list then gives the monomial count of the function. Later on, it was discovered that using the built-in function *.set()* is much faster, and is used in programs for $n = 5$. Other minor improvements to the program can also be done, e.g. changing the initialisation of the minimal monomial count (Line 13) to the original f's monomial count, and the same corresponding change to *min_new_f*, in Line 16. All of these changes were addressed and implemented for $n = 5$.

For $n = 4$, because this program would have had to run for approximately 32 days, it was run as 16 iterations, where each iteration calculated thickness for $\frac{65536}{16} = 4096$ functions. These iterations were then run on various available computers, and the full process took approximately 6 days to complete. The resulting data sets where zipped together and checked for validity – i.e., a program verified that all Boolean polynomials where iterated through, and that the stored matrix and vector integers *in actuality* mapped the functions to the saved functions in the data set, before analysis was started.

After analysis of the data set (gathered in Chapter 5, see Section 5.1), and inspection of what functions were being mapped *to*, it was found that, in most cases, the functions of the same thickness were mapped to the same – or in some cases, a small number of similar – function(s). These were then collected and further analysed, the results of which introduced a property that was used for calculation of the thickness distribution in $n = 5$ variables. This is the subject of the following chapter.

```
from sage.crypto.boolean_function import BooleanFunction

def calc_thickness(bool_pols_by_deg, matrices, vectors):
    n_matrices = len(matrices)
    n_vectors = len(vectors)

    for bool_pols in bool_pols_by_deg:
        n_bool_pols = len(bool_pols)

        for f in bool_pols:
            iter_mat = 0

            min_mapped_monomials = 2^n + 1 # global var n
            min_matrix = 0
            min_vector = 0
            min_new_f = 0

            while (iter_mat < n_matrices):
                iter_vec = 0
                while (iter_vec < n_vectors):
                    # Map the function, for n = 4
                    map_ = matrices[iter_mat][1] +
                            vectors[iter_vec][1]
                    new_f = f(map_[0], map_[1], map_[2], map_[3])
                    mapped_monomials = len(new_f.monomials())

                    # Check if fewer monomials
                    if (mapped_monomials < min_mapped_monomials):
                        min_mapped_monomials = mapped_monomials
                        min_matrix = matrices[iter_mat][0]
                        min_vector = vectors[iter_vec][0]
                        min_new_f = new_f

                    # Move on, if minimal monomials
                    if (mapped_monomials == 1):
                        iter_mat = n_matrices
                        iter_vec = n_vectors
                    iter_vec += 1
                iter_mat += 1

            # Write data to file
            nonlin = BooleanFunction(min_new_f).nonlinearity()
            toText = open(file_name, 'a')
            toText.write("%d|%d|%d|%d|%s|%s\n" %
            (min_mapped_monomials, nonlin, min_vector, min_matrix,
    f, min_new_f))
            toText.close()
    return
```

Figure 3.5: Code: Calculating thickness distribution for $n = 4$

Chapter 4

Calculating thickness distribution for $n = 5$

From the data collected by execution of the program discussed in Chapter 3, the Boolean functions that showed the most promise were the functions with the same monomial count (in their ANF) as their corresponding algebraic thickness. Naturally, any Boolean function can be mapped to (at least) one of these, following the definition of algebraic thickness (see Definition 2.10). Further study of this class of Boolean functions showed promise in relation to calculation of algebraic thickness in higher numbers of variables. In Section 4.1, this class of functions is defined and explored, and is further applied in the definition of a related concept, in Section 4.2. Together, these two associated concepts built the foundation of calculating the distribution of algebraic thickness for $n = 5$.

Remark. This thesis does not claim these definitions as *new concepts* within the realm of Boolean functions – attempts were made to find equivalent definitions in related work, that define these concepts clearly in relation to algebraic thickness, but none could be found. Carlet defines algebraic thickness to be "the minimum number of monomials with non-zero coefficients in the ANF of the functions $f \circ \mathcal{A}$", and as described in Section 4.1, the class of functions defined relates specifically to the functions with the mentioned minimum number of monomials. The following concepts may not be ground-breaking, but – as will be presented – proved helpful in the work at hand.

4.1 Rigid functions

Given a Boolean function f, the algebraic thickness $\mathcal{T}(f)$ – by Definition 2.10 – is the minimum number of terms in the ANF of the affine transformations of said function. Let f_{min} be (one of) the function(s) with minimum number of terms in its ANF that f can be mapped to, through an affine transformation. The algebraic thickness of f_{min} itself is then, clearly, the number of terms in the ANF of itself.

Definition 4.1. (Rigid Boolean functions)
A Boolean function f with m monomials in its ANF, where $\mathcal{T}(f) = m$, is a *rigid function.*

In other words, a rigid function is any Boolean function f where the monomial count of f is equal to the algebraic thickness of f. By this definition, a rigid Boolean function cannot be mapped to a function with *lower* monomial count, through *any* affine transformation, by the very definition of algebraic thickness. Furthermore, *any Boolean function can be mapped to a rigid function.* The reason for this should be clear, but for completion, this short proof is included:

Proof. Given a Boolean function $f \in \mathcal{B}_n$, let g be a function in the orbit of f (through affine transformations), where the monomial count of g is equal to $\mathcal{T}(f)$. If g is not a rigid function, then g does not have the minimum monomial count in its orbit. Suppose h is in the orbit of g, and has lower monomial count than g. Since f maps to g and g maps to h, then by composition of transformations, f maps to h as well. Thus, we reach a contradiction. □

The set of all rigid functions in n variables will be denoted by $\mathcal{S}_n$. Experimentally, it was found that $\mathcal{S}_n \subset \mathcal{S}_{n+1}$, for $n \leq 3$, and – by further examination – this was found to be true for any n, as summarized in the following theorem.

Theorem 4.1. *All rigid functions in n variables are also rigid functions in $(n+1)$ variables, i.e. $\mathcal{S}_n \subset \mathcal{S}_{n+1}$.*

Remark. As is customary in this area (for easy writing), in the following proof, we disregard the usual linear algebra convention of matrix-vector multiplication and regard $\mathbf{x}$ and $\mathbf{b}$ both as a row- and a column vector, when there is no danger of confusion.

Proof. Let $f \in \mathcal{S}_n$ with $\mathcal{T}(f) = t$. We embed f in $n+1$ variables, and we denote its embedding by $\tilde{f}$, such that $\tilde{f}(x_1, \ldots, x_n, x_{n+1}) = f(x_1, \ldots, x_n)$. Let a non-zero affine transformation of the input of $\tilde{f}$ be given by $\mathbf{x} \mapsto \tilde{A}\mathbf{x} + \mathbf{b}$, where $\tilde{A}$ is a $(n+1) \times (n+1)$ matrix. We label the first n rows and n columns in $\tilde{A}$ by A and so,

$$\tilde{A} = \begin{pmatrix} & & a_{1,n+1} \\ & A & \vdots \\ a_{n+1,1} & \cdots & a_{n+1,n+1} \end{pmatrix}.$$

Thus,

$$\tilde{A}\mathbf{x} + \mathbf{b} = \begin{pmatrix} A\mathbf{x} + x_{n+1} \begin{pmatrix} a_{1,n+1} \\ \vdots \\ a_{n,n+1} \end{pmatrix} + \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix} \\ a_{n+1,1}x_1 + \cdots + a_{n+1,n+1}x_{n+1} + b_{n+1} \end{pmatrix},$$

and so

$$\tilde{f}(\tilde{A}\mathbf{x} + \mathbf{b}) = f\left(A\mathbf{x} + x_{n+1} \begin{pmatrix} a_{1,n+1} \\ \vdots \\ a_{n,n+1} \end{pmatrix} + \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix} \right),$$

from which our claim is inferred. □

n	Number of rigid functions
0	2
1	3
2	6
3	28
4	588
5	211 259

Table 4.1: Number of rigid functions in $n \leq 5$ variables

In summary, the introduction of x_{n+1} does not induce any further monomial eliminations not already possible in n variables. Therefore, for a rigid function f with monomial count t,

$$\mathcal{T}_n(f) = t = \mathcal{T}_{n+1}(f).$$

Furthermore, an immediate result of this theorem is given in Corollary 4.1.1.

Corollary 4.1.1. *For any Boolean function f in n variables:*

$$\mathcal{T}_n(f) = \mathcal{T}_{n+1}(f)$$

Proof. Given a rigid Boolean function f in n variables, let $\mathcal{A}_n(f)$ be the the orbit of f through all nonzero affine transformations, and let $\mathcal{T}_n(f) = t$. As we know, from the definition of algebraic thickness, any Boolean function $g \in \mathcal{A}_n(f)$ satisfies $\mathcal{T}_n(g) = t$, as well. Since f is rigid, $\mathcal{T}_{n+1}(f) = t$, by Theorem 4.1. Clearly, then, $\mathcal{A}_n(f) \subseteq \mathcal{A}_{n+1}(f)$, by the very same affine transformations as in n variables (leaving the new variable x_{n+1} mapped to itself), and therefore all functions in $\mathcal{A}_n(f)$ have thickness t in $n+1$ variables, as well. □

The definition for rigid functions is closely related to Carlet's definition for algebraic thickness, given in Definition 2.10. A property of algebraic thickness in relation to its distribution, is given in the following lemma.

Lemma 4.2. *Given all Boolean functions with monomial count k in n variables: if there are* no rigid functions *with k monomials, then there are no functions in n variables with $\mathcal{T}_n = k$.*

In other words, in n variables, if there are no rigid functions with monomial count k, then there are no functions in n variables with algebraic thickness k.

Lemma 4.2 is easily shown by studying Definitions 2.10 and 4.1, observing that if all functions of a certain size can be mapped to another function with fewer monomials, then the algebraic thickness cannot be that specific size.

The realization of this lemma, after defining the concept of rigid functions, was the inception of the program described in Section 4.3, and the program given (partially) in Section 4.4 (and in full in Appendix C). Therefore it is stated as a lemma, here.

The distribution of the number of rigid functions in $n \leq 5$ variables is listed in Table 4.1, where: for $n \leq 4$ variables, these numbers were collected from

analysis of the data sets calculated by brute-force, and for $n = 5$, the number was (along with double-checking values for $n < 5$) collected from analysis of the data sets calculated by the program described in Section 4.4.

Having the concept of rigid functions clearly defined can be helpful in the search for the distribution of algebraic thickness for $n > 4$, because of the size (and growth in $n+1$) of $\mathcal{B}_n$. Instead of iterative calculations through *all* Boolean functions (which can be considered impossible for $n > 5$ by modern computing standards), searching for rigid functions and – most importantly – disregarding non-rigid functions, should improve the efficiency of any program (at the very least, it improves the program given in Chapter 3) attempting this calculation by a similar method, effectively increasing the potential search space.

Regarding the analysis of the algebraic thickness of Boolean functions, rigid functions are important – both in relation to the definition of algebraic thickness, and for use in programs analysing Boolean functions, as argued above. Determining which functions are rigid functions in n variables yields information regarding the thickness distribution in $n + 1$ variables as well, by Theorem 4.1. Furthermore, by Corollary 4.1.1, unveiling the distribution of all functions in $\mathcal{B}_n$ immediately gives the distribution of 2^{2^n} functions in $\mathcal{B}_{n+1}$ – which may be a small portion compared to $2^{2^{n+1}}$, but is nonetheless a start.

However, since some of the rigid functions found shared a similar structure, further analysis showed that some of them were *affinely equivalent*. Another definition was needed for thickness calculation in 5 variables, which is given in Section 4.2.

Further properties of rigid functions and algebraic thickness are discussed and presented as conjectures in Section 6.1.6.

4.1.1 Examples of rigid functions

The functions in $\mathcal{S}_0, \mathcal{S}_1, \mathcal{S}_2$ (i.e., the rigid functions in $n = 0, 1, 2$ variables) are shown here in full:

$$\mathcal{S}_0 = \{0,\ 1\}$$
$$\mathcal{S}_1 = \{0,\ 1,\ x_1\}$$
$$\mathcal{S}_2 = \{0,\ 1,\ x_1,\ x_2,\ x_1x_2,\ (x_1x_2 + 1)\}$$

Since the sets $\mathcal{S}_3, \mathcal{S}_4$ are of rather large sizes (28 and 588, respectively), they will not be listed here. However, a select few examples will be presented in Section 4.2.1.

4.2 Representative functions

n	Number of equivalence classes
1	3
2	5
3	10
4	32
5	382
6	15 768 919

Table 4.2: Number of equivalence classes in the affine group, from [16]

For a given Boolean function f, the *orbit through affine transformations* of this function consists of *all Boolean functions* that f can be mapped to, through all affine transformations – as described in Section 2.2. Any function in $\mathcal{B}_n$ is contained in only *one* orbit, thus the orbits are *equivalence classes*, as they divide the set $\mathcal{B}_n$ into disjoint subsets (the intersection of any two such is the empty set). The number of equivalence classes in the general affine group (i.e., under affine transformations, as mentioned in Section 2.2) is given in Table 4.2, as presented by Harrison in [16].

By uncovering *one* function ϕ for each of these orbits, every function in n variables can be generated from a corresponding ϕ, by iteration through all affine transformations for each one. Calculating the algebraic thickness of each ϕ yields the thickness distribution for all functions in $\mathcal{B}_n$, as $\mathcal{T}$ is (trivially) an affine invariant. Since these ϕ would be representing their orbits, the name *representative function* was chosen. As the rigid functions are the functions with the minimum number of monomials in their ANF, these representative functions were chosen to be the *smallest* rigid functions in their orbit. The following definition contains what was used for choosing such a function, defining what 'smallest' would mean.

Definition 4.2. (Representative Boolean functions)
Let f be any Boolean function in n variables. In the orbit of f through all affine transformations, there exists at least one rigid function f_{min}. The *representative function* is chosen among the rigid functions in the orbit of f, using the following method:

I If f_{min} is the only rigid function in the orbit, then f_{min} is the representative function of the orbit of f.[1]

II If there are more than one rigid function in the orbit, we choose the *smallest* rigid function – i.e. the function with minimal sum of the degrees of each monomial in its ANF – to be the representative function.

III If there are still more than one function to choose from, we choose the smallest function (as in II) that can be represented by *the lowest indexed variables*, in lexicographical order, *in descending order by degree of monomials.*

[1]This is only applicable for the trivial cases, i.e. the functions 0, 1, and $x_1 \cdots x_n$. All other representatives will have more than one rigid function in their orbit, by variable permutation.

Part III of this definition is purely implementation specific and the choice of representative from the rigid functions of the orbit will not affect any properties related to algebraic thickness. The last sentence of this part (in italics) is further explained in Example 4.2. As with rigid functions, the set of all representative functions will be denoted as $\mathcal{R}_n$ for representatives in n variables. Trivially, $\mathcal{R}_n \subseteq \mathcal{S}_n$.

Representative functions can be a tool both for reducing the set of functions to discuss in the context of algebraic thickness, without losing valuable information, and to help in attempts at finding patterns for the generation of rigid functions.

4.2.1 Examples of choosing representative functions

As with the rigid functions, the first three sets of representatives can be shown in full as examples:

$$\begin{aligned}&\mathcal{R}_0 = \mathcal{S}_0 = \{0,\, 1\}\\ &\mathcal{R}_1 = \mathcal{S}_1 = \{0,\, 1,\, x_1\}\\ &\mathcal{R}_2 = \{0,\, 1,\, x_1,\, x_1x_2,\, (x_1x_2 + 1)\},\end{aligned}$$

that is, $\mathcal{R}_2 = \mathcal{S}_2 - \{x_2\}$. Because both x_1 and x_2 are in the same orbit, x_1 is chosen as the representative function, because of the lower indexing.

The full list of representatives for $n \leq 4$ can be seen in Appendix D. For now, examples of how the representatives are chosen will be listed, for $n = 3$ variables.

Example 4.1. For $n = 3, \mathcal{T}_3 = 3$:
In the only orbit with maximum thickness in $n = 3$ variables, there are 9 rigid functions:

1. $x_1x_2x_3 + x_3 + 1$
2. $x_1x_2x_3 + x_2 + 1$
3. $x_1x_2x_3 + x_1 + 1$
4. $x_1x_2x_3 + x_2 + x_3$
5. $x_1x_2x_3 + x_1 + x_3$
6. $x_1x_2x_3 + x_1 + x_2$
7. $x_1x_2x_3 + x_2x_3 + x_1$
8. $x_1x_2x_3 + x_1x_3 + x_2$
9. $x_1x_2x_3 + x_1x_2 + x_3$

Since there are more than one rigid function, Part I of Definition 4.2 does not apply. For Part II, we find that in (7)–(9), the sum of monomial degree of each function is 6; for (4)–(6), the sum is 5; and in (1)–(3), it is 4. Thus, by Part II, one of the functions in (1)–(3) is the representative function.

The monomial with highest degree is equal for all three functions (for an instance where this is not the case, refer to Example 4.2), thus the monomial

with second-highest degree is examined. Of these, x_1 is lowest indexed, and therefore the function in (3) is the representative of its orbit, by Part III in Definition 4.2.

Example 4.2. For $n = 3, \mathcal{T}_3 = 2$:
In one of the orbits with $\mathcal{T}_3 = 2$, there are three rigid functions:

1. $x_1x_2 + x_3$
2. $x_1x_3 + x_2$
3. $x_2x_3 + x_1$

These three rigid functions all have the same sum of degrees of each monomial, and as such, are all possible representatives. However, *the lowest indexed variables* of these functions could be ambiguous here, which is why it is specified, in Part III of Definition 4.2 (in italics), that the lowest index for the monomials with higher degrees is preferred. Thus the monomials $x_1x_2 < x_1x_3 < x_2x_3$, and therefore $x_1x_2 + x_3 < x_1x_3 + x_2 < x_2x_3 + x_1$. Thus, the chosen representative is $x_1x_2 + x_3$, given in (1), here.

In short, all 2^{2^n} Boolean functions in n variables can be mapped to one (and only one) representative function – the representative function is a representative of its *orbit*, and therefore there are the same number of representative functions in n variables as there are orbits in n variables, which is equal to the number of equivalence classes under action of the general affine group. A further discussion on this subject will be continued in Chapter 5.

4.3 Focusing on monomial counts

When the algebraic thickness distribution for $n = 2, 3, 4$ was calculated (as discussed in Chapter 3), the focus was put on the algebraic degree of the functions – both because algebraic degree is an affine invariant, and the known results regarding quadratic functions, as given in Section 2.3 – when iterating through all elements of $\mathcal{B}_4$. However, having the concept of rigid functions defined, it became clear that focusing on the number of monomials in the ANF of the functions would prove to be more efficient in the context of revealing the distribution of algebraic thickness.

Defined in Section 2.2, the term *monomial count* of a Boolean function is used in this thesis to refer to the number of monomials in a function's ANF. The number of Boolean functions in n variables that have exactly m monomials in their ANF is $\binom{2^n}{m}$, since this number equals the number of ways of "choosing" m nonzero coefficients among the possible 2^n monomials. (See Section 2.1.2 for the definition of the binomial coefficient, given in (2.2)). Similarly, the number of Boolean functions with at least m monomials is the sum of the binomial coefficients $\binom{2^n}{i}$, where $i \geq m$, that is,

$$\sum_{i=m}^{2^n} \binom{2^n}{i}. \tag{4.1}$$

A full distribution of the number of functions with monomial count m in $n = 2, 3, 4, 5$ variables is given in Table 4.3. The binomial coefficient distribution follows a *normal distribution*, which is seen in that the highest number of Boolean functions have $m = 2^{n-1}$ monomials (midpoint), and the number of functions decreases at the same rate for $m \pm 1, m \pm 2, \dots$.

Inspired by the formula given in (4.1), all functions in $n = 5$ variables with monomial count greater than or equal to $\left\lfloor \frac{2}{3} 2^5 \right\rfloor = 21$ were checked experimentally for *rigidness* (i.e., whether the functions were rigid or not), based on Carlet's upper-bound for algebraic thickness in n variables (see Section 2.3.2). As expected, no rigid functions were found – and unexpectedly, this process took a few hours.

Continuing this process, all monomial counts of descending values were checked, and it was found that – as the values decreased – the time spent on checking the functions increased. However, not because of the increase in the number of functions ($m = 16$ was completed surprisingly fast). Rather, it was because the number of functions with lower monomial counts decreased as well – thus the time spent to find a function with lower monomial count for each function increased (the time to execute the program for 9 monomials took significantly much longer than 10, which again took longer than 11, etc.). The program used for this calculation is what the program described in Section 4.4 is based on, and will therefore not be explained further.

Ultimately, the first monomial count where a rigid function could be found, was $m = 8$ (i.e., first, in descending order). Thus, the maximum thickness of $n = 5$ is 8, by Lemma 4.2.

Monomial count	$n = 2$	$n = 3$	$n = 4$	$n = 5$
0	1	1	1	1
1	4	8	16	32
2	6	28	120	496
3	4	56	560	4 960
4	1	70	1820	35 960
5	-	56	4368	201 376
6	-	28	8008	906 192
7	-	8	11 440	3 365 856
8	-	1	12 870	10 518 300
9	-	-	11 440	28 048 800
10	-	-	8008	64 512 240
11	-	-	4368	129 024 480
12	-	-	1820	225 792 840
13	-	-	560	347 373 600
14	-	-	120	471 435 600
15	-	-	16	565 722 720
16	-	-	1	601 080 390
17	-	-	-	565 722 720
18	-	-	-	471 435 600
19	-	-	-	347 373 600
20	-	-	-	225 792 840
21	-	-	-	129 024 480
22	-	-	-	64 512 240
23	-	-	-	28 048 800
24	-	-	-	10 518 300
25	-	-	-	3 365 856
26	-	-	-	906 192
27	-	-	-	201 376
28	-	-	-	35 960
29	-	-	-	4 960
30	-	-	-	496
31	-	-	-	32
32	-	-	-	1
Sum	16	256	65 536	4 294 967 296

Table 4.3: Distribution of functions in $n \leq 5$ variables over monomial count

4.4 Program: Finding representatives in $n = 5$

Monomial count	Number of functions
0	1
1	6
2	28
3	134
4	625
5	2 674
6	10 195
7	34 230
8	100 577
9	258 093

Table 4.4: Distribution of variable-permutation unique functions in $\mathcal{B}_5$

In having found the maximum thickness of functions in $n = 5$ variables, and changing the focus of thickness calculations from algebraic degree to the monomial count of functions in $\mathcal{B}_5$, there is a total of

$$\sum_{i=0}^{8} \binom{2^5}{i} = 15\ 033\ 173$$

Boolean functions to search through for rigid functions (see Table 4.3, in column $n = 5$, sum of rows 0–8). Although this number of functions is significantly smaller than $|\mathcal{B}_5| = 2^{2^5}$, it is still of considerable size.

However, as mentioned in the remark found below Definition 2.4 (affine transformations), *variable permutations* of Boolean functions is a subset of all affine transformations – therefore, the removal of duplicate functions, after mapping all functions to their lowest indexed variable version[2], will not remove any possible representatives. The total number of variable permutations in 5 variables is $5! = 120$, which is relative small in comparison with the number of affine transformations in total (see Table 2.4), and can be completed in a comparatively short amount of time. Therefore, by mapping all 15 033 173 functions with monomial count 8 or lower to their lowest indexed function – which can be considered a *variable-permutation representative* – the number of functions to search through for rigid functions is lowered to 148 470. The distribution of the number of these variable-permutation unique functions within each monomial count value, is shown in Table 4.4. Note that 148 470 is the sum of rows 0–8 – row 9 is included only to show how the number of variable-permutation unique functions grows.

The program for processing the functions in these nine (0–8) monomial count values will be described in detail in the section below, and is shown in full in Appendix C. After completion, the output were the 382 representative functions as desired (cf. Table 4.2), which will be further discussed in Chapter 5.

[2] As in Definition 4.2 Part III, using the code given in Figure 4.1.

```python
# Maps function to lowest indexed form
def map_to_smallest_index(f):
    smallest = f
    for f_map in x_map: # x_map is global variable, see App. C
        mapped_f = f(f_map[0], f_map[1], f_map[2],
                     f_map[3], f_map[4])

        if (mapped_f > smallest): # note: x1 > x2 in .pbori
            smallest = mapped_f
    return smallest
```

Figure 4.1: Code: Mapping a function into its lowest indexed form

4.4.1 Constructing relevant Boolean functions

The Boolean functions that were analysed, by the program described in Section 4.4.3, were generated by the same method shown in Section 3.2.1, i.e. Figure 3.2 – however, as explained, only those functions with monomial count less than or equal to 8 were collected; discarding all others. The evaluation of monomial count was accomplished by using the built-in method *f.set()*, and checking the length of the resulting set, where *type(f)* = *pbori.BooleanPolynomial*, in SageMath [25]. This method was used instead of the previously mentioned *f.monomials()* in Chapter 3, as tests showed a significant improvement in efficiency, and both methods served the same purpose.

The complete list of the 15 033 173 functions mentioned above were then re-mapped to their lowest indexed variable equivalent function, using the code shown in Figure 4.1, resulting in a list of size 148 470, which was divided and written into one text-file for each monomial count value. Trivially, the only function in monomial count 0, and the six functions in monomial count 1, were already in representative form, after this variable permutation process. The full program in Appendix C was run on the remaining seven (2–8) functions sets.

Collection of Boolean functions from the text files created was done by methods listed in the appendix, i.e. converting a string into a Boolean polynomial. This could also have been done by using the built-in conversion method in SageMath – but the listed method (*convert_to_boolpol()*) was found to be faster for the purposes in this context.

4.4.2 Generating invertible matrices and vectors

The affine transformations required for each function were generated as in the brute-force implementation, but with some minor adjustments (see Figure 4.2):

1. Removed the method *matrix_value()* and the variables involved with it, as the requirement of storing what matrices and vectors map the functions to their representative was abandoned.
2. Used *int*s for generation of vectors instead of *string*s
3. Changed conversion of binary number-strings into vectors to use the built-in function *map()*, as *int*s were used.

In total, this improved the matrix generation time from 35 minutes to 25 minutes, when run on the available resources (for $n = 5$).

```
def generate_invertible_matrices_and_vectors(n_var, M, x_vector):
    binary_list = []
    base = [0] * n_var
    vectors = [vector(GF(2), base)]  # 0-vector added here

    for index in range(1, 2^n_var):
        binary = map(int, bin(index)[2:])
        vec = base[len(binary):] + binary
        binary_list.append(vec)
        vectors.append(vector(GF(2), vec))

    permutations = Arrangements(range((2^n_var)-1), n_var).list()

    matrices = []
    for permutation in permutations:
        accumulator = []
        for index in permutation:
            accumulator += binary_list[index]
        inv_matrix = M.matrix(accumulator)
        if inv_matrix.is_invertible():
            matrix_map = inv_matrix*x_vector
            matrices.append(matrix_map)
    return matrices, vectors
```

Figure 4.2: Code: Matrix maps and vector generation in $n = 5$

4.4.3 Searching for representatives in $n = 5$

In the folder where the program was stored, a complete list of all representatives currently found was kept and updated regularly, as a text file. After collecting the functions with a specific monomial count value (as in Section 4.4.1), the program collected all current representatives and stored them in a *set()* – i.e. the built-in data structure *set()* in Python. This structure was used, because "A set object is an unordered collection of distinct *hashable* objects" [22], which allows efficient "element *in* set" computation. This made it possible to check if the current function had already been found as a representative, without significant impact on the run time of the program.

The code shown in Figure 4.3 is the method used for analysis of all functions in a set monomial count value, and contains two global variables: *n_monoms*, the monomial count of the current iteration; and *representatives*, a *set()* containing all currently found representatives with monomial count values equal to the specified *n_monoms*. As with the methods described in Chapter 3, various "quality-of-life" method calls for printing out current positions – and saving the positions of the iterations, in case of power failure – have been omitted for readability.

The method (in Figure 4.3) makes an initial assumption, for each function f, that f is rigid, and attempts to disprove this assumption by checking the monomial count of each affine transformation of said function. A variable *min_f* is initialised, which stores the *smallest* rigid function f can be mapped to through the process that follows (*smallest*, as in Definition 4.2 Part II – Part III is applied further below). Then, three important checks are made, for all f_{map} that f maps to through any affine transformation:

1. (Lines 17–21): If f can indeed be mapped to a function with a lower monomial count value, the function is proven not to be rigid, and is abandoned.
2. (Lines 23–28): If f_{map} equals a representative that is already found, f is abandoned.
3. (Lines 30–34): If none of the above are true, in which case f *may* be rigid, a check is made to see if f_{map} is smaller than the current minimal function f maps to, as explained above.

Iteration through the affine transformations is accomplished through the use of *while*-loops, because of the added benefit of being able to break out of the loop. In general, breaking out of a loop is not good code practice (mainly for readability, as it can disrupt the flow), but because the difference in time elapsed *per function* between exhausting the set and breaking out of the loop can be more than 10 hours, this feature is a vital part of the algorithm.

After the complete set of affine transformations has been exhausted – or, if f was proven not to be rigid – a final check is made for f's rigidness. If f is indeed rigid (and because of Lines 23–28 it hasn't been found previously), min_f is mapped to its lowest indexed variable form (see Figure 4.1), reflecting Part III of Definition 4.2. The resulting representative is then saved to a text file, and added to the *set()*-structure containing all current representatives.

Remark. The "bottleneck" of finding representatives of functions in five variables is the number of affine transformations to go through for each function – but also the fact that the number of affine transformations is much larger than the number of functions in any orbit (cf. the *pigeonhole principle*). This means that there are several affine transformations that, for each f, maps f to the same function. However, since there is no way of predicting, as far as we know, *which* transformations will do this, it cannot be avoided.

In an attempt to work around this, a *set()* object containing all functions f had previously been mapped to, together with a check for each f_{map} of whether it had been checked previously, was added; and tests were run for a test set containing a few functions in $n = 5$ variables. It was found that, compared to a program without this set, the computation time was almost unaffected, but the memory usage was increased drastically. Because of this, it was not included in the final program.

The final program used for finding all 382 representatives in $n = 5$ variables is listed in Appendix C. This program – as the brute-force program for $n = 4$ – runs on iterations, such that several iterations can be run simultaneously, effectively speeding up the calculation time. However, since each iteration has its own set of representatives, this meant that some of the iterations found and analysed the same representatives – a problem that would be fixed were the methods implemented using parallel processing (see the introduction of Chapter 3 for a short discussion of why this was not implemented). For the purposes of this thesis, this only posed a minor problem. Any two representatives found were equal, and all duplicates in the data set were removed. The additional time spent was, in any such case, much less than the execution time of only running *one* process. (See Section 4.4.4 for a discussion on execution time of this program.) For the reader interested in how these iterations were further calculated, please view the main-section of the program listed in Appendix C.

```python
def calculate_thickness(functions, matrices, vectors, file_write):
    for f in functions:
        min_f = f
        rigid_function = True # Assume rigid, attempt to disprove

        iter_mat = 0
        while (iter_mat < n_matrices):
            iter_vec = 0
            while (iter_vec < n_vectors):
                # Apply map to function f
                f_map = matrices[iter_mat] + vectors[iter_vec]

                mapped_f = f(f_map[0], f_map[1], f_map[2],
                             f_map[3], f_map[4])
                mapped_monomials = len(mapped_f.set())

                # If function not rigid, skip to next function
                if (mapped_monomials < n_monoms):
                    rigid_function = False
                    iter_mat = n_matrices
                    iter_vec = n_vectors

                # If representative already found,
                # skip to next function
                elif (mapped_f in representatives):
                    rigid_function = False
                    iter_mat = n_matrices
                    iter_vec = n_vectors

                # If function may be rigid,
                # check if better representative
                elif (mapped_monomials == n_monoms):
                    if (mapped_f < min_f):
                        min_f = mapped_f

                iter_vec += 1
            iter_mat += 1

        if (rigid_function):
            # Map to lowest indexed version
            min_f_rep = map_to_smallest_index(min_f)

            # Save results for each viable function to file
            text_file = open(file_write, 'a')
            text_file.write("%s|\n" % min_f_rep)
            text_file.close()

            # Add found representative to set
            representatives.add(min_f_rep)
    return
```

Figure 4.3: Code: Method for calculating thickness in $n = 5$

Mon. count	Functions/Iterations	Min. time	Max. time	Total time (add.)
2	28 / 3	4h	4h	12h
3	134 / 4	6h	12h	1d 12h
4	625 / 4	1d 3h	1d 7h	4d 21h
5	2674 / 8	4d 10h	5d 5h	38d 14h
6	10 195 / 14	1d 14h	3d 19h	39d 17h
7	34 230 / 15	1d 4h	3d 15h	36d 16h
8	100 577 / 20	24s	5d 1h	11d 20h
Total			19d 15h	131d 16h

Table 4.5: Execution time of the iterations completed by Chapter 4's program

4.4.4 Execution time

As noted in Chapter 3, if the brute-force program listed there was run for $n = 5$ variables, it would take approximately 2 450 000 years – or 24 500 years, if run on 100 simultaneous iterations. As previously explained, this rough calculation was estimated by averaging the time elapsed for each function to 5 hours, and multiplying with $|\mathcal{B}_5| = 2^{2^5}$. Although this estimate is not at all precise, it successfully shows the magnitude of the problem, and may serve as a reference point for comparison.

Listed in Table 4.5 is an overview of the time spent on each monomial count value, by the designed program. It shows the number of functions together with the number of iterations the functions were divided over, and the minimum & maximum time spent on any one iteration within that monomial count value. The rightmost column is the additive time spent, i.e. addition of all the time values of each iteration – showing an estimate of how long the program would take when run as one iteration. In practice, because the iterations of any one monomial count value were run simultaneously, the actual execution time of a specific monomial count value was the elapsed time of the longest iteration. The time spent in total of all monomial count values is shown in the bottom row, under the column for maximum time. Note that only the time for executing the method for thickness calculation (given in Figure 4.3) is included, all other methods are excluded (i.e., collection of functions and affine transformation generation).

The shortest iteration took *only 24 seconds*, and most of the iterations for eight monomials were executed quickly (14 of the iterations finished in less than 30 minutes). Even though there were 5028 functions to process for each iteration running on monomial count value 8, all functions in some of these iterations could be shown very quickly not to be rigid. However, in the iterations that did indeed find rigid functions for this monomial count, the time spent was much longer, i.e. at most 5 days 1 hour.

Even though the complete run time of the program was 19 days 15 hours, it stands as a great reduction compared to the original estimate by brute-force.

The results of the stated program are the 382 representatives of their orbit over affine transformations in $\mathcal{B}_5$. The full distribution and properties of their orbits will be discussed in the next chapter, in Section 5.2.

Chapter 5

Analysis and Assessment

$\mathcal{T}$	$n=0$	$n=1$	$n=2$	$n=3$	$n=4$	$n=5$
0	1	1	1	1	1	1
1	1	2	3	4	5	6
2	-	-	1	4	10	19
3	-	-	-	1	10	46
4	-	-	-	-	5	81
5	-	-	-	-	1	111
6	-	-	-	-	-	81
7	-	-	-	-	-	33
8	-	-	-	-	-	4
Sum	2	3	5	10	32	382
max($\mathcal{T}_n$)	1	1	2	3	5	8

Table 5.1: Distribution of representatives within each thickness value

The full distribution of algebraic thickness of the representative functions in $n \leq 5$ variables is given in Table 5.1, summarizing the results of the data collection conducted by use of the program explained in Chapter 4. The distribution of number of functions within each thickness value is further detailed and described in Sections 5.1 and 5.2.

The Sum-row of Table 5.1 shows the total amount of representatives, which is equal to the number of equivalence classes under affine transformation in n variables [16, 20]. As discussed in Section 4.2, this is true for any n.

The maximum thickness for each $n \leq 5$ (bottom row of Table 5.1) follows the Fibonacci-sequence (see (2.3) of Section 2.1.2), that is

$$max(\mathcal{T}_n) = F(n+1), \text{ for } n \leq 5. \tag{5.1}$$

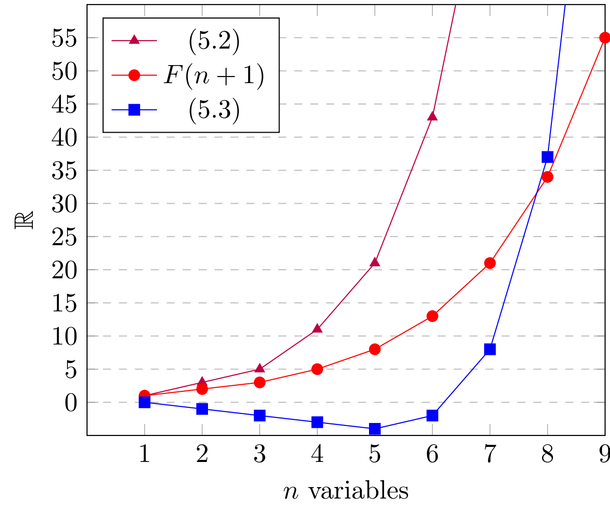


Figure 5.1: Plot of upper- and lower bound of maximum $\mathcal{T}$ versus Fibonacci

As a reminder, in [8, 4] Carlet proved that the upper- and lower bounds of $max_{f \in \mathcal{B}_n}(\mathcal{T}(f))$ are:

$$\text{Upper bound: } \frac{2}{3}2^n, \tag{5.2}$$

$$\text{Lower bound: } 2^{n-1} - n2^{\frac{n-1}{2}}, \tag{5.3}$$

that is, for any given $n > 0$: (5.2) the algebraic thickness of any Boolean function is $\frac{2}{3}2^n$ or lower, and (5.3) there exists[1] a Boolean function f with $\mathcal{T}_n(f)$ greater than $2^{n-1} - n2^{\frac{n-1}{2}}$. (See Section 2.3 for further reference.)

Although the improvement of the bounds stated here would be of great interest, following the apparent Fibonacci-pattern would most likely not yield any results, as it can be shown that the Fibonacci-sequence for $(n + 1)$ is only greater than the lower bound of $max_{f \in \mathcal{B}_n}(\mathcal{T}(f))$ for $n < 8$. For any $n \geq 8$, the lower bound exceeds $F(n + 1)$, and thus there must exist a Boolean function f with $\mathcal{T}_n(f) > F(n+1)$; simultaneously showing that $max_{f \in \mathcal{B}_n}(\mathcal{T}(f))$ cannot be equal to $F(n + 1)$ for all n, and that (5.3) is the better lower bound.

Calculation of these values is a trivial task, and Figure 5.1 is given here to illustrate how the Fibonacci-sequence is contained within the bounds for $n < 8$, but is then exceeded by the lower bound at $n = 8$, a trend which continues for higher n. (That is, (5.3) grows faster than $F(n + 1)$.)

Returning back to Table 5.1, the columns for variables $n = 0, 1, 2$ were first manually calculated, before the columns for $n \leq 4$ variables were calculated by the brute-force program from Chapter 3; and finally, using the program given in Appendix C, discussed in Chapter 4, *all* representatives for $n \leq 5$ were calculated. The manual calculation was done to confirm that the brute-force program gave *correct* results, and all variables were input to the program from Chapter 4 to confirm both programs gave the *same* results for $n < 5$ – which they did, indeed.

[1] In fact, according to Carlet in [4], *almost all* Boolean functions have $\mathcal{T}_n$ greater than this.

$\mathcal{N}$	$n=2$	$n=3$	$n=4$	$n=5$
0	8	16	32	64
1	8	128	512	2048
2	-	112	3840	31 744
3	-	-	17 920	317 440
4	-	-	28 000	2 301 440
5	-	-	14 336	12 888 064
6	-	-	896	57 996 288
7	-	-	-	215 414 784
8	-	-	-	647 666 880
9	-	-	-	1 362 452 480
10	-	-	-	1 412 100 096
11	-	-	-	556 408 832
12	-	-	-	27 387 136
max($\mathcal{N}$)	1	2	6	12

Table 5.2: Distribution of number of $f \in \mathcal{B}_n$ with given $\mathcal{N}$-value, $n \leq 5$

A full overview of all representatives in $n \leq 4$ variables is given in Section 5.1, followed by an analysis of what properties are present for any functions within each algebraic thickness value (e.g. how many of the functions in $n = 4$ variables with $\mathcal{T}_4 = 3$ are bent). This analysis is based on data collected by iteration through each of the orbits of every representative, counting the various properties by use of various programs based on the methods given in Chapters 3 and 4. This data set for $n = 4$ is given in full in Appendix E.

For $n = 5$, the same property analysis is given in Section 5.2. As there are 382 representatives – spanning 10 pages – they are only listed in full in Appendix F, together with the full data set of the orbit analysis.

The full distribution of how many functions in n variables have which value of nonlinearity ($\mathcal{N}$) is given in Table 5.2. This table has been calculated through an exhaustive search using SageMath, by visiting every function in $\mathcal{B}_n$, for $n \leq 5$, confirming the results listed in [26]. Columns for $n = 2, 3$ are not strictly relevant to the following property analysis, but are included for completeness. (Note: the functions in $n = 0, 1$ are all linear, i.e. have $\mathcal{N} = 0$.)

Furthermore, the distribution of the number of orbits within each possible $\mathcal{N}$-value (i.e., the distribution of nonlinearity of the representatives) is shown in Table 5.3 – recall that nonlinearity is an affine invariant, described in Section 2.2. E.g., from the table, there are 16 representatives (and therefore orbits) in $n = 5$ variables where $\mathcal{N} = 5$. Further, we can see that there are two orbits with maximum nonlinearity in $n = 4$ (and therefore two orbits that contain all bent functions in $n = 4$), and 14 orbits with maximum nonlinearity in $n = 5$ ($\mathcal{N} = 6$ and $\mathcal{N} = 12$, respectively). Refer to the mentioned appendix F for more specific details regarding which representatives that have which $\mathcal{N}$-value.

As noted in Section 3.1, the properties of Boolean functions in this chapter have all been calculated using SageMath's packages related to Boolean func-

$\mathcal{N}$	$n=2$	$n=3$	$n=4$	$n=5$
0	3	3	3	3
1	2	4	4	4
2	-	3	5	5
3	-	-	6	6
4	-	-	8	12
5	-	-	4	16
6	-	-	2	31
7	-	-	-	46
8	-	-	-	68
9	-	-	-	72
10	-	-	-	73
11	-	-	-	32
12	-	-	-	14

Table 5.3: Distribution of number of orbits with given $\mathcal{N}$-value, $n \leq 5$

tions, i.e. the *pbori*-package for Boolean polynomials, and the *boolean_function*-package (mainly used for its *nonlinearity()*-method).

5.1 Results and analysis for $n = 4$

After the completion of the execution of the program explained in Section 3.2, a full algebraic thickness distribution of each function in $\mathcal{B}_2$, $\mathcal{B}_3$, and $\mathcal{B}_4$ was stored in text files, where each file consisted of 14, 254, 65 534 lines each, respectively (constant functions 0 and 1 were omitted). The full data set for $n = 2$ is given (and explained) in Appendix A. The full data sets for $\mathcal{B}_3$ and $\mathcal{B}_4$ are too large to be shown in this thesis – however, an attempt to summarize the interesting details of the orbits of $n = 4, 5$ are given in the following sections (Sections 5.1.1 and 5.2.1). The analysis of the text file for $n = 4$ was the inception of the definitions of rigid- and representative functions, as explained in the introduction of Chapter 4.

The representatives for each unique orbit of Boolean functions are, by Definition 4.2, the minimal function of its orbit, both in number of monomials and degree of each monomial. As shown in Examples 4.1 and 4.2, in most cases there are more than one such function (e.g. x_1x_2 and x_2x_3), which is why Part III of Definition 4.2 specifies that *the function with the lowest indices in its variables* is chosen (so, x_1x_2 represents its orbit, here).

However, since it is the structure of the representatives that should be of interest, not the trivial implementation specifics (for instance, indices may start with 0, not 1) this last remark of the definition can be avoided by replacing the variables $\{x_1, x_2, x_3, \dots\}$ with capital letters {A, B, C, ...} (by choosing a bijective map between the sets, i.e. never mapping the same variable to more than one capital letter), and sorting by alphabetical order within the monomials. Thus, the smallest rigid functions from Example 4.1, Section 4.2.1,

can be mapped:

$$
\begin{aligned}
x_1x_2x_3 + x_3 + 1 &\equiv \text{ BCA + A + 1 = ABC + A + 1},\\
x_1x_2x_3 + x_2 + 1 &\equiv \text{ BAC + A + 1 = ABC + A + 1},\\
x_1x_2x_3 + x_1 + 1 &\equiv \text{ ABC + A + 1},
\end{aligned}
$$

by corresponding bijective maps, and are all represented by the same representative function. In this thesis, the term "ABC-form" will denote the capital letter representation of the representatives, while $x_1x_2x_3$-form will denote the classical form, i.e., using x-variables.

The ABC-form may serve to better show the emerging patterns in the structure of the representatives (this is further discussed in Section 6.1.4). In Table 5.4 the representatives in $n = 0, 1, 2, 3, 4$ are all listed in this form. Additionally, in the interest of avoiding letting the representation of the representative functions distract from what is interesting, the $x_1x_2x_3$-form of Table 5.4 have been included in appendix D. To reiterate, this presentation is purely cosmetic.

$\mathcal{T}$	$n = 0$	$n = 1$	$n = 2$	$n = 3$	$n = 4$
0	0	0	0	0	0
1	1	1 A	1 A AB	1 A AB ABC	1 A AB ABC ABCD
2			AB + 1	AB + 1 AB + C ABC + 1 ABC + A	AB + 1 AB + C AB + CD ABC + 1 ABC + A ABC + D ABC + AD ABCD + 1 ABCD + A ABCD + AB
3				 ABC + A + 1	AB + CD + 1 ABC + A + 1 ABC + AD + 1 ABC + AD + B ABC + AB + CD ABCD + A + 1 ABCD + AB + 1 ABCD + AB + A ABCD + AB + C ABCD + AB + CD
4					ABC + AB + CD + 1 ABCD + AB + A + 1 ABCD + AB + C + 1 ABCD + AB + CD + 1 ABCD + AB + CD + A
5					ABCD + AB + CD + A + 1

Table 5.4: Representatives in $n \leq 4$: sorted by thickness, clustered by degree

Properties		Nonlinearity		Degrees	
Number of functions	307	0	31	0	1
Homogeneous functions	52	1	16	1	30
Rigid functions	16	2	120	2	140
Balanced functions	30	3	0	3	120
Bent functions	0	4	140	4	16
Orbits	5	5	0		
Bent orbits	0	6	0		
Balanced orbits	1				

Table 5.5: Property distribution of functions in $\mathcal{B}_4$ with $\mathcal{T}_4 = 1$

5.1.1 Property distribution in $n = 4$, sorted by thickness

Given in Tables 5.5–5.9 is a distribution of the properties listed below, for all functions in $n = 4$ variables with the stated thickness. The property distributions may serve as a summary of what properties the functions with the listed algebraic thickness have. Since $\mathcal{B}_2 \subset \mathcal{B}_3 \subset \mathcal{B}_4$ – and that the focus of this thesis is $n = 4, 5$ variables – the property distributions for functions in $n < 4$ variables are excluded.

The properties collected (refer to Section 2.2 for definitions) in this analysis are: the complete number of functions with the given algebraic thickness, the number of homogeneous-, rigid-, balanced-, and bent functions; the number of orbits (cf. representatives), bent orbits and balanced orbits (as these are affine invariants); and an overview of the nonlinearity-, and the algebraic degree distribution of the functions. Table 5.10 is a summary of all the property distributions of Tables 5.5–5.9, i.e. a complete property distribution of all functions in $\mathcal{B}_4$, not dependent on algebraic thickness.

In both Sections 5.1.1 and 5.2.1, the property distribution of thickness zero has been omitted, as there is only one function in $\mathcal{T}_n = 0$ (for any n), namely $f(\mathbf{x}) = 0$. Trivially, but stated here for completeness, this means that the property distribution of $\mathcal{T}_n = 0$ will contain 1 homogeneous function, 0 (semi-) bent functions, 0 balanced functions; the nonlinearity of $f(\mathbf{x}) = 0$ is 0, and the algebraic degree is 0. (*Note: in some programming languages, for instance SageMath, the algebraic degree of* $f(\mathbf{x}) = 0$ *is defined as* (-1).) There is 1 rigid function, and therefore 1 representative function: $f(\mathbf{x}) = 0$ itself. This property distribution is equal for any n, since no non-zero function can be mapped to 0 through an invertible affine transformation.

Remark. In the interest of avoiding confusion, please remark that this property analysis is not only of the *representatives*, but *all* functions in $\mathcal{B}_n$. For instance, let $f \in \mathcal{B}_4$ with ANF:

$$f = x_1x_2 + x_1x_3 + x_1x_4 + x_2x_3 + x_2x_4 + x_3x_4.$$

The algebraic thickness of f is 3, and therefore it belongs to Table 5.7, the overview of functions with $\mathcal{T}_4 = 3$. *(Also note that this is the only homogeneous function with* $\mathcal{T}_4 > 2$.*)*

Properties		Nonlinearity		Degrees	
Number of functions	6804	0	0	0	0
Homogeneous functions	42	1	256	1	0
Rigid functions	64	2	2880	2	1428
Balanced functions	2760	3	560	3	4560
Bent functions	448	4	2660	4	816
Orbits	10	5	0		
Bent orbits	1	6	448		
Balanced orbits	2				

Table 5.6: Property distribution of functions in $\mathcal{B}_4$ with $\mathcal{T}_4 = 2$

Properties		Nonlinearity		Degrees	
Number of functions	33 448	0	0	0	0
Homogeneous functions	1	1	240	1	0
Rigid functions	188	2	840	2	448
Balanced functions	10 080	3	8960	3	19 320
Bent functions	448	4	18 480	4	13 680
Orbits	10	5	4480		
Bent orbits	1	6	448		
Balanced orbits	1				

Table 5.7: Property distribution of functions in $\mathcal{B}_4$ with $\mathcal{T}_4 = 3$

Properties		Nonlinearity		Degrees	
Number of functions	22 288	0	0	0	0
Homogeneous functions	0	1	0	1	0
Rigid functions	271	2	0	2	0
Balanced functions	0	3	8400	3	6720
Bent functions	0	4	6720	4	15 568
Orbits	5	5	7168		
Bent orbits	0	6	0		
Balanced orbits	0				

Table 5.8: Property distribution of functions in $\mathcal{B}_4$ with $\mathcal{T}_4 = 4$

Properties		Nonlinearity		Degrees	
Number of functions	2688	0	0	0	0
Homogeneous functions	0	1	0	1	0
Rigid functions	48	2	0	2	0
Balanced functions	0	3	0	3	0
Bent functions	0	4	0	4	2688
Orbits	1	5	2688		
Bent orbits	0	6	0		
Balanced orbits	0				

Table 5.9: Property distribution of functions in $\mathcal{B}_4$ with $\mathcal{T}_4 = 5$

Properties	Total amount
Number of functions	65536
Homogeneous functions	96
Rigid functions	588
Balanced functions	12 870
Bent functions	896
Orbits	32
Bent orbits	2
Balanced orbits	4

Table 5.10: Summary of the property distribution of $n = 4$

5.1.2 Bent functions in $n = 4$ (and $n = 2$)

The 896 bent functions (as defined in Section 2.2, refer to Definition 2.8) in $n = 4$ are divided equally between thickness 2 and 3 (see Tables 5.6 and 5.7). This is because there are only two bent orbits of $n = 4$, represented by (excerpt from Appendix E):

Representative	H	Rigids	Orbit length
AB + CD	27	3	448
AB + CD + 1	1	3	448

where 'H' is the number of homogeneous functions, and 'Rigids' is the number of rigid functions, in their orbits. None of the bent orbits in $\mathcal{B}_4$ are balanced. Both representatives also have the same orbit lengths, perhaps because the two functions are the complement of each other (i.e., the complement of f is $\bar{f} = f + 1$, see Section 6.1.3 for further notes on orbit lengths of such "pairs"). The nonlinearity of the bent functions in $n = 4$ is 6, equal to the right-hand side of the equation given in (2.8) (maximum bound of $\mathcal{N}$) – that is $6 = 2^{4-1} - 2^{\frac{4}{2}-1}$.

For comparison, there are eight bent Boolean functions in $n = 2$, also divided over two orbits, whose representative is shown here:

Function	Representative
x_1x_2	AB
$x_1x_2 + 1$	AB + 1
$x_1x_2 + x_1$	AB
$x_1x_2 + x_2$	AB
$x_1x_2 + x_1 + 1$	AB + 1
$x_1x_2 + x_2 + 1$	AB + 1
$x_1x_2 + x_1 + x_2$	AB + 1
$x_1x_2 + x_1 + x_2 + 1$	AB

The nonlinearity of any bent function in $n = 2$ is 1, all (eight) other functions in $\mathcal{B}_2$ have $\mathcal{N} = 0$.

As there are no bent functions in odd n, a corresponding analysis for semi-bent functions is given in Section 5.2.2.

5.1.3 Balanced functions in $n = 4$ (and $n = 2$)

Since the truth table of a Boolean function in n variables has 2^n entries, and a balanced function will have exactly 2^{n-1} ones, the number of balanced Boolean functions equals the number of ways of inserting 2^{n-1} ones in the possible 2^n positions. Hence the number of balanced Boolean functions is exactly $\binom{2^n}{2^{n-1}}$, as observed also for $n = 4, 5$. Thus, there are 12 870 balanced functions in $n = 4$ variables, and these functions belong to four different balanced orbits:

Representative	$\mathcal{N}$	H	Rigids	Orbit length
A	0	15	4	30
AB + C	4	10	12	840
ABC + D	2	0	4	1920
ABC + AD + B	4	0	24	10080

The maximum algebraic thickness of a balanced function in $\mathcal{B}_4$ is, then, $\mathcal{T}_4 = 3$.

Again, for comparison, in $\mathcal{B}_2$ there are 6 balanced functions, in only one balanced orbit: the orbit represented by A (or x_1):

Function	Representative
x_1	A
x_2	A
$x_1 + x_2$	A
$x_1 + 1$	A
$x_2 + 1$	A
$x_1 + x_2 + 1$	A

Their nonlinearity is 0, as implied in Section 5.1.2.

Properties		Nonlinearity		Degrees	
Number of f	2451	0	63	0	1
Homogeneous f	203	1	32	1	62
Rigid f	32	2	496	2	620
Balanced f	62	3	0	3	1240
Semi-Bent f	0	4	1240	4	496
Orbits	6	5	0	5	32
Semi-Bent orbits	0	6	0		
Balanced orbits	1	7	0		
		8	620		
		9	0		
		10	0		
		11	0		
		12	0		

Table 5.11: Property distribution of functions in $\mathcal{B}_5$ with $\mathcal{T}_5 = 1$

5.2 Results and analysis for $n = 5$

When the program explained in Chapter 4 was completed, the results where the 382 representative functions in $n = 5$ variables, as desired. Because of the number of pages needed to contain *all* Boolean functions in $n = 5$ (the number of pages would be in the realm of 100 million), the choice was made not to list the full distribution of all functions, and rather focus on the representatives and the properties of their respective orbits, effectively summarizing the full distribution. All 382 representatives are listed in Appendix F, in their ABC-form.

As with the representatives in $n = 4$, all orbits of the representatives in $n = 5$ were iterated through, counting various properties, using the same tools as in the previous section. Along with the representatives in Appendix F is also the data set resulting from said analysis, containing the number of rigid functions in each orbit, the length of each orbit, etc. This is also summarized in Section 5.2.1, for each value of algebraic thickness.

5.2.1 Property distribution in $n = 5$, sorted by thickness

The following section contains the property distribution analysis for $n = 5$, using the same tools and definitions as for $n = 4$ in Section 5.1.1 – the only difference in properties being that, as no function is bent in odd dimensions, the definition for semi-bent functions is used instead.

Tables 5.11–5.18 are listed on the next pages, containing a full overview of the properties of all functions in the orbits with the given algebraic thickness value. The complete data set is given in Appendix F, which was calculated by iteration through all affine transformations of all representative functions.

Properties		Nonlinearity		Degrees	
Number of f	695 796	0	0	0	0
Homogeneous f	987	1	1024	1	0
Rigid f	336	2	23 808	2	23 188
Balanced f	84 072	3	4960	3	466 736
Semi-Bent f	13 888	4	104 160	4	194 928
Orbits	19	5	0	5	10 944
Semi-Bent orbits	1	6	45 136		
Balanced orbits	3	7	4960		
		8	180 420		
		9	0		
		10	317 440		
		11	0		
		12	13 888		

Table 5.12: Property distribution of functions in $\mathcal{B}_5$ with $\mathcal{T}_5 = 2$

Properties		Nonlinearity		Degrees	
Number of f	31 424 328	0	0	0	0
Homogeneous f	859	1	992	1	0
Rigid f	2480	2	7440	2	41 664
Balanced f	4 228 896	3	158 720	3	7620792
Semi-Bent f	874 944	4	1 536 360	4	22 119 120
Orbits	46	5	34 720	5	1 642 752
Semi-Bent orbits	3	6	2 138 752		
Balanced orbits	6	7	853 120		
		8	15 323 920		
		9	317 440		
		10	9 900 160		
		11	277 760		
		12	874 944		

Table 5.13: Property distribution of functions in $\mathcal{B}_5$ with $\mathcal{T}_5 = 3$

Properties		Nonlinearity		Degrees	
Number of f	240 101 200	0	0	0	0
Homogeneous f	61	1	0	1	0
Rigid f	11 520	2	0	2	0
Balanced f	15 582 336	3	153 760	3	23 290 176
Semi-Bent f	2 499 840	4	659 680	4	168 597 840
Orbits	81	5	1 416 576	5	48 213 184
Semi-Bent orbits	2	6	10 731 952		
Balanced orbits	6	7	17 541 536		
		8	112 334 080		
		9	18 213 120		
		10	63 162 624		
		11	10 888 192		
		12	4 999 680		

Table 5.14: Property distribution of functions in $\mathcal{B}_5$ with $\mathcal{T}_5 = 4$

Properties		Nonlinearity		Degrees	
Number of f	1 086 598 112	0	0	0	0
Homogeneous f	0	1	0	1	0
Rigid f	47 220	2	0	2	0
Balanced f	187 210 240	3	0	3	27 664 896
Semi-Bent f	2 666 496	4	0	4	763 701 120
Orbits	111	5	7 936 992	5	295 232 096
Semi-Bent orbits	1	6	42 413 952		
Balanced orbits	11	7	53 524 352		
		8	364 837 760		
		9	193 162 240		
		10	375 614 848		
		11	40 608 512		
		12	8 499 456		

Table 5.15: Property distribution of functions in $\mathcal{B}_5$ with $\mathcal{T}_5 = 5$

Properties		Nonlinearity		Degrees	
Number of f	1 842 215 424	0	0	0	0
Homogeneous f	0	1	0	1	0
Rigid f	59 760	2	0	2	0
Balanced f	308 646 912	3	0	3	7 999 488
Semi-Bent f	7 999 488	4	0	4	951 105 792
Orbits	81	5	3 499 776	5	883 110 144
Semi-Bent orbits	2	6	2 666 496		
Balanced orbits	9	7	96 827 136		
		8	154 990 080		
		9	694 122 240		
		10	788 449 536		
		11	88 660 992		
		12	12 999 168		

Table 5.16: Property distribution of functions in $\mathcal{B}_5$ with $\mathcal{T}_5 = 6$

Properties		Nonlinearity		Degrees	
Number of f	935 273 472	0	0	0	0
Homogeneous f	0	1	0	1	0
Rigid f	64 470	2	0	2	0
Balanced f	85 327 872	3	0	3	0
Semi-Bent f	0	4	0	4	174 655 488
Orbits	33	5	0	5	760 617 984
Semi-Bent orbits	0	6	0		
Balanced orbits	2	7	46 663 680		
		8	0		
		9	436 638 720		
		10	174 655 488		
		11	277 315 584		
		12	0		

Table 5.17: Property distribution of functions in $\mathcal{B}_5$ with $\mathcal{T}_5 = 7$

Properties		Nonlinearity		Degrees	
Number of f	158 656 512	0	0	0	0
Homogeneous f	0	1	0	1	0
Rigid f	25 440	2	0	2	0
Balanced f	0	3	0	3	0
Semi-Bent f	0	4	0	4	0
Orbits	4	5	0	5	158 656 512
Semi-Bent orbits	0	6	0		
Balanced orbits	0	7	0		
		8	0		
		9	19 998 720		
		10	0		
		11	138 657 792		
		12	0		

Table 5.18: Property distribution of functions in $\mathcal{B}_5$ with $\mathcal{T}_5 = 8$

Properties	Total amount
Number of functions	4 294 967 296
Homogeneous functions	2111
Rigid functions	211 259
Balanced functions	601 080 390
Semi-Bent functions	14 054 656
Number of orbits	382
Semi-Bent orbits	9
Balanced orbits	38

Table 5.19: Summary of the property distribution of $n = 5$

$\mathcal{T}$	Representative	Balanced	Orbit length
2	AB + CD	-	13 888
3	AB + CD + 1	-	13 888
3	AB + CD + E	Y	27 776
3	ABC + AD + BE	-	833 280
4	ABC + AD + BE + 1	-	833 280
4	ABC + AD + BE + C	Y	1 666 560
5	ABC + ABD + ACE + BC + DE	-	2 666 496
6	ABC + ABD + ACE + BC + DE + 1	-	2 666 496
6	ABC + ABD + ACE + BC + DE + A	Y	5 332 992
			14 054 656

Table 5.20: All semi-bent representatives in $n = 5$.

5.2.2 Semi-Bent functions in $n = 5$ (and $n = 3$)

In total, there are 9 semi-bent orbits in $n = 5$, as defined in Definition 2.9. The representatives of these orbits are listed in Table 5.20, with an overview of which of these orbits are balanced, and their respective orbit lengths. In total there are 14 054 656 semi-bent functions in $n = 5$ variables (as seen in Table 5.19), i.e. the sum of the orbit lengths of the semi-bent orbits.

As the orbits shown in Table 5.20 are semi-bent, they all have the maximum nonlinearity in $n = 5$, i.e. $\mathcal{N} = 12$. There are five other orbits with the same nonlinearity that are *not* semi-bent (excerpt from Appendix F):

$\mathcal{T}$	Representative	D	$\mathcal{N}$	H	Ba	Rigids	Orbit length
4	ABCD + ABE + AC + BD	4	12	0	-	60	2 499 840
5	ABCD + ABE + AC + BD + 1	4	12	0	-	60	2 499 840
5	ABCD + ABE + AC + BD + E	4	12	0	-	60	3 333 120
6	ABCD + ABE + AC + BD + E + 1	4	12	0	-	600	3 333 120
6	ABCD + ABE + AC + BD + C + D	4	12	0	Y	180	1 666 560

where 'D' is algebraic degree, 'H' is homogeneous functions, 'Ba' designates a balanced orbit, 'Rigids' is the number of rigid functions, and the rest should be self-explanatory.

Common in all of the orbits listed here is their algebraic degree being equal to 4. As stated in Section 2.2, from [14], it is known that the maximum algebraic degree of a semi-bent function is $\frac{n+1}{2}$, for an odd n. The results shown above and below, and in Table 5.20, are in accordance with this.

For comparison, the number of semi-bent functions in $n = 3$ variables is 112, and there are three semi-bent orbits, where one orbit is balanced:

$\mathcal{T}$	Representative	D	$\mathcal{N}$	H	Ba	Rigids	Orbit length
1	AB	2	2	6	-	3	28
2	AB + 1	2	2	0	-	3	28
2	AB + C	2	2	1	Y	3	56

5.2.3 Balanced functions in $n = 5$ (and $n = 3$)

As seen in the summary of the property distribution for $n = 5$ (Table 5.19), there are 38 balanced orbits in $\mathcal{B}_5$. These balanced orbits are represented by the representative functions listed in Table 5.21, which is an excerpt from Appendix F. *(Note that this table is quite large, and is therefore given on the next page.)*

As before, 'D' is algebraic degree, $\mathcal{N}$ denotes nonlinearity, 'H' is the number of homogeneous functions, 'Be' is the number of semi-bent orbit, 'R' is the number of rigid functions, and 'O' is the orbit length.

Indicated in Section 5.2.2, there are three balanced orbits that are semi-bent as well. All 601 080 390 balanced functions in $\mathcal{B}_5$ are represented by one of the orbits listed here. The maximum thickness for balanced functions in $n = 5$ variables is $\mathcal{T}_5 = 7$ (where there are two balanced orbits, listed at the bottom of Table 5.21).

In $n = 3$ variables, there are 70 balanced functions over two balanced orbits:

$\mathcal{T}$	Representative	D	$\mathcal{N}$	H	Be	R	O
1	A	1	0	7	-	3	14
2	AB + C	2	2	1	Y	3	56

$\mathcal{T}$	Representative	D	$\mathcal{N}$	H	Be	R	O
1	A	1	0	31	-	5	62
2	AB + C	2	8	65	-	30	8680
2	ABC + D	3	4	0	-	20	59520
2	ABCD + E	4	2	0	-	5	15872
3	AB + CD + E	2	12	252	Y	15	27776
3	ABC + AD + E	3	8	0	-	60	833280
3	ABC + AD + B	3	8	0	-	120	312480
3	ABCD + AE + B	4	8	0	-	60	1666560
3	ABCD + AB + E	4	6	0	-	30	555520
3	ABCD + ABE + C	4	4	0	-	60	833280
4	ABC + AD + BE + C	3	12	0	Y	60	1666560
4	ABC + CDE + AB + C	3	8	0	-	30	555520
4	ABCD + AB + CE + D	4	10	0	-	60	6666240
4	ABCD + AB + CD + E	4	10	0	-	15	444416
4	ABCD + ABE + AC + D	4	8	0	-	120	4999680
4	ABCD + ABE + AC + B	4	8	0	-	120	1249920
5	ABC + CDE + AB + A + D	3	8	0	-	1380	9999360
5	ABCD + ABE + CD + C + E	4	10	0	-	240	13332480
5	ABCD + ABE + CDE + A + C	4	8	0	-	180	39997440
5	ABCD + ABE + ACE + BD + C	4	8	0	-	120	19998720
5	ABCD + ABE + CDE + AB + E	4	6	0	-	30	8888320
5	ABCD + ABC + ADE + B + E	4	6	0	-	1680	9999360
5	ABCD + ABC + ADE + BE + D	4	8	0	-	120	9999360
5	ABCD + ABC + ADE + BD + E	4	10	0	-	120	9999360
5	ABCD + ABC + CDE + AB + C	4	8	0	-	330	1666560
5	ABCD + ABC + ABE + DE + C	4	8	0	-	980	53329920
5	ABCD + ABC + ABE + DE + A	4	8	0	-	1380	9999360
6	ABC + ABD + ACE + BC + DE + A	3	12	0	Y	300	5332992
6	ABCD + ABE + CDE + A + C + 1	4	8	0	-	1380	39997440
6	ABCD + ABE + AC + BD + C + D	4	12	0	-	180	1666560
6	ABCD + ABE + CDE + AB + AC + E	4	10	0	-	360	79994880
6	ABCD + ABE + ACE + AB + DE + C	4	10	0	-	120	39997440
6	ABCD + ABC + ADE + BD + C + E	4	10	0	-	1260	19998720
6	ABCD + ABC + ADE + BD + CE + A	4	10	0	-	420	39997440
6	ABCD + ABC + CDE + AB + C + 1	4	8	0	-	420	1666560
6	ABCD + ABC + ABE + AC + DE + B	4	10	0	-	1740	79994880
7	ABCD + ABE + CDE + AC + BD + A + B	4	10	0	-	720	31997952
7	ABCD + ABE + CDE + AB + AC + BD + E	4	10	0	-	900	53329920

Table 5.21: All balanced orbits in $n = 5$.

5.2.4 Functions with maximum thickness in $n = 5$

As detailed previously (in Section 2.2, with mentions elsewhere in this thesis), the complexity criteria for "good" cryptographic Boolean functions (for use in cryptographic systems) are often attributed to the algebraic degree and nonlinearity of the functions [8], and whether or not they are balanced [10]. (Other criteria not mentioned here also exist, cf. *nonnormality*.) The functions with the highest nonlinearity (in even n) are the bent functions, which is why the distribution of such was given and discussed in Section 5.1.2. For odd n, we instead focus on semi-bent functions, as this class of functions, in general, has high nonlinearity (See Definition 2.9, refer to [23]). While there are only one orbit with maximum algebraic thickness in all $n < 5$ variables, in $n = 5$ there are four:

Representative	$\mathcal{N}$	Rigids	Orbit length
ABCDE + ABC + ADE + AB + DE + A + D + 1	9	12960	19998720
ABCDE + ABC + ABD + ACE + BC + DE + B + 1	11	4260	53329920
ABCDE + ABC + ABD + ACE + BC + DE + A + 1	11	2160	31997952
ABCDE + ABC + ABD + ACE + BC + DE + A + B	11	6060	53329920

All of the four orbits with $\mathcal{T}_5 = 8$ have maximum algebraic degree. However, none of them are balanced, or semi-bent – indeed, none of them have the maximum nonlinearity. (As a note, we also include that none of them contain any homogeneous functions.)

From all orbits with maximum thickness in $2 < n \leq 5$ variables, we observe that this pattern seems to be a common factor for representatives with maximum thickness. In $n = 3$, the representative ABC + A + 1 has maximum algebraic degree, is not balanced, is not semi-bent, and has nonlinearity 1 (maximum is 2, refer to Table 5.3). In $n = 4$, the representative ABCD + AB + CD + A + 1 also has maximum algebraic degree, is not balanced, is not bent, and has nonlinearity 5 (maximum is 6; for a full property analysis of this representative, refer to Table 5.9, or Appendix E).

This could have implications for functions of higher numbers of variables, however, further study is needed.

Remark. Three of the here listed representatives in $n = 5$ contain a constant. What happens to the only representative without a constant when a constant is added to it, is further discussed in Section 6.1.3.

$\mathcal{T}$	Balanced	$\mathcal{N}$	Rigids	Representative
5	-	10	180	ABCD + ABE + AC + DE + C
6	-	10	420	ABCD + ABE + AC + DE + C + 1
6	-	10	360	ABCD + ABE + CDE + AC + AE + B
6	Y	10	360	ABCD + ABE + CDE + AB + AC + E
6	-	10	1500	ABCD + ABE + CDE + AB + AC + B
6	Y	10	1740	ABCD + ABC + ABE + AC + DE + B
6	-	9	360	ABCDE + ABC + ADE + BD + CE + B
6	-	9	660	ABCDE + ABC + CDE + AB + AD + E
6	-	9	120	ABCDE + ABCD + ABE + CDE + AC + B
7	-	9	4110	ABCDE + ABC + ADE + AB + DE + C + D
7	-	9	1320	ABCDE + ABC + ABD + CDE + AC + DE + B
7	-	9	360	ABCDE + ABCD + ABE + CDE + AC + B + 1

Table 5.22: All orbits in $n = 5$ with maximum orbit length (79 994 880)

5.2.5 Details of orbit lengths in $n = 5$

The orbits in $n = 5$ with the largest length (i.e., the largest numbers of different Boolean functions in the orbit of a representative) are listed in Table 5.22, the length being 79 994 880. In this table, the algebraic thickness is listed in the leftmost column, followed by whether the orbits are balanced or not, then the nonlinearity of the orbit, and the number of rigid functions in the orbit. None of the orbits are semi-bent (as can be seen from their $\mathcal{N}$-value), and none of the orbits contain any homogeneous functions.

In total there are 52 unique orbit lengths, detailed in Table 5.23. As this table shows, the orbit length of the most representatives (43) is 6 666 240. The five orbits that have unique orbit lengths are:

Representative	Orbit Length
A	62
ABCD + E	15 872
ABC + D	59 520
ABC + AD + B	312 480
ABC + ABD + ACE + BC + DE + A	5 332 992

The orbits of smallest lengths are trivially the constant $f(\mathbf{x}) = 0$ and $f(\mathbf{x}) = 1$ functions, where both orbits only contain themselves. The second shortest lengths are 32, represented by ABCDE and (ABCDE + 1). Further work on orbit lengths is discussed in Section 6.1.3.

Length	Orbits	Length	Orbits
6 666 240	43	317 440	4
39 997 440	26	208 320	4
19 998 720	26	104 160	4
9 999 360	25	52 080	4
1 666 560	19	4960	4
3 333 120	18	8 888 320	3
833 280	18	1 249 920	3
26 664 960	14	8680	3
2 222 080	14	952 320	2
4 999 680	13	624 960	2
79 994 880	12	83 328	2
2 499 840	10	17 360	2
444 416	9	14 880	2
416 640	8	13 888	2
277 760	8	7440	2
53 329 920	6	1240	2
2 666 496	6	992	2
166 656	6	620	2
138 880	6	496	2
119 040	6	32	2
34 720	6	1	2
31 997 952	5	5 332 992	1
13 332 480	5	31 2480	1
27 776	5	59 520	1
4 444 160	4	15 872	1
555 520	4	62	1

Table 5.23: The 52 unique orbit lengths in $n = 5$

5.3 General Results and Analysis

In general, for $n \leq 5$ variables, the number of representatives with maximum algebraic thickness is low compared to the other thickness values (and especially compared to $|\mathcal{B}_n|$). In $n < 5$ there is *one* such representative, and in $n = 5$ there are four. A common property shown in all values of $n \leq 5$ is that none of the functions with maximum algebraic thickness are balanced nor bent (or semi-bent if odd n). However, in both $n = 4$ and $n = 5$, there are orbits with maximum thickness that have the second-highest nonlinearity.

Boolean functions used in cryptographic systems should, as Carlet details in [9] and [5], have high algebraic degree, high nonlinearity, and be balanced. In relation to algebraic thickness, all functions with maximum thickness (at least for $n \leq 5$, but this seems to be a general case) have maximum algebraic degree, and none of them have maximum nonlinearity (although, the nonlinearity is still high). The balanced functions in $n = 4$ only have a maximum thickness of 3 out of 5, but in $n = 5$ they can be found to have 7 out of 8.

5.3.1 Symmetric Property of Thickness Distribution

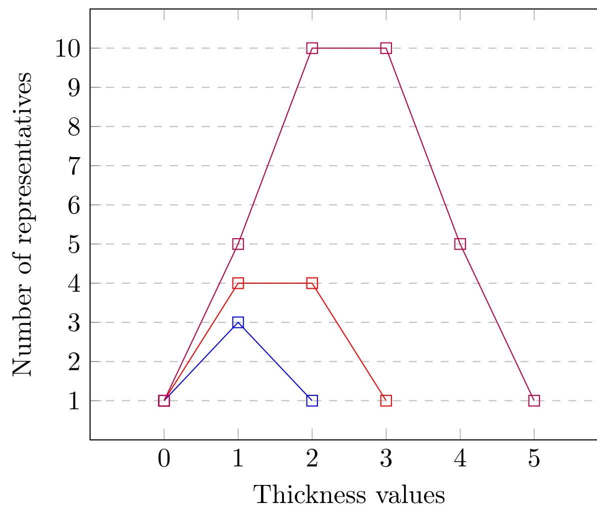


Figure 5.2: Visual representation of distribution of representatives in $n = 2, 3, 4$

As discussed in Section 4.3 and shown in Table 4.3, the number of Boolean functions with m monomials, for $m \in \{0, \ldots, 2^n\}$, follows a normal distribution, as the number is related to the binomial coefficient $\binom{2^n}{m}$. When the distribution of representatives in $n = 2, 3, 4$ was discovered – following the completion of the brute-force program – the first obvious pattern to notice, shown in Table 5.1 in the columns for $n = 2, 3, 4$, was that the distribution was symmetric:

n	Distribution
2	1, 3, 1
3	1, 4, 4, 1
4	1, 5, 10, 10, 5, 1

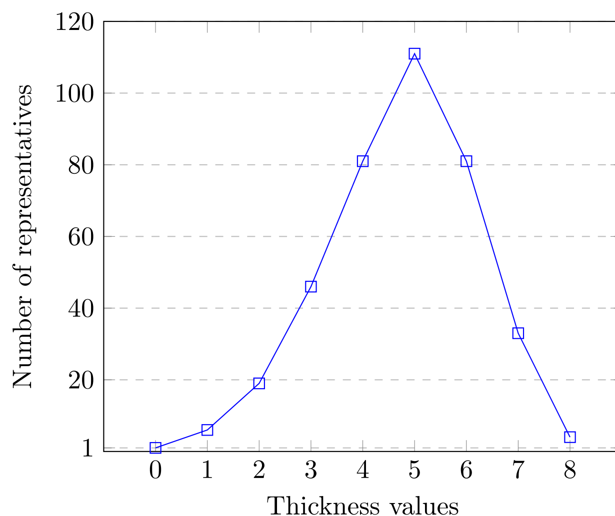


Figure 5.3: Visual representation of distribution of representatives in $n = 5$

This symmetry is shown visually in Figure 5.2.

However, after the completion of the program for $n = 5$, this was shown not necessarily to be the case for higher values of n, as can be seen in the distribution of the column for $n = 5$ in Table 5.1, which is illustrated here in Figure 5.3. This apparent symmetry in the distribution of representatives in $n = 2, 3, 4$ is therefore thought to be coincidental, perhaps as a result of the low amount of orbits in the lower values of n. It is worth noting that the distribution of representatives in $n = 1$ is also not symmetric, which could indicate that the symmetry only holds for even n, as it is the case that some properties do not hold for Boolean functions in odd n (e.g. bent functions), and that the distribution for $n = 3$ is the coincidence – this, however, is only speculation. It would be of interest to inspect the distribution of $n = 6$, to see if this could reveal information regarding this subject. (See Section 6.1.2 for thoughts on further work related to calculation of sequences in the distribution of algebraic thickness.)

5.4 Validity of shown programs and results

All built-in methods in SageMath were rigorously tested, and studied in the source code of their packages, to make sure the data collected was as intended. The generation of all Boolean functions (Section 3.2.1) follows its mathematical definition, but even then the resulting sets of functions where checked, ensuring that all functions were unique and accounted for. By having these complete function sets, all orbits of all representative functions were checked, and shown to be unique and non-intersecting.

Since the sum of all calculated orbit lengths in $n \leq 5$ variables (given in Tables 5.10 and 5.19, for $n = 4, 5$; respectively) is equal to 2^{2^n} – which is the size of $|\mathcal{B}_n|$ – it is reasonable to conclude that the shown orbits are correct, since the intersection of the produced sets are empty. Additionally, when collecting the data given in Sections 5.1.1 and 5.2.1, the monomial count of each function in the orbit of the representatives was checked, ensuring that the stated algebraic thickness was indeed correct. (Some of this data was used for the illustration of the orbit of $f = x_1x_2x_3 + x_1x_4 + x_2$, given in the introduction of this thesis.)

The nonlinearity distribution shown in Table 5.2, by iteration through *all functions* in $\mathcal{B}_n$ and with calculation in SageMath (using the previously mentioned *boolean_function*-package), is equal to the distribution listed in [26], thereby further confirming the calculation of nonlinearity is correct.

Furthermore, as noted in Section 2.4, the validity of the programs is connected to the validity of the respective mathematical definitions, which are given in Sections 2.1 and 2.2.

Chapter 6

Conclusions

$\mathcal{T}$	$n=0$	$n=1$	$n=2$	$n=3$	$n=4$	$n=5$
0	1	1	1	1	1	1
1	1	2	3	4	5	6
2	-	-	1	4	10	19
3	-	-	-	1	10	46
4	-	-	-	-	5	81
5	-	-	-	-	1	111
6	-	-	-	-	-	81
7	-	-	-	-	-	33
8	-	-	-	-	-	4
Sum	2	3	5	10	32	382
$\max(\mathcal{T}_n)$	1	1	2	3	5	8

Table 6.1: Relisting: Distribution of representatives within each thickness value.

In summary, the main results and definite conclusions of this thesis, are summarized in Table 5.1, re-listed here in Table 6.1 – only for convenience. This table includes the total distribution of algebraic thickness for $n \leq 5$ variables, with the number of orbits and maximum thickness listed.

Each orbit through affine transformations can be represented by *one* Boolean function: the representative function, which is the *smallest* function in its orbit, i.e., it has the minimum sum of degrees in its ANF, and is represented by the lowest indexed variables; further defined in Section 4.2, Definition 4.2. The number of equivalence classes in $\mathcal{B}_n$ is given by Harrison in [16], which is equal to the number of orbits (and thus, representatives) for any n, as discussed in Section 4.2.

By using the concepts of rigid- and representative functions defined in Sections 4.1 and 4.2, the thickness distribution of $n \leq 5$ can be calculated in significantly less time than the time estimation of a brute-force application, by (roughly) $2 * 10^6$ years.

The complete overview of all (orbit) representatives in $n = 4$ variables is given in Appendix E. Correspondingly, the complete overview of all representatives in $n = 5$ variables is given in Appendix F. These overviews include a full analysis of each orbit – by the definitions given in Section 2.2 – which includes: the algebraic degree and nonlinearity of the given orbits, the number of homogeneous functions, the number of rigid functions, whether the orbits are balanced, and whether the orbits are bent (respectively semi-bent); and the full size of each orbit.

The representatives are given in the ABC-form, discussed in Section 5.1, with the intention to avoid having implementation specifics obscure the results, and for ease of reading. Correspondingly, the list of all representatives in $n \leq 4$ variables is given in the commonly used $x_1x_2x_3$-form in Appendix D.

Regarding nonlinearity, the maximum in 4 variables is 6 and the maximum in 5 variables is 12, and it is known that there are 896 functions in $\mathcal{B}_4$ with $\mathcal{N} = 6$, and 27 387 136 functions in $\mathcal{B}_5$ with $\mathcal{N} = 12$. (cf. [26], and Table 5.2)

Since nonlinearity is an affine invariant (see Section 2.2), the nonlinearity of a representative ρ equals the nonlinearity of any affine transformation of ρ. Therefore, the nonlinearity distribution of all orbits in $n = 2, 3, 4, 5$ variables is given in Table 5.3 ($n = 0, 1$ excluded, as they are linear), comparing lower to higher numbers of variables. From this table, we see that there are two orbits in $\mathcal{B}_4$ with the maximum nonlinearity, referred to as the two *bent orbits* of $\mathcal{B}_4$. In $\mathcal{B}_5$, there are 14 orbits with the maximum nonlinearity, and by the analysis shown in Section 5.2, nine of these are semi-bent (by Definition 2.9). In total there are 14 054 656 semi-bent functions in $\mathcal{B}_5$.

None of the orbits with maximum *algebraic thickness* have the maximum nonlinearity, in $n = 4, 5$ variables.

The 12 870 balanced functions in $\mathcal{B}_4$ all belong to one of four *balanced orbits*, as shown in Section 5.1.3, where the maximum nonlinearity of these orbits is 4 (i.e., there are no bent functions), and the maximum algebraic thickness is 3.

The 601 080 390 balanced functions in $\mathcal{B}_5$ belong to 38 orbits, and are given in Section 5.2.3. Of these 38 orbits, four have maximum nonlinearity, where three are semi-bent. The maximum algebraic thickness of a balanced function in $\mathcal{B}_5$ is 7.

None of the orbits with maximum *algebraic thickness* are balanced, in $n = 4, 5$ variables.

The designed programs listed and explained in Chapters 3 and 4 show two distinct strategies of calculating algebraic thickness distribution in n variables: the former using brute-force, checking every function; and the latter using the rigid (and representative) functions as a tool to quickly dismiss functions not of interest. Also used are the variable permutations (as remarked below Definition 2.4) to further lower the size of the set to be searched. For instance, as described in Section 4.3, to check if there are any rigid functions with 8 terms in their ANF (monomial count) in five variables, there are $\binom{2^5}{8} = 10\ 518\ 300$ functions, which can be reduced to 100 577 after variable permutation. A significantly lower number than 2^{2^5}.

In short, the concept of the program used for $n = 5$ can be summarized as such: (1) Any function that can be mapped through affine transformations

to a function with lower monomial count can immediately be discarded. (2) If no rigid functions are found in a given monomial count m, then there are no functions with thickness m.

6.1 Further Work

Having the distribution for algebraic thickness determined for functions in $n \leq 5$ variables may have implications for larger values of n, and the rigid (and representative) functions of these function sets can be a tool for further exploration. In a simple sense, the obvious future projects of interest would be to determine the algebraic thickness distribution in $n = 6, 7, 8$, with special interest in the maximum $\mathcal{T}$ of these variable counts, and determining the individual distributions of representatives within each $\mathcal{T}$-value.

With a focus on the monomial counts of the Boolean functions in n variables, as explained in Section 4.3, the total number of functions to check can be reduced significantly. For instance, as shown in Section 4.4, there are $\binom{2^5}{9} = 28\ 048\ 800$ functions in five variables with nine terms in their ANF, and this number can be reduced down to 258 093 functions by variable permutation – as mentioned, this was part of proving that the maximum $\mathcal{T}_5 < 9$.

However, because of the (more than exponential) growth of the size of $\mathcal{B}_n$, attempts to reveal the distribution of these function sets may require alternative methods of implementation and calculation. Thoughts related to this are discussed in Sections 6.1.1 and 6.1.5 below, and a method for generating representative functions is discussed in Section 6.1.4.

In this thesis, it is shown that the number of representatives in n variables is equal to the number of equivalence classes under affine transformations, but the full distribution of algebraic thickness within each specific thickness value in n variables could be explored further. This concept is discussed in Section 6.1.2 below.

Explored briefly in Chapter 5, specifically in Section 5.2.5, and listed for each orbit in $n = 4, 5$ variables in Appendices E and F, are the orbit sizes (or lengths) of the representatives. A pattern found appertaining to these is described in Section 6.1.3, with some examples. There is potential for further study of these.

Finally, in Section 6.1.6, two conjectures are stated with regards to rigid functions, of which no proof has been found as of this time. Proving these conjectures could have implications for further use in revealing the algebraic thickness distribution of higher values of n. In the first conjecture, a weaker lemma is proved, which might be of aid in proving the conjecture itself.

6.1.1 Rotation Symmetric functions

Exploring various classes of Boolean functions in $n > 5$ variables may be a good first approach. The *rotation symmetric* functions can be a first step in this, a class of functions defined in [10] as the following.

First, let $(x_1, x_2, \ldots, x_n) \in \mathbb{F}_2^n$, and for $1 \leq k \leq n$, define

$$\rho_n^k = \begin{cases} x_{i+k} & \text{if } i+k \leq n, \\ x_{i+k-n} & \text{if } i+k > n. \end{cases}$$

(cf. the modulo operation.) This definition is then further extended to tuples and monomials, and is used to define the rotation symmetric functions:

Definition 6.1. (Rotation Symmetric Functions) [10]
A Boolean function f is *rotation symmetric* if and only if for any $(x_1, \ldots, x_n) \in \mathbb{F}_2^n$,

$$f(\rho_n^k(x_1, \ldots, x_n)) = f(x_1, \ldots, x_n)$$

for any $1 \leq k$.

In other words, rotation symmetric functions are invariant under shifting indices, i.e. the cyclic group. In this case, the size of the sets of Boolean functions of interest is reduced significantly, i.e. instead of 2^{2^n} functions to check, there are 2^6, 2^8, 2^{14}, and 2^{20} functions, for respectively $n = 4, 5, 6, 7$. For more on rotation symmetric functions, refer to [10].

6.1.2 Thickness Sequences

Number sequences can be dangerous[1] to attempt to analyse, especially without enough data. The fact that the number of representatives for each value of n is equal to the number of equivalence classes in the same n, should be quite clear. But, for instance, observing the maximum thickness value for $n \leq 5$, it could be tempting to attempt to prove that this sequence follows the Fibonacci-sequence – as discussed and disproved in the beginning of Chapter 5.

Nonetheless, it may be interesting to study some of the sequences of the number of representatives within each algebraic thickness value as n grows, and there are two trivial number sequences in Table 5.1, listed here:

1. The number of representatives in $\mathcal{T}_n = 0$, which by definition will only contain one function: the constant 0 function, $f(\mathbf{x}) = 0$. No other function will have zero monomials.

2. The number of representatives in $\mathcal{T}_n = 1$, which can be seen to grow by 1 as n grows, is equal to $(n+1)$. This is, of course, because the only functions with *one* monomial after variable permutations are $x_1, \ldots, x_1 \cdots x_n$, of which there are n possibilities, plus the constant 1 function, $f(\mathbf{x}) = 1$. Thus, the number of representatives in $\mathcal{T}_n = 1$ is $n + 1$.

The remaining number sequences of Table 5.1 are harder to analyse without further data at this time. From the work of Harrison, in [16] (see Table 4.2 for a full overview of the sizes in $n \leq 6$ variables), it is known that the number of equivalence classes in $n = 6$ is 15 768 919, and thus the number of orbits (and therefore representatives) is the same. Since this number is quite high in comparison with the lower n (i.e., $n < 6$), it is fair to assume that the distribution of orbits in $n = 6$ within each thickness value will be higher as well. Thus, the number of orbits in – for instance – $\mathcal{T}_6 = 2$ could be much higher than one would expect from the current growth of $\mathcal{T}_n = 2$ for $n < 6$:

[1] Or at least, a potential waste of valuable time.

$$0, 0, 1, 4, 10, 19.$$

However, there are only $\binom{64}{2} = 2016$ functions with monomial count 2 in $n = 6$ variables, and $\binom{128}{2} = 8128$ in $n = 7$. Determining the number of representatives with $\mathcal{T}_n = 2$, in $n = 6, 7$ should be possible – by using the techniques described in this thesis – and may give a pointer to what the sequence of $\mathcal{T}_n = 2$ could be, for any n. (See Section 6.1.5 for some notes on implementing methods from this thesis for higher n.)

6.1.3 Orbit lengths and function pairs

Many of the representative functions listed in Appendices E and F come in "pairs", where a representative ρ with *no* constant (i.e., ρ consists of various monomials with algebraic degree minimum 1) has a twin representative *with* a constant, i.e. $\rho + 1$, where, when both ρ and $\rho + 1$ are representative functions, $\mathcal{T}_n(\rho + 1) = \mathcal{T}_n(\rho) + 1$. The orbit lengths of these twins are equal. Examples of this can be found in the referenced appendices: choose a representative without a constant, and check if it has a "pair partner"; if it does, then observe the orbit lengths. This seems to be a general case.

The only representative of $\mathcal{T}_5 = 8$ in $n = 5$ that does not have a constant, as given in Section 5.2.4 (for reference, we call it ϕ), is:

$$\phi = \text{ABCDE} + \text{ABC} + \text{ABD} + \text{ACE} + \text{BC} + \text{DE} + \text{A} + \text{B}$$

The orbit length of ϕ is 53 329 920, and the *twin* of ϕ, by the logic explained above, is then:

$$\phi + 1 = \text{ABCDE} + \text{ABC} + \text{ABD} + \text{ACE} + \text{BC} + \text{DE} + \text{A} + \text{B} + 1$$

Naturally, $\phi + 1$ (as given here) is not a representative, since that would imply it was a rigid function of monomial count 9 – which is impossible in $n = 5$. Mapping $\phi+1$ to its $x_1x_2x_3$-form, by the inverse of the map given in Appendix D, and finding its representative, yields:

$$\text{ABCDE} + \text{ABC} + \text{ABD} + \text{CDE} + \text{AC} + \text{BE} + \text{D},$$

which has $\mathcal{T} = 7$ and indeed has the same orbit length as ϕ – of course, with a *lower* thickness, because $\phi + 1$ is itself *not* a representative.

However, there is also another representative with similar structure to ϕ:

$$\psi = \text{ABCDE} + \text{ABC} + \text{ABD} + \text{ACE} + \text{BC} + \text{DE} + \text{B}$$

where $\phi = \psi + \text{A}$. These two orbits also have the same length. We suspect, therefore, that it could be the case that given any function f with orbit length m, then the orbit length of $(f + \ell)$ is $c * m$, where ℓ is any affine function, and $c > 0$ is an integer – and when $\ell = 1$, $c = 1$ as well. E.g., in $n = 5$, AB has length 140, AB + 1 has length 140, AB + C has length $840 = 6 * 140$. This has not been checked further than shown here, and therefore needs verification and specification before any projects are attempted.

The patterns in the structure of the representatives may prove to be useful for continued generation of representatives, and may be used in revealing the thickness distribution for $n > 5$. In this case, further study is needed. (Refer to Section 6.1.4 for more on this.)

Remark also that all non-trivial (i.e., excluding constant 0 and 1 representatives) orbit lengths in $n = 4, 5$ are divisible by 2, which can be seen in Table 5.23, or in the respective representative appendices. The reason for this is currently unknown to us, and working on this could not be prioritized at this time – which is why it is given here, as a potential future project for further study.

6.1.4 Generating representative functions

By studying the representatives in $\mathcal{R}_4$ and $\mathcal{R}_5$, listed in Appendices E and F, there are some patterns emerging immediately, one of which is described in Example 6.1. Determining an exact algorithm for construction of representative functions in $\mathcal{R}_{n+1}$ given $\mathcal{R}_n$, encompassing these patterns, may be a method for uncovering the distribution of thickness in larger values of n. Generation of functions within specific classes is not a new concept for Boolean functions, for instance the construction of bent Boolean functions, where much literature has been written on their construction (see [10] for further reference, and examples).

Example 6.1. In $\mathcal{T}_n = 2$ for $n = 4, 5$, the representatives with algebraic degree 2 are:

1. AB + 1,
2. AB + C,
3. AB + CD.

From this, it is clear to see that in all three representatives the first term is the monomial with degree 2, while the second term is (1) zero variables (i.e. $x_i^0 = 1$), (2) one variable, and (3) two variables. In all three instances, the second term does not contain a variable given in the first. These three instances are the same for $n = 4, 5$, and we believe these are the only representatives in any n where $\mathcal{T}_n = 2$ and the algebraic degree is 2. Looking further into $\mathcal{T}_n = 2$, for algebraic degree 3, the representatives are

1. ABC + 1, $(n = 4, 5)$
2. ABC + A, $(n = 4, 5)$
3. ABC + D, $(n = 4, 5)$
4. ABC + AD, $(n = 4, 5)$
5. ABC + DE, $(n = 5)$
6. ABC + ADE. $(n = 5)$

Here, (3) conforms to the pattern mentioned previously, where the second term is not contained in the first, but in (2), this is not true. A few attempts were made while working on this thesis to define and construct these patterns, but as none of them bore fruit, they were abandoned and other work was prioritized.

Defining a language (and algebra?) for discussion surrounding these patterns, and generating all representatives in $n \leq 5$, then attempting to generate all 15 768 919 representatives in $n = 6$ could be a possible way to reveal the complete distribution of $\mathcal{T}_6$.

6.1.5 Similar implementation for $n > 5$

A project similar to this thesis could be attempted for values of $n > 5$, using the concepts and claims as defined and explained earlier. In undertaking such a task, there are some challenges which will be discussed here, with possible solutions, or otherwise items to consider.

The size of $\mathcal{B}_n$ for such values can be quite intimidating. Conducting an exhaustive search and visiting all functions in $n = 6, 7, 8$ variables can be considered impossible, even with a supercomputer available. As discussed in Section 4.3, a focus on searching for rigid functions within the specific monomial count values of the functions as a strategic approach can prove to be useful, and – in that case – only checking the rigidness of variable permutation unique functions should be of interest, as done in this thesis. For n variables there are $n!$ permutation maps, which for $n < 8$ should not be too large.

Instead of generating *all* functions in $\mathcal{B}_n$ and from the generated set picking the functions with the specified monomial count, one can modify the construction of Boolean functions (see (3.1) in Section 3.2.1) such that only the functions of interest are generated. This is a simple task, and would generate the needed functions efficiently.

In searching for the maximum thickness within n variables, taking the midpoint of Carlet's lower- and upper-bounds for $\mathcal{T}$ as a starting point, then continuously choosing a midpoint up or down the monomial count values based on rigid functions being found or not, should be a good start. E.g. for $n = 7$, the lower bound is 8 and upper bound is 85. Choose monomial count 46 as starting point. If no rigid functions are found, take the next midpoint lower down, that is, 27. If rigid functions are found here, check the next midpoint higher up (between 27 and 46), which is 36. Continue this process until rigid functions are found in m monomials, but none exist in $(m + 1)$. (See also Conjecture 6.2, in Section 6.1.6, for a possible applicable property, which (if proven) would speed up this process further.)

If there are no rigid functions in a given monomial count value – especially if said value is far off from the maximum – the iteration through all such functions processes quickly, as evidenced in Section 4.4.4. (Several iterations that processed thousands of functions took a few seconds.)

Other methods can also be implemented. There is still more work to be done regarding rigid- and representative functions, some of which is presented in the next section (see Section 6.1.6).

The main issue with an implementation similar to the programs presented in this thesis, will be the storing and creation of affine transformations. Here, these were generated and stored through invertible matrices and vectors over $\mathbb{F}_2$. Using the method presented in Section 4.4.2, all (invertible) affine transformations in $n = 4$ are generated in seconds, and in $n = 5$ are generated in approximately 25 minutes. Additionally, keeping these transformations *in* memory, per program, used less than 1 GB of RAM for $n = 4$, and 12-15 GB for $n = 5$ – showing that this number grows fast.

For $n > 5$, an alternative method for construction and/or storing of affine transformations may be needed. In this project, a "low memory" approach was also designed, for use on resources with low amount of available RAM. This approach successfully reduced the RAM needed down to 4 GB for $n = 5$, but was time consuming (as the transformations were generated once per function).

6.1.6 Conjectures

While studying the rigid and representative functions as defined in Chapter 4, many patterns were found and explored briefly, as previously mentioned. Analysis of some of these patterns resulted in the formulated theorem, corollary, and lemma given in Section 4.1, and in other cases the patterns either did not yield any interesting results, or were proven to not be the case quite quickly (e.g. the Fibonacci-sequence). Presented as conjectures in the following are the patterns found that could not yet be explored further, because of time constraints, but could be interesting to prove or disprove.

Multiplication with new variable conserves thickness

From Section 4.1, following the definition of rigid functions, (i.e. Definition 4.1), it is known that all rigid functions in $\mathcal{B}_n$ are also rigid functions in $\mathcal{B}_{n+1}$, by Theorem 4.1. And in the corollary that follows, for $f \in \mathcal{B}_n$, $\mathcal{T}_n(f) = \mathcal{T}_{n+1}(f)$ as well. (Recall, from below Definition 2.10, that the subscript of $\mathcal{T}_n(g)$ denotes the algebraic thickness of $g \in \mathcal{B}_n$, i.e. in specifically n variables.) These properties give insight into the distribution of algebraic thickness in $(n+1)$ variables, when the distribution for n variables is known.

A similar concept that was discovered during this project, is summarized in the following conjecture. As of this moment, the general consensus (after a lengthy discussion) is that the conjecture is *false*, as there may be instances where the new variable introduces new eliminations that could not occur in lower amounts of variables – but no such instance is yet to be found, possibly because of the low amount of choices for $n \leq 3$. Therefore, it is listed here as further work, as it could be interesting to see if a proof could be found, either in proving or disproving the conjecture.

Conjecture 6.1 (Thickness conservation)**.** Given the vector of variables $\mathbf{x} = (x_1, ..., x_n)$, for any Boolean function $f \in \mathcal{B}_n$, let x_{n+1} be the new variable for Boolean functions introduced in $\mathcal{B}_{n+1}$. Then:

$$\mathcal{T}_{n+1}(f * x_{n+1}) = \mathcal{T}_n(f).$$

Experimentally, the conjecture has been checked for all functions $f \in \mathcal{B}_n$ for $n = 2, 3$, i.e. all functions in two variables have been multiplied and thickness calculated in their corresponding three-variable form – and the same for all functions in three variables. Some functions have also been checked for $n = 4$, but because of the calculation time for checking algebraic thickness of $f * x_5$, only a select few have been checked – of those checked, all were found to hold true.

A weaker version of this conjecture is given in the proof below:

Lemma 6.1. *Given the vector of variables $\mathbf{x} = (x_1, ..., x_n)$, for any Boolean function $f \in \mathcal{B}_n$, let x_{n+1} be the new variable for Boolean functions introduced in $\mathcal{B}_{n+1}$. Then:*

$$\mathcal{T}_{n+1}(f * x_{n+1}) \leq \mathcal{T}_n(f).$$

Proof. : Given a Boolean function $f \in \mathcal{B}_n$, with known algebraic thickness $\mathcal{T}_n(f) = t$, with variables $(x_1, \ldots, x_n)$, let $f_{min} \in \mathcal{B}_n$ be the representative function with monomial count t of the orbit of f, and let π denote the affine transformation such that $\pi(f) = f_{min}$. As before, x_{n+1} is the new variable introduced in $\mathcal{B}_{n+1}$.

In $\mathcal{B}_{n+1}$, then, $\pi'(f * x_{n+1}) = f_{min} * x_{n+1}$, by the transformation $\pi'(x_j) = \pi(x_j)$, for $j < (n+1)$, and $\pi'(x_{n+1}) = x_{n+1}$. Since f_{min} has monomial count t, $f_{min} * x_{n+1}$ also has monomial count t, and therefore $\mathcal{T}_{n+1}(f * x_{n+1}) \leq \mathcal{T}_n(f)$. □

Proven here, then, is that the thickness of a given function in higher variable counts cannot be higher than in lower counts. As for the stronger claim of equivalence, i.e., that the thickness can also not be lower, no proof is yet to be found, nor have we found a counter-example. Therefore it is given here as a conjecture.

There are no gaps in thickness distribution

From observing the algebraic thickness distributions listed in Table 5.1, visualized in Figures 5.2 and 5.3, it is trivial to see that, for $n \leq 5$ and $m > 0$, if there exists a representative with $\mathcal{T}_n = m$, then there exists a representative with $\mathcal{T}_n = (m-1)$, and conversely: if there are no representatives with $\mathcal{T}_n = (m-1)$, then there are no representatives with $\mathcal{T}_n = m$. The following conjecture is an extension of Lemma 4.2.

Conjecture 6.2. For any n, in any given monomial count $m \leq 2^n$: if there are no rigid functions with m monomials, then for any $f \in \mathcal{B}_n$,

$$\mathcal{T}_n(f) < m.$$

The idea here is that if there are no rigid functions in a set monomial count m, then there are no rigid functions in any monomial count M, where $M > m$. Proving this would have implications for further attempts at determining maximum algebraic thickness (and the following thickness distribution) using the methods described in this thesis, as finding no rigid functions in n variables with monomial count (e.g.) 2^{n-1} would imply there are no rigid functions with monomial count greater than 2^{n-1}, thus eliminating half of the set of functions to search through. (Cf. the implementation discussed in Section 6.1.5.)

Bibliography

[1] Anderson, I., *A First Course in Discrete Mathematics*, Springer-Verlag London Ltd., 2001.

[2] Bourque, P., and Fairley, R. E., *Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0.* IEEE Computer Society Press, Washington, DC, United States, 2014.

[3] Boyar, J., and Find, M. G., *Constructive Relationships Between Algebraic Thickness and Normality.* In Fundamentals of Computation Theory, Lecture Notes in Comput. Sci., 9210, Springer, Cham. 2015, pp. 106—117. doi: 10.1007/978-3-319-22177-9_9.

[4] Carlet, C., *On the Degree, Nonlinearity, Algebraic Thickness, and Nonnormality of Boolean Functions, With Developments on Symmetric Functions* IEEE Trans. on Inf. Theory 50:9 (September 2004), pp. 2178–2185.

[5] Carlet, C., *Correlation Immune and Resilient Boolean Functions.* In: van Tilborg H.C.A., Jajodia S. (eds) Encyclopedia of Cryptography and Security (2011). Springer, Boston, MA.

[6] Carlet, C., *Boolean Functions for Cryptography and Error Correcting Codes* Chapter of the monography "Boolean Models and Methods in Mathematics, Computer Science, and Engineering", (2010) pp. 257–397. Cambridge University Press, Yves Crama and Peter L. Hammer (eds.).

[7] Carlet, C., *Boolean Functions.* In: van Tilborg H.C.A., Jajodia S. (eds) Encyclopedia of Cryptography and Security, (2011) pp. 162–164. Springer, Boston, MA.

[8] Carlet, C., *On Cryptographic Complexity of Boolean Functions.* In: Proc. 6th Conf. Finite Fields With Applications to Coding Theory, Cryptography and Related Areas., G. L. Mullen, H. Stichtenoth, and H. Tapia-Recillas, Eds., Springer, 2002, pp. 53–69.

[9] Carlet, C., *On the confusion and diffusion properties of Maiorana–McFarland's and extended Maiorana–McFarland's functions.* Journal of Complexity 20:2 (2004), pp. 182–204. Elsevier.

[10] Cusick, T. W., and Stănică, P., Cryptographic Boolean Functions and Applications (2nd ed.). Elsevier-Academic Press, 2017.

[11] Devlin, K., *The Joy of Sets: Fundamentals of Contemporary Set Theory – 2nd ed.*, Springer-Verlag New York, Inc., 1992.

[12] Dummit, D. S., and Foote, R. M., *Abstract Algebra (3rd ed.)* Hoboken, N.J. 2004.

[13] Fraleigh, J. B., *A First Course in Abstract Algebra (7th ed.)*, Pearson Education Ltd. 2014.

[14] Gangopadhyay, S., Pasalic, E., and Stănică, P., *A note on generalized bent criteria for Boolean functions.* IEEE Trans. Inf. Theory, vol. 59, no. 5, pp. 3233-3236. May 2013.

[15] Haggarty, R., *Discrete Mathematics for Computing.*, Pearson Education Ltd. 2002.

[16] Harrison, M. A., *On the classification of Boolean functions by the general linear and affine groups,* Journal of the Society for Industrial and Applied Mathematics, vol. 12, no. 2, pp. 285–299, 1964.

[17] Lay, D. C., *Linear Algebra and Its Applications (International 4th Ed.)*, Pearson Education Inc., 1994.

[18] Martin, R. C., *Agile Software Development, Principles, Patterns, and Practices (International Ed.)*, Pearson Education Limited, 2013.

[19] Meier, W., and Staffelbach, O., *Nonlinearity criteria for cryptographic functions,* in Advances in Cryptology, EUROCRYPT' 89 (Lecture Notes in Computer Science). Berlin, Germany: Springer-Verlag, 1990, vol. 434, pp. 549–562.

[20] Online Encyclopedia of Integer Sequences - Sloan.
https://oeis.org/search?q=3%2C5%2C10%2C32. First Accessed: 2020-02-05.

[21] Paar, C., and Pelzl, J., *Understanding Cryptography*, Springer-Verlag Berlin Heidelberg, 2009.

[22] Python Documentation.
https://docs.python.org/. First Accessed: 2019-03-10.

[23] Riera, C., Sole, P., and Stănică, P., *A complete characterization of plateaued Boolean functions in terms of their Cayley graphs,* Proc. Africacrypt (Marrakesh-Morroco), LNCS, Springer-Verlag LNCS 10831, 2018, pp. 3-10.

[24] Rout, R. K., Choudhury, P. P., and Sahoo, S., *Classification of Boolean Functions Where Affine Functions Are Uniformly Distributed.* Journal of Discrete Mathematics, 2013. (Article ID 270424) (2013), 12 pages. doi:10.1155/2013/270424.

[25] SageMath- Open-Source Mathematical Software System.
http://www.sagemath.org/. First Accessed: 2019-03-10.

[26] Sertkaya, I. and Doğanaksoy, A. (2010). *On the Affine Equivalence and Nonlinearity Preserving Bijective Mappings.* IACR Cryptology ePrint Archive. 2010. 655.

[27] Werman, M. *Affine Invariants.* In: Ikeuchi K. (eds) Computer Vision. Springer, Boston, MA. 2014

Appendix A

Data: $n = 2$ Raw Data

The results for $n = 2$ is listed below, as generated by the program listed in Appendix B – results for $n = 3, 4$ consist of too many lines to be shown. Each line consists of six entries, each entry separated by "|", and the entry indices 0..5 represents a data point for all $(2^{2^n} - 2)$ functions (the Boolean polynomials $f(\mathbf{x}) = 0$ and $f(\mathbf{x}) = 1$ were excluded from these data sets):

Index	Data point
0	Algebraic Thickness
1	Nonlinearity
2	Vector Generating Integer
3	Matrix Generating Integer
4	Boolean Polynomial
5	Mapped Boolean Polynomial

The raw data for $n = 2$ therefore consists of the 14 lines as shown, and we can see that there are four Boolean polynomials with thickness 2, and ten with thickness 1. Please note that this data was generated before the concept of representatives was defined, and therefore the mapped functions are only rigid functions, not necessarily representative functions.

```
1|0|0|6|x1|x2
1|0|2|6|x1 + 1|x2
1|0|0|6|x2|x1
1|0|1|6|x2 + 1|x1
1|0|0|7|x1 + x2|x1
1|0|1|7|x1 + x2 + 1|x1
1|1|0|6|x1*x2|x1*x2
2|1|0|6|x1*x2 + 1|x1*x2 + 1
1|1|1|6|x1*x2 + x1|x1*x2
2|1|1|6|x1*x2 + x1 + 1|x1*x2 + 1
1|1|2|6|x1*x2 + x2|x1*x2
2|1|2|6|x1*x2 + x2 + 1|x1*x2 + 1
2|1|3|6|x1*x2 + x1 + x2|x1*x2 + 1
1|1|3|6|x1*x2 + x1 + x2 + 1|x1*x2
```

E.g., the vector generated by 2 is (1, 0) and the matrix generated by 7 is

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

Appendix B

Program: Brute-force calculation, $n = 4$

The following code is discussed in Chapter 3. The results of this program is discussed and analysed in Chapter 5, specifically Section 5.1, and listed in Appendix E.

```
1  from sage.crypto.boolean_function import BooleanFunction
2
3  ####### MATRIX GENERATION #######
4
5  # Generates invertible matrix maps and vectors
6  def generate_invertible_matrices_and_vectors(n_var, M, x_vector):
7      binary_list = []
8      base = ['0'] * n_var
9      vectors = [(0, vector(GF(2), base))]
10
11     for index in range(2^n_var)[1:]:
12         binary = list(bin(index))[2:]
13         vec = base[len(binary):] + binary
14         binary_list.append((index, vec))
15         vectors.append((index, vector(GF(2), vec)))
16
17     permutations = Arrangements(range((2^n_var)-1), n_var).list()
18
19     matrices = []
20     for permutation in permutations:
21         accumulator = []
22         value_list = []
23         for index in permutation:
24             value_list.append(binary_list[index][0])
25             accumulator += binary_list[index][1]
26         inv_matrix = M.matrix(accumulator)
27         if inv_matrix.is_invertible():
28             generating_int = matrix_value(n_var, value_list)
29             matrix_map = inv_matrix*x_vector
30             matrices.append((generating_int, matrix_map))
31     return matrices, vectors
32
33 # Calculate generating integer for given matrix
34 def matrix_value(n_var, value_list):
35     return sum( [ (value_list[i] * 2^(n_var*(n_var-i-1)))
36                 for i in range(len(value_list)) ] )
```

```
37
38 ####### POLYNOMIAL GENERATION #######
39 def generate_functions(variables):
40     generated_funcs = [0, 1]  # n = 0
41
42     # iterate through variables {x_1,.., x_n}
43     for new_x in variables:
44         temp_funcs = []
45         for f1 in generated_funcs:
46             for f2 in generated_funcs:
47                 temp_funcs.append( f1*new_x + f2 )
48         generated_funcs = temp_funcs
49
50     return generated_funcs
51
52 def sort_by_degree(func_list):
53     bool_pols = [ [] for i in range(n)]
54     add_function = lambda i: bool_pols[i-1].append
55     for f in func_list:
56         f_deg = f.degree()
57         if (f_deg > 0):
58             add_function(f_deg)(f)
59     return bool_pols
60
61 ####### MAIN METHOD #######
62 def calc_thickness(bool_pols_by_deg, matrices, vectors):
63     n_matrices = len(matrices)
64     n_vectors = len(vectors)
65
66     for bool_pols in bool_pols_by_deg:
67         n_bool_pols = len(bool_pols)
68
69         for f in bool_pols:
70             iter_mat = 0
71
72             min_mapped_monomials = 2^n + 1 # global var n
73             min_matrix = 0
74             min_vector = 0
75             min_new_f = 0
76
77             while (iter_mat < n_matrices):
78                 iter_vec = 0
79                 while (iter_vec < n_vectors):
80                     # Map the function, for n = 4
81                     map_ = matrices[iter_mat][1] +
82                             vectors[iter_vec][1]
83                     new_f = f(map_[0], map_[1], map_[2], map_[3])
84                     mapped_monomials = len(new_f.monomials())
85
86                     # Check if fewer monomials
87                     if (mapped_monomials < min_mapped_monomials):
88                         min_mapped_monomials = mapped_monomials
89                         min_matrix = matrices[iter_mat][0]
90                         min_vector = vectors[iter_vec][0]
91                         min_new_f = new_f
92
93                     # Move on, if minimal monomials
94                     if (mapped_monomials == 1):
95                         iter_mat = n_matrices
96                         iter_vec = n_vectors
97                     iter_vec += 1
98                 iter_mat += 1
```

```
99
100             # Write data to file
101             nonlin = BooleanFunction(min_new_f).nonlinearity()
102             toText = open(file_name, 'a')
103             toText.write("%d|%d|%d|%d|%s|%s\n" %
104             (min_mapped_monomials, nonlin, min_vector, min_matrix,
    f, min_new_f))
105             toText.close()
106     return
107
108
109 ############## PROGRAM ##############
110
111 ####### SETUP PHASE #######
112 # When running this program with several instances,
113 # start instance i+1 at the same number as instance i ended on.
114 # i.e., start=x starts at x, and
115 #        end=y is "up to but not including y"
116
117 n = 4
118
119 ## Division of functions per iteration
120 division = 16  # because this was what was available
121 print "Dividing functions into %d equal parts." % division
122 current_session = input("Which calculation iteration to run (0..%d)
    ?: "
123                     % (division-1))
124 function_iterations = range(0, 2^2^n, 2^2^n/division)
125
126 start = function_iterations[current_session]
127 if (current_session == division-1):
128     end = 2^(2^n)
129 else:
130     end = function_iterations[current_session+1]
131
132 file_folder = "" # Add folder if needed
133 file_alias = "results_n=%d_from_%d_to_%d.txt" % (n, start, end)
134 file_name = file_folder + file_alias
135
136 M = MatrixSpace(GF(2), n)
137 B_ring = BooleanPolynomialRing(n, 'x')
138 x = B_ring.gens()
139 xvec = vector(B_ring, x)
140
141 ## SETUP: Generation of matrices and vectors
142 matrices, vectors = generate_invertible_matrices_and_vectors(n, M,
    xvec)
143
144 ## SETUP: Construction of Boolean Polynomials
145 bool_pols = generate_functions(x)[start:end]
146 bool_pols_by_deg = sort_by_degree(bool_pols)
147
148
149 ####### MAIN PHASE: Thickness calculation
150
151 calc_thickness(bool_pols_by_deg, matrices, vectors)
152
153 ####### CLOSING PHASE: Summary printing
154 # [...] # Various printouts of summaries
155         # that are removed from this version.
```

Appendix C

Program: Representatives collection for $n = 5$

The following code is discussed in Chapter 4. The results of this program is discussed and analysed in Chapter 5, specifically Section 5.2, and listed in Appendix F.

```
# Method for converting a string into a Boolean Polynomial
def convert_to_boolpol(func_str, R):
    if (len(func_str) == 1):
        return R(func_str)

    # Remove whitespace, and split each monomial up
    monomials = func_str.replace(" ", "").split("+")

    # Convert each monomial string into a function monomial
    func_monomials = []
    add_monomial = func_monomials.append
    for mon in monomials:
        temp_monom = R(1)
        # if no * in string, return the single monomial
        if (mon.find("*") == -1):
            if (mon == "1"):
                temp_monom = R(1)
            else:
                temp_monom = x[int(mon[1])]
        # else go through monomial, multiplying each variable
        else:
            mult_mons = mon.split("*")
            for variable in mult_mons:
                temp_monom *= x[int(variable[1])]

        add_monomial(temp_monom)

    # Sum each function monomial into a complete function
    return sum(func_monomials)

# Collecting already-found representatives
def collect_funcs_of_size_from_file(mon_count, file_name):
    funcs = set()

    # Usable for Windows and Linux implementations
    file_stop = set(["\n", "\r", "\r\n", ""])
```

```
37
38      try:
39          data_file = open(file_name, 'r')
40          inp_string = data_file.readline()
41          while (inp_string not in file_stop):
42              line = inp_string.split('|')
43              f = convert_to_boolpol(line[0], ring)
44
45              if (len(f.set()) == mon_count):
46                  funcs.add(f)
47              inp_string = data_file.readline()
48          data_file.close()
49      except IOError:
50          print "No file: %s" % file_name
51      print "Collected %d representatives with size %d."
52            % (len(funcs), mon_count)
53      return funcs
54
55  # Collecting relevant functions for this iteration
56  def collect_remapped_functions(file_name):
57      funcs = []
58      file_stop = set(["\n", "\r", "\r\n", ""])
59
60      try:
61          data_file = open(file_name, 'r')
62          inp_string = data_file.readline()
63          while (inp_string not in file_stop):
64              line = inp_string.split('|')
65              f = convert_to_boolpol(line[0], ring)
66              funcs.append(f)
67              inp_string = data_file.readline()
68          data_file.close()
69      except IOError:
70          print "No file: %s\n\nABORT!\n" % file_name
71      print "Collected %d remapped functions." % len(funcs)
72      return funcs
73
74  def generate_invertible_matrices_and_vectors(n_var, M, x_vector):
75      binary_list = []
76      base = [0] * n_var
77      vectors = [vector(GF(2), base)]  # 0-vector added here
78
79      for index in range(1, 2^n_var):
80          binary = map(int, bin(index)[2:])
81          vec = base[len(binary):] + binary
82          binary_list.append(vec)
83          vectors.append(vector(GF(2), vec))
84
85      permutations = Arrangements(range((2^n_var)-1), n_var).list()
86
87      matrices = []
88      for permutation in permutations:
89          accumulator = []
90          for index in permutation:
91              accumulator += binary_list[index]
92          inv_matrix = M.matrix(accumulator)
93          if inv_matrix.is_invertible():
94              matrix_map = inv_matrix*x_vector
95              matrices.append(matrix_map)
96      return matrices, vectors
97
98
```

```
99  def calculate_thickness(functions, matrices, vectors, file_write):
100     for f in functions:
101         min_f = f
102         rigid_function = True # Assume rigid, attempt to disprove
103 
104         iter_mat = 0
105         while (iter_mat < n_matrices):
106             iter_vec = 0
107             while (iter_vec < n_vectors):
108                 # Apply map to function f
109                 f_map = matrices[iter_mat] + vectors[iter_vec]
110 
111                 mapped_f = f(f_map[0], f_map[1], f_map[2],
112                              f_map[3], f_map[4])
113                 mapped_monomials = len(mapped_f.set())
114 
115                 # If function not rigid, skip to next function
116                 if (mapped_monomials < n_monoms):
117                     rigid_function = False
118                     iter_mat = n_matrices
119                     iter_vec = n_vectors
120 
121                 # If representative already found,
122                 # skip to next function
123                 elif (mapped_f in representatives):
124                     rigid_function = False
125                     iter_mat = n_matrices
126                     iter_vec = n_vectors
127 
128                 # If function may be rigid,
129                 # check if better representative
130                 elif (mapped_monomials == n_monoms):
131                     if (mapped_f < min_f):
132                         min_f = mapped_f
133 
134                 iter_vec += 1
135             iter_mat += 1
136 
137         if (rigid_function):
138             # Map to lowest indexed version
139             min_f_rep = map_to_smallest_index(min_f)
140 
141             # Save results for each viable function to file
142             text_file = open(file_write, 'a')
143             text_file.write("%s|\n" % min_f_rep)
144             text_file.close()
145 
146             # Add found representative to set
147             representatives.add(min_f_rep)
148     return
149 
150 ## x_permutations is global variable, defined further down!
151 def map_to_smallest_index(f):
152     smallest = f
153     for f_map in x_permutations:
154         mapped_f = f(f_map[0], f_map[1], f_map[2],
155                      f_map[3], f_map[4])
156 
157         if (mapped_f > smallest): # note: x1 > x2 in .pbori
158             smallest = mapped_f
159     return smallest
160 
```

```
161 ##### COLLECTING INPUTS #####
162 n_monoms = int(raw_input("Monomial count = "))
163 division = -1
164 while (division < 3):
165     division = int(raw_input("Division of functions (>2): "))
166 iteration = -1
167 while (iteration not in range(division)):
168     iteration = int(raw_input("Iter (0..%d): " % (division-1)))
169
170 ##### BASIC SETUP #####
171 n = 5
172 ring = BooleanPolynomialRing(n, "x")
173 x = ring.gens()
174 m_space = MatrixSpace(GF(2), n)
175 x_vec = vector(ring, x)
176 x_permutations = Arrangements(x, n) # permutations of variables
177
178 file_folder = "" # Add as needed
179 file_read_name = file_folder + "n=%d_m=%d_functions.txt"
180                                 % (n, n_monoms)
181 functions_total = collect_remapped_functions(file_read_name)
182 n_functions = len(functions_total)
183
184 ##### ITERATIONS #####
185 print "Dividing %d functions into %d parts.\n"
186       % (n_functions, division)
187 each_iteration = int(n_functions/division)
188
189 # If divisible by division variable, two ends must be added
190 if (mod(n_functions, division) == 0):
191     function_iterations = range(each_iteration,
192     n_functions-each_iteration, each_iteration) +
193     [n_functions-each_iteration, n_functions]
194
195 # otherwise: only add the last end
196 else:
197     function_iterations = range(each_iteration,
198     n_functions-each_iteration, each_iteration) +
199     [n_functions]
200
201 if (iteration == 0):
202     start = 0
203 else:
204     start = function_iterations[iteration-1]
205 end = function_iterations[iteration]
206 functions = functions_total[start:end]
207
208 ##### COLLECTING FOUND REPRESENTATIVES #####
209 file_reps = file_folder + "n=%d_representatives_list.txt" % n
210 representatives = collect_funcs_of_size_from_file(n_monoms,
        file_reps)
211
212 ##### AFFINE TRANSFORMATIONS #####
213 matrices, vectors = generate_invertible_matrices_and_vectors(n,
        m_space, x_vec)
214 n_matrices = len(matrices)
215 n_vectors = len(vectors)
216
217 ############## FINDING REPRESENTATIVES ##############
218 file_write_name = file_folder + "n=%d_m=%d_it_%d_%d-%d_reps.txt"
219                             % (n, n_monoms, iteration, start, end)
220 calculate_thickness(functions, matrices, vectors, file_write_name)
```

Appendix D

List: $x_1x_2x_3$-form of representatives

The representatives are – in the thesis and in the following appendices (Appendices E and F) – presented in their ABC-form. As explained in Chapter 4, to avoid the visual aspect of the presentation being a problem, they are listed here in their $x_1x_2x_3$-form, as Boolean functions are usually presented in scientific literature.

Table D.1 is listed on the next page (because of its size) and is equal to Table 5.4, by the bijective substitution map:

$$\begin{aligned} \mathrm{A} &\mapsto x_1, \\ \mathrm{B} &\mapsto x_2, \\ \mathrm{C} &\mapsto x_3, \\ \mathrm{D} &\mapsto x_4; \end{aligned}$$

as should be expected.

$\mathcal{T}$	$n=0$	$n=1$	$n=2$	$n=3$	$n=4$
0	0	0	0	0	0
1	1	1	1	1	1
		x_1	x_1	x_1	x_1
			x_1x_2	x_1x_2	x_1x_2
				$x_1x_2x_3$	$x_1x_2x_3$
					$x_1x_2x_3x_4$
2			$x_1x_2 + 1$	$x_1x_2 + 1$	$x_1x_2 + 1$
				$x_1x_2 + x_3$	$x_1x_2 + x_3$
					$x_1x_2 + x_3x_4$
				$x_1x_2x_3 + 1$	$x_1x_2x_3 + 1$
				$x_1x_2x_3 + x_1$	$x_1x_2x_3 + x_1$
					$x_1x_2x_3 + x_4$
					$x_1x_2x_3 + x_1x_4$
					$x_1x_2x_3x_4 + 1$
					$x_1x_2x_3x_4 + x_1$
					$x_1x_2x_3x_4 + x_1x_2$
3					$x_1x_2 + x_3x_4 + 1$
				$x_1x_2x_3 + x_1 + 1$	$x_1x_2x_3 + x_1 + 1$
					$x_1x_2x_3 + x_1x_4 + 1$
					$x_1x_2x_3 + x_1x_4 + x_2$
					$x_1x_2x_3 + x_1x_2 + x_3x_4$
					$x_1x_2x_3x_4 + x_1 + 1$
					$x_1x_2x_3x_4 + x_1x_2 + 1$
					$x_1x_2x_3x_4 + x_1x_2 + x_1$
					$x_1x_2x_3x_4 + x_1x_2 + x_3$
					$x_1x_2x_3x_4 + x_1x_2 + x_3x_4$
4					$x_1x_2x_3 + x_1x_2 + x_3x_4 + 1$
					$x_1x_2x_3x_4 + x_1x_2 + x_1 + 1$
					$x_1x_2x_3x_4 + x_1x_2 + x_3 + 1$
					$x_1x_2x_3x_4 + x_1x_2 + x_3x_4 + 1$
					$x_1x_2x_3x_4 + x_1x_2 + x_3x_4 + x_1$
5					$x_1x_2x_3x_4 + x_1x_2 + x_3x_4 + x_1 + 1$

Table D.1: Lowest indexed representatives ($x_1x_2x_3$-form) in $n = 0, 1, 2, 3, 4$ – sorted by thickness, clustered by degree

Appendix E

List: Representatives in $n = 4$

The following list contains the 32 representative functions in $n = 4$ variables. The orbit of each representative has been analysed and checked for the seven following properties:

1. **D**: The algebraic degree of the orbit
2. **N**: The nonlinearity of the orbit
3. **H**: Number of homogeneous functions in the orbit
4. **Be**: Whether the orbit is bent or not (Y = bent)
5. **Ba**: Whether the orbit is balanced or not (Y = balanced)
6. **R**: Rigid functions in the orbit
7. **O**: Number of functions in the orbit

Representative	D	N	H	Be	Ba	R	O
Thickness 0							
0	0	0	1	-	-	1	1
Thickness 1							
1	0	0	1	-	-	1	1
A	1	0	15	-	Y	4	30
AB	2	4	25	-	-	6	140
ABC	3	2	10	-	-	4	120
ABCD	4	1	1	-	-	1	16
Thickness 2							
AB + 1	2	4	0	-	-	6	140
AB + C	2	4	10	-	Y	12	840
AB + CD	2	6	27	Y	-	3	448
ABC + 1	3	2	0	-	-	4	120
ABC + D	3	2	0	-	Y	4	1920

Representative	D	N	H	Be	Ba	R	O
ABC + A	3	2	0	-	-	12	840
ABC + AD	3	4	5	-	-	12	1680
ABCD + 1	4	1	0	-	-	1	16
ABCD + A	4	1	0	-	-	4	240
ABCD + AB	4	3	0	-	-	6	560
Thickness 3							
AB + CD + 1	2	6	1	Y	-	3	448
ABC + A + 1	3	2	0	-	-	36	840
ABC + AD + 1	3	4	0	-	-	12	1680
ABC + AD + B	3	4	0	-	Y	24	10080
ABC + AB + CD	3	4	0	-	-	42	6720
ABCD + A + 1	4	1	0	-	-	14	240
ABCD + AB + 1	4	3	0	-	-	6	560
ABCD + AB + C	4	3	0	-	-	12	6720
ABCD + AB + CD	4	5	0	-	-	3	4480
ABCD + AB + A	4	3	0	-	-	36	1680
Thickness 4							
ABC + AB + CD + 1	3	4	0	-	-	54	6720
ABCD + AB + C + 1	4	3	0	-	-	130	6720
ABCD + AB + CD + 1	4	5	0	-	-	3	4480
ABCD + AB + A + 1	4	3	0	-	-	42	1680
ABCD + AB + CD + A	4	5	0	-	-	42	2688
Thickness 5							
ABCD + AB + CD + A + 1	4	5	0	-	-	48	2688

Appendix F

List: Representatives in $n = 5$

The following list contains the 382 representative functions in $n = 5$ variables. The orbit of each representative has been analysed and checked for the seven following properties:

1. **D**: The algebraic degree of the orbit
2. **N**: The nonlinearity of the orbit
3. **H**: Number of homogeneous functions in the orbit
4. **Be**: Whether the orbit is semi-bent or not (Y = semi-bent)
5. **Ba**: Whether the orbit is balanced or not (Y = balanced)
6. **R**: Rigid functions in the orbit
7. **O**: Number of functions in the orbit

Representative	D	N	H	Be	Ba	R	O
Thickness 0							
0	0	0	1	-	-	1	1
Thickness 1							
1	0	0	1	-	-	1	1
A	1	0	31	-	Y	5	62
AB	2	8	90	-	-	10	620
ABC	3	4	65	-	-	10	1240
ABCD	4	2	15	-	-	5	496
ABCDE	5	1	1	-	-	1	32
Thickness 2							
AB + 1	2	8	0	-	-	10	620
AB + C	2	8	65	-	Y	30	8680
AB + CD	2	12	600	Y	-	15	13888

Representative	D	N	H	Be	Ba	R	O
ABC + 1	3	4	0	-	-	10	1240
ABC + D	3	4	0	-	Y	20	59520
ABC + DE	3	10	0	-	-	10	317440
ABC + A	3	4	0	-	-	30	8680
ABC + AD	3	8	75	-	-	60	52080
ABC + ADE	3	6	232	-	-	15	27776
ABCD + 1	4	2	0	-	-	5	496
ABCD + E	4	2	0	-	Y	5	15872
ABCD + A	4	2	0	-	-	20	7440
ABCD + AE	4	8	0	-	-	20	119040
ABCD + AB	4	6	0	-	-	30	17360
ABCD + ABE	4	4	15	-	-	30	34720
ABCDE + 1	5	1	0	-	-	1	32
ABCDE + A	5	1	0	-	-	5	992
ABCDE + AB	5	7	0	-	-	10	4960
ABCDE + ABC	5	3	0	-	-	10	4960
Thickness 3							
AB + CD + 1	2	12	16	Y	-	15	13888
AB + CD + E	2	12	252	Y	Y	15	27776
ABC + DE + 1	3	10	0	-	-	10	317440
ABC + A + 1	3	4	0	-	-	90	8680
ABC + DE + A	3	10	0	-	-	30	2222080
ABC + AD + 1	3	8	0	-	-	60	52080
ABC + AD + E	3	8	0	-	Y	60	833280
ABC + AD + B	3	8	0	-	Y	120	312480
ABC + AD + BE	3	12	15	Y	-	60	833280
ABC + ADE + 1	3	6	0	-	-	15	27776
ABC + ADE + B	3	6	0	-	-	60	833280
ABC + ADE + BD	3	10	210	-	-	60	1666560
ABC + ADE + A	3	6	6	-	-	15	27776
ABC + AB + CD	3	8	0	-	-	210	208320
ABC + CDE + AB	3	8	360	-	-	60	277760
ABCD + A + 1	4	2	0	-	-	70	7440
ABCD + AE + 1	4	8	0	-	-	20	119040
ABCD + AE + B	4	8	0	-	Y	60	1666560
ABCD + AB + 1	4	6	0	-	-	30	17360
ABCD + AB + E	4	6	0	-	Y	30	555520
ABCD + AB + C	4	6	0	-	-	60	208320
ABCD + AB + CE	4	10	0	-	-	60	3333120
ABCD + AB + CD	4	10	0	-	-	15	138880
ABCD + AB + A	4	6	0	-	-	180	52080
ABCD + ABE + 1	4	4	0	-	-	30	34720
ABCD + ABE + E	4	4	0	-	-	30	555520

Representative	D	N	H	Be	Ba	R	O
ABCD + ABE + C	4	4	0	-	Y	60	833280
ABCD + ABE + CE	4	8	0	-	-	60	6666240
ABCD + ABE + CD	4	10	0	-	-	30	2222080
ABCD + ABE + CDE	4	8	0	-	-	15	4444160
ABCD + ABE + A	4	4	0	-	-	60	104160
ABCD + ABE + AC	4	8	0	-	-	120	624960
ABCD + ABC + DE	4	8	0	-	-	80	119040
ABCD + ABC + ADE	4	6	0	-	-	210	416640
ABCDE + A + 1	5	1	0	-	-	20	992
ABCDE + AB + 1	5	7	0	-	-	10	4960
ABCDE + AB + C	5	7	0	-	-	30	138880
ABCDE + AB + CD	5	11	0	-	-	15	277760
ABCDE + AB + A	5	7	0	-	-	70	14880
ABCDE + ABC + 1	5	3	0	-	-	10	4960
ABCDE + ABC + D	5	3	0	-	-	20	119040
ABCDE + ABC + DE	5	9	0	-	-	10	317440
ABCDE + ABC + A	5	3	0	-	-	30	34720
ABCDE + ABC + AD	5	7	0	-	-	60	416640
ABCDE + ABC + ADE	5	7	0	-	-	15	277760
ABCDE + ABC + AB	5	5	0	-	-	90	34720
Thickness 4							
ABC + DE + A + 1	3	10	0	-	-	90	2222080
ABC + AD + BE + 1	3	12	0	Y	-	60	833280
ABC + AD + BE + C	3	12	0	Y	Y	60	1666560
ABC + ADE + B + 1	3	6	0	-	-	210	833280
ABC + ADE + BD + 1	3	10	0	-	-	60	1666560
ABC + ADE + BD + C	3	10	0	-	-	120	9999360
ABC + ADE + A + 1	3	6	0	-	-	15	27776
ABC + ADE + BD + A	3	10	0	-	-	60	1666560
ABC + AB + CD + 1	3	8	0	-	-	270	208320
ABC + CDE + AB + 1	3	8	0	-	-	60	277760
ABC + CDE + AB + D	3	8	60	-	-	330	3333120
ABC + CDE + AB + C	3	8	0	-	Y	30	555520
ABCD + AB + C + 1	4	6	0	-	-	650	208320
ABCD + AB + CE + 1	4	10	0	-	-	60	3333120
ABCD + AB + CE + D	4	10	0	-	Y	60	6666240
ABCD + AB + CD + 1	4	10	0	-	-	15	138880
ABCD + AB + CD + E	4	10	0	-	Y	15	444416
ABCD + AB + A + 1	4	6	0	-	-	210	52080
ABCD + AB + CD + A	4	10	0	-	-	210	83328
ABCD + ABE + E + 1	4	4	0	-	-	120	555520
ABCD + ABE + CE + 1	4	8	0	-	-	60	6666240
ABCD + ABE + CE + D	4	8	0	-	-	60	26664960

Representative	D	N	H	Be	Ba	R	O
ABCD + ABE + CD + 1	4	10	0	-	-	30	2222080
ABCD + ABE + CD + E	4	10	0	-	-	30	2222080
ABCD + ABE + CDE + 1	4	8	0	-	-	15	4444160
ABCD + ABE + CDE + E	4	8	0	-	-	15	4444160
ABCD + ABE + A + 1	4	4	0	-	-	90	104160
ABCD + ABE + CE + A	4	8	0	-	-	120	19998720
ABCD + ABE + CD + A	4	10	0	-	-	60	6666240
ABCD + ABE + CDE + A	4	8	0	-	-	60	26664960
ABCD + ABE + AE + B	4	6	0	-	-	60	833280
ABCD + ABE + AC + 1	4	8	0	-	-	120	624960
ABCD + ABE + AC + E	4	8	0	-	-	120	4999680
ABCD + ABE + AC + D	4	8	0	-	Y	120	4999680
ABCD + ABE + AC + DE	4	10	0	-	-	120	19998720
ABCD + ABE + AC + C	4	8	0	-	-	180	1249920
ABCD + ABE + AC + B	4	8	0	-	Y	120	1249920
ABCD + ABE + AC + BD	4	12	0	-	-	60	2499840
ABCD + ABE + CDE + AB	4	6	1	-	-	280	444416
ABCD + ABC + DE + 1	4	8	0	-	-	100	119040
ABCD + ABC + DE + A	4	8	0	-	-	420	833280
ABCD + ABC + ADE + 1	4	6	0	-	-	210	416640
ABCD + ABC + ADE + E	4	6	0	-	-	270	1666560
ABCD + ABC + ADE + D	4	6	0	-	-	60	833280
ABCD + ABC + ADE + B	4	6	0	-	-	480	4999680
ABCD + ABC + ADE + BD	4	10	0	-	-	300	4999680
ABCD + ABC + ADE + A	4	6	0	-	-	270	416640
ABCD + ABC + AB + DE	4	10	0	-	-	420	833280
ABCD + ABC + CDE + AB	4	8	0	-	-	300	1666560
ABCD + ABC + ABE + DE	4	8	0	-	-	920	3333120
ABCDE + AB + C + 1	5	7	0	-	-	340	138880
ABCDE + AB + CD + 1	5	11	0	-	-	15	277760
ABCDE + AB + CD + E	5	11	0	-	-	15	444416
ABCDE + AB + A + 1	5	7	0	-	-	80	14880
ABCDE + AB + CD + A	5	11	0	-	-	210	166656
ABCDE + ABC + D + 1	5	3	0	-	-	130	119040
ABCDE + ABC + DE + 1	5	9	0	-	-	10	317440
ABCDE + ABC + DE + D	5	9	0	-	-	110	952320
ABCDE + ABC + A + 1	5	3	0	-	-	60	34720
ABCDE + ABC + DE + A	5	9	0	-	-	30	2222080
ABCDE + ABC + AD + 1	5	7	0	-	-	60	416640
ABCDE + ABC + AD + E	5	7	0	-	-	60	6666240
ABCDE + ABC + AD + D	5	7	0	-	-	120	833280
ABCDE + ABC + AD + B	5	7	0	-	-	120	2499840
ABCDE + ABC + AD + BE	5	11	0	-	-	60	9999360
ABCDE + ABC + AD + A	5	7	0	-	-	650	416640

Representative	D	N	H	Be	Ba	R	O
ABCDE + ABC + ADE + 1	5	7	0	-	-	15	277760
ABCDE + ABC + ADE + B	5	7	0	-	-	60	3333120
ABCDE + ABC + ADE + BD	5	9	0	-	-	60	9999360
ABCDE + ABC + ADE + A	5	7	0	-	-	15	277760
ABCDE + ABC + AB + 1	5	5	0	-	-	90	34720
ABCDE + ABC + AB + D	5	5	0	-	-	210	833280
ABCDE + ABC + AB + DE	5	9	0	-	-	120	2222080
ABCDE + ABC + AB + C	5	5	0	-	-	30	138880
ABCDE + ABC + AB + CD	5	9	0	-	-	90	1666560
ABCDE + ABC + CDE + AB	5	7	0	-	-	60	2222080
ABCDE + ABC + AB + A	5	5	0	-	-	210	104160
ABCDE + ABC + ADE + AB	5	5	0	-	-	210	166656
ABCDE + ABCD + ABE + E	5	5	0	-	-	30	138880
ABCDE + ABCD + ABE + CE	5	9	0	-	-	60	833280
ABCDE + ABCD + ABE + CDE	5	7	0	-	-	15	444416
Thickness 5							
ABC + ADE + BD + C + 1	3	10	0	-	-	840	9999360
ABC + ADE + BD + A + 1	3	10	0	-	-	150	1666560
ABC + CDE + AB + D + 1	3	8	0	-	-	630	3333120
ABC + CDE + AB + A + D	3	8	0	-	Y	1380	9999360
ABC + ABD + ACE + BC + DE	3	12	0	Y	-	240	2666496
ABCD + AB + CD + A + 1	4	10	0	-	-	240	83328
ABCD + ABE + CE + D + 1	4	8	0	-	-	480	26664960
ABCD + ABE + CE + C + E	4	8	0	-	-	60	3333120
ABCD + ABE + CD + E + 1	4	10	0	-	-	120	2222080
ABCD + ABE + CD + C + E	4	10	0	-	Y	240	13332480
ABCD + ABE + CDE + E + 1	4	8	0	-	-	15	4444160
ABCD + ABE + CE + A + 1	4	8	0	-	-	180	19998720
ABCD + ABE + CD + A + 1	4	10	0	-	-	90	6666240
ABCD + ABE + CDE + A + 1	4	8	0	-	-	90	26664960
ABCD + ABE + CDE + A + E	4	8	0	-	-	60	26664960
ABCD + ABE + CDE + A + C	4	8	0	-	Y	180	39997440
ABCD + ABE + AE + CD + B	4	8	0	-	-	60	3333120
ABCD + ABE + CDE + AE + B	4	6	0	-	-	60	6666240
ABCD + ABE + AC + E + 1	4	8	0	-	-	900	4999680
ABCD + ABE + AC + DE + 1	4	10	0	-	-	120	19998720
ABCD + ABE + AC + C + 1	4	8	0	-	-	300	1249920
ABCD + ABE + AC + DE + C	4	10	0	-	-	180	79994880
ABCD + ABE + CDE + AC + E	4	8	0	-	-	60	39997440
ABCD + ABE + AC + DE + B	4	10	0	-	-	120	39997440
ABCD + ABE + AC + BE + E	4	10	0	-	-	480	4999680
ABCD + ABE + AC + BE + D	4	10	0	-	-	180	9999360
ABCD + ABE + AC + BD + 1	4	12	0	-	-	60	2499840

Representative	D	N	H	Be	Ba	R	O
ABCD + ABE + AC + BD + E	4	12	0	-	-	60	3333120
ABCD + ABE + CDE + AC + BD	4	10	0	-	-	30	26664960
ABCD + ABE + AC + DE + A	4	10	0	-	-	720	19998720
ABCD + ABE + ACE + BD + C	4	8	0	-	Y	120	19998720
ABCD + ABE + CDE + AB + 1	4	6	0	-	-	280	444416
ABCD + ABE + CDE + AB + E	4	6	0	-	Y	30	8888320
ABCD + ABE + CDE + AB + C	4	6	0	-	-	980	6666240
ABCD + ABE + CDE + AB + AC	4	10	0	-	-	900	19998720
ABCD + ABC + DE + A + 1	4	8	0	-	-	1620	833280
ABCD + ABC + ADE + E + 1	4	6	0	-	-	630	1666560
ABCD + ABC + ADE + B + 1	4	6	0	-	-	2310	4999680
ABCD + ABC + ADE + B + E	4	6	0	-	Y	1680	9999360
ABCD + ABC + ADE + BE + D	4	8	0	-	Y	120	9999360
ABCD + ABC + ADE + BD + 1	4	10	0	-	-	300	4999680
ABCD + ABC + ADE + BD + E	4	10	0	-	Y	120	9999360
ABCD + ABC + ADE + BD + C	4	10	0	-	-	120	9999360
ABCD + ABC + ADE + BD + CE	4	10	0	-	-	120	39997440
ABCD + ABC + ADE + A + 1	4	6	0	-	-	270	416640
ABCD + ABC + ADE + BD + A	4	10	0	-	-	1020	4999680
ABCD + ABC + ADE + AD + BE	4	8	0	-	-	1740	9999360
ABCD + ABC + AB + DE + 1	4	10	0	-	-	480	833280
ABCD + ABC + AB + DE + C	4	10	0	-	-	540	3333120
ABCD + ABC + CDE + AB + 1	4	8	0	-	-	300	1666560
ABCD + ABC + CDE + AB + E	4	8	0	-	-	360	26664960
ABCD + ABC + CDE + AB + D	4	8	0	-	-	120	3333120
ABCD + ABC + CDE + AB + C	4	8	0	-	Y	330	1666560
ABCD + ABC + AB + DE + A	4	10	0	-	-	2130	2499840
ABCD + ABC + CDE + AB + A	4	8	0	-	-	1740	9999360
ABCD + ABC + ABE + DE + 1	4	8	0	-	-	920	3333120
ABCD + ABC + ABE + DE + C	4	8	0	-	Y	980	53329920
ABCD + ABC + ABE + CDE + E	4	6	0	-	-	360	2666496
ABCD + ABC + ABE + DE + A	4	8	0	-	Y	1380	9999360
ABCD + ABC + ABE + AC + DE	4	10	0	-	-	1500	39997440
ABCD + ABC + ABE + AB + DE	4	8	0	-	-	1200	3333120
ABCD + ABC + ABD + CE + D	4	10	0	-	-	60	3333120
ABCDE + AB + CD + E + 1	5	11	0	-	-	280	444416
ABCDE + AB + CD + A + 1	5	11	0	-	-	240	166656
ABCDE + ABC + DE + D + 1	5	9	0	-	-	120	952320
ABCDE + ABC + DE + A + 1	5	9	0	-	-	60	2222080
ABCDE + ABC + DE + A + D	5	9	0	-	-	420	6666240
ABCDE + ABC + AD + E + 1	5	7	0	-	-	1040	6666240
ABCDE + ABC + AD + D + 1	5	7	0	-	-	180	833280
ABCDE + ABC + AD + B + 1	5	7	0	-	-	1140	2499840
ABCDE + ABC + AD + B + D	5	7	0	-	-	1110	2499840

Representative	D	N	H	Be	Ba	R	O
ABCDE + ABC + AD + BE + 1	5	11	0	-	-	60	9999360
ABCDE + ABC + AD + BE + C	5	11	0	-	-	60	19998720
ABCDE + ABC + AD + A + 1	5	7	0	-	-	650	416640
ABCDE + ABC + AD + BE + A	5	11	0	-	-	960	9999360
ABCDE + ABC + ADE + B + 1	5	7	0	-	-	150	3333120
ABCDE + ABC + ADE + B + D	5	7	0	-	-	60	4999680
ABCDE + ABC + ADE + BD + 1	5	9	0	-	-	60	9999360
ABCDE + ABC + ADE + BD + C	5	9	0	-	-	120	39997440
ABCDE + ABC + ADE + BD + CE	5	9	0	-	-	30	19998720
ABCDE + ABC + ADE + BD + B	5	9	0	-	-	720	9999360
ABCDE + ABC + ADE + A + 1	5	7	0	-	-	15	277760
ABCDE + ABC + ADE + BD + A	5	9	0	-	-	60	9999360
ABCDE + ABC + AB + D + 1	5	5	0	-	-	1020	833280
ABCDE + ABC + AB + DE + 1	5	9	0	-	-	120	2222080
ABCDE + ABC + AB + DE + D	5	9	0	-	-	1230	6666240
ABCDE + ABC + AB + DE + C	5	9	0	-	-	30	8888320
ABCDE + ABC + AB + CD + 1	5	9	0	-	-	90	1666560
ABCDE + ABC + AB + CD + E	5	9	0	-	-	90	6666240
ABCDE + ABC + AB + CD + C	5	9	0	-	-	510	1666560
ABCDE + ABC + CDE + AB + 1	5	7	0	-	-	60	2222080
ABCDE + ABC + CDE + AB + D	5	7	0	-	-	150	13332480
ABCDE + ABC + CDE + AB + C	5	7	0	-	-	30	2222080
ABCDE + ABC + AB + A + 1	5	5	0	-	-	300	104160
ABCDE + ABC + AB + DE + A	5	9	0	-	-	270	6666240
ABCDE + ABC + CDE + AB + A	5	7	0	-	-	180	6666240
ABCDE + ABC + ADE + AB + 1	5	5	0	-	-	210	166656
ABCDE + ABC + ADE + AB + DE	5	9	0	-	-	750	6666240
ABCDE + ABC + ADE + AB + C	5	5	0	-	-	60	1666560
ABCDE + ABC + ADE + AB + B	5	5	0	-	-	660	3333120
ABCDE + ABC + ADE + AB + A	5	5	0	-	-	240	166656
ABCDE + ABC + ABD + CE + D	5	9	0	-	-	60	26664960
ABCDE + ABCD + ABE + CE + 1	5	9	0	-	-	60	833280
ABCDE + ABCD + ABE + CE + D	5	9	0	-	-	60	6666240
ABCDE + ABCD + ABE + CD + E	5	9	0	-	-	30	8888320
ABCDE + ABCD + ABE + CDE + 1	5	7	0	-	-	15	444416
ABCDE + ABCD + ABE + CDE + E	5	7	0	-	-	15	444416
ABCDE + ABCD + ABE + CE + A	5	9	0	-	-	120	2499840
ABCDE + ABCD + ABE + CDE + A	5	7	0	-	-	60	6666240
ABCDE + ABCD + ABE + CDE + AC	5	9	0	-	-	60	6666240
ABCDE + ABCD + ABC + ADE + E	5	5	0	-	-	120	1666560
Thickness 6							
ABC + ABD + ACE + BC + DE + 1	3	12	0	Y	-	240	2666496
ABC + ABD + ACE + BC + DE + A	3	12	0	Y	Y	300	5332992

Representative	D	N	H	Be	Ba	R	O
ABCD + ABE + CDE + A + E + 1	4	8	0	-	-	840	26664960
ABCD + ABE + CDE + A + C + 1	4	8	0	-	Y	1380	39997440
ABCD + ABE + CDE + A + C + E	4	8	0	-	-	120	39997440
ABCD + ABE + AC + DE + C + 1	4	10	0	-	-	420	79994880
ABCD + ABE + AC + DE + B + 1	4	10	0	-	-	1800	39997440
ABCD + ABE + AC + BE + E + 1	4	10	0	-	-	660	4999680
ABCD + ABE + AC + BD + E + 1	4	12	0	-	-	600	3333120
ABCD + ABE + AC + BD + C + D	4	12	0	-	Y	180	1666560
ABCD + ABE + CDE + AC + BD + 1	4	10	0	-	-	30	26664960
ABCD + ABE + CDE + AC + BD + E	4	10	0	-	-	30	39997440
ABCD + ABE + AC + DE + A + 1	4	10	0	-	-	720	19998720
ABCD + ABE + CDE + AC + BE + A	4	10	0	-	-	180	6666240
ABCD + ABE + CDE + AC + AE + B	4	10	0	-	-	360	79994880
ABCD + ABE + CDE + AB + C + E	4	6	0	-	-	540	2666496
ABCD + ABE + CDE + AB + AC + 1	4	10	0	-	-	900	19998720
ABCD + ABE + CDE + AB + AC + E	4	10	0	-	Y	360	79994880
ABCD + ABE + CDE + AB + AC + B	4	10	0	-	-	1500	79994880
ABCD + ABE + ACE + AB + DE + C	4	10	0	-	Y	120	39997440
ABCD + ABC + ADE + BD + C + E	4	10	0	-	Y	1260	19998720
ABCD + ABC + ADE + BD + CE + 1	4	10	0	-	-	120	39997440
ABCD + ABC + ADE + BD + A + 1	4	10	0	-	-	1200	4999680
ABCD + ABC + ADE + BD + CE + A	4	10	0	-	Y	420	39997440
ABCD + ABC + ADE + AD + BE + 1	4	8	0	-	-	2340	9999360
ABCD + ABC + CDE + AB + E + 1	4	8	0	-	-	1110	26664960
ABCD + ABC + CDE + AB + C + 1	4	8	0	-	Y	420	1666560
ABCD + ABC + AB + DE + A + 1	4	10	0	-	-	3330	2499840
ABCD + ABC + CDE + AB + A + 1	4	8	0	-	-	2040	9999360
ABCD + ABC + ABE + AC + DE + 1	4	10	0	-	-	1620	39997440
ABCD + ABC + ABE + CDE + AC + E	4	10	0	-	-	780	39997440
ABCD + ABC + ABE + AC + DE + B	4	10	0	-	Y	1740	79994880
ABCD + ABCE + ABD + CDE + AD + BE	4	10	0	-	-	180	2666496
ABCDE + ABC + DE + A + D + 1	5	9	0	-	-	1170	6666240
ABCDE + ABC + AD + B + D + 1	5	7	0	-	-	2490	2499840
ABCDE + ABC + AD + BE + D + E	5	11	0	-	-	300	6666240
ABCDE + ABC + AD + BE + C + 1	5	11	0	-	-	1290	19998720
ABCDE + ABC + AD + BE + A + 1	5	11	0	-	-	1080	9999360
ABCDE + ABC + ADE + B + D + 1	5	7	0	-	-	900	4999680
ABCDE + ABC + ADE + BD + C + 1	5	9	0	-	-	540	39997440
ABCDE + ABC + ADE + BD + C + E	5	9	0	-	-	300	39997440
ABCDE + ABC + ADE + BD + CE + 1	5	9	0	-	-	30	19998720
ABCDE + ABC + ADE + BD + B + 1	5	9	0	-	-	720	9999360
ABCDE + ABC + ADE + BD + B + E	5	9	0	-	-	960	39997440
ABCDE + ABC + ADE + BD + CE + B	5	9	0	-	-	360	79994880
ABCDE + ABC + ADE + BD + A + 1	5	9	0	-	-	120	9999360

Representative	D	N	H	Be	Ba	R	O
ABCDE + ABC + ADE + BD + CE + A	5	9	0	-	-	30	19998720
ABCDE + ABC + ADE + BD + A + B	5	9	0	-	-	1020	9999360
ABCDE + ABC + AB + DE + D + 1	5	9	0	-	-	1350	6666240
ABCDE + ABC + AB + DE + C + D	5	9	0	-	-	1020	26664960
ABCDE + ABC + AB + CD + C + 1	5	9	0	-	-	690	1666560
ABCDE + ABC + CDE + AB + D + 1	5	7	0	-	-	300	13332480
ABCDE + ABC + CDE + AB + C + 1	5	7	0	-	-	90	2222080
ABCDE + ABC + CDE + AB + DE + C	5	7	0	-	-	30	444416
ABCDE + ABC + AB + DE + A + 1	5	9	0	-	-	390	6666240
ABCDE + ABC + AB + DE + A + D	5	9	0	-	-	4530	19998720
ABCDE + ABC + CDE + AB + A + 1	5	7	0	-	-	240	6666240
ABCDE + ABC + CDE + AB + A + D	5	7	0	-	-	420	39997440
ABCDE + ABC + CDE + AB + A + C	5	7	0	-	-	900	6666240
ABCDE + ABC + CDE + AB + AD + E	5	9	0	-	-	660	79994880
ABCDE + ABC + CDE + AB + AD + B	5	9	0	-	-	300	39997440
ABCDE + ABC + ADE + AB + DE + 1	5	9	0	-	-	750	6666240
ABCDE + ABC + ADE + AB + DE + D	5	9	0	-	-	3060	19998720
ABCDE + ABC + ADE + AB + DE + C	5	9	0	-	-	90	26664960
ABCDE + ABC + ADE + AB + B + 1	5	5	0	-	-	1920	3333120
ABCDE + ABC + ADE + AB + DE + B	5	9	0	-	-	1140	13332480
ABCDE + ABC + ADE + AB + A + 1	5	5	0	-	-	240	166656
ABCDE + ABC + ADE + AB + DE + A	5	9	0	-	-	480	6666240
ABCDE + ABC + ADE + AB + AD + CE	5	11	0	-	-	180	6666240
ABCDE + ABC + ABD + CDE + CE + D	5	7	0	-	-	180	6666240
ABCDE + ABC + ABD + ACE + BC + DE	5	11	0	-	-	240	31997952
ABCDE + ABCD + ABE + CE + A + 1	5	9	0	-	-	180	2499840
ABCDE + ABCD + ABE + CDE + A + 1	5	7	0	-	-	90	6666240
ABCDE + ABCD + ABE + CDE + A + C	5	7	0	-	-	60	6666240
ABCDE + ABCD + ABE + AC + DE + C	5	11	0	-	-	180	6666240
ABCDE + ABCD + ABE + CDE + AC + 1	5	9	0	-	-	60	6666240
ABCDE + ABCD + ABE + CDE + AC + E	5	9	0	-	-	60	6666240
ABCDE + ABCD + ABE + CDE + AC + B	5	9	0	-	-	120	79994880
ABCDE + ABCD + ABC + ADE + BE + C	5	9	0	-	-	240	39997440
ABCDE + ABCD + ABC + CDE + AB + E	5	9	0	-	-	150	26664960
ABCDE + ABCD + ABC + ADE + AB + CE	5	11	0	-	-	270	6666240
Thickness 7							
ABCD + ABE + CDE + AC + BD + E + 1	4	10	0	-	-	1170	39997440
ABCD + ABE + CDE + AC + BD + A + B	4	10	0	-	Y	720	31997952
ABCD + ABE + CDE + AB + AC + C + E	4	10	0	-	-	1740	39997440
ABCD + ABE + CDE + AB + AC + BD + E	4	10	0	-	Y	900	53329920
ABCD + ABE + CDE + AB + AC + A + C	4	10	0	-	-	1560	6666240
ABCD + ABCE + ABD + CDE + AD + BE + 1	4	10	0	-	-	180	2666496
ABCDE + ABC + AD + BE + D + E + 1	5	11	0	-	-	1620	6666240

Representative	D	N	H	Be	Ba	R	O
ABCDE + ABC + ADE + BD + B + E + 1	5	9	0	-	-	4080	39997440
ABCDE + ABC + ADE + BD + A + B + 1	5	9	0	-	-	1800	9999360
ABCDE + ABC + AB + DE + A + D + 1	5	9	0	-	-	8430	19998720
ABCDE + ABC + CDE + AB + A + D + 1	5	7	0	-	-	7980	39997440
ABCDE + ABC + CDE + AB + A + C + 1	5	7	0	-	-	1620	6666240
ABCDE + ABC + CDE + AB + AD + DE + C	5	9	0	-	-	240	6666240
ABCDE + ABC + ADE + AB + DE + D + 1	5	9	0	-	-	3630	19998720
ABCDE + ABC + ADE + AB + DE + C + D	5	9	0	-	-	4110	79994880
ABCDE + ABC + ADE + AB + DE + B + 1	5	9	0	-	-	1650	13332480
ABCDE + ABC + ADE + AB + DE + B + D	5	9	0	-	-	4380	39997440
ABCDE + ABC + ADE + AB + DE + A + 1	5	9	0	-	-	960	6666240
ABCDE + ABC + ADE + AB + DE + A + D	5	9	0	-	-	9030	19998720
ABCDE + ABC + ADE + AB + AD + CE + 1	5	11	0	-	-	210	6666240
ABCDE + ABC + ADE + AB + AD + CE + B	5	11	0	-	-	360	19998720
ABCDE + ABC + ABD + CDE + AC + CE + D	5	9	0	-	-	810	19998720
ABCDE + ABC + ABD + CDE + AC + DE + B	5	9	0	-	-	1320	79994880
ABCDE + ABC + ABD + CDE + AC + BE + D	5	11	0	-	-	120	53329920
ABCDE + ABC + ABD + ACE + BC + DE + 1	5	11	0	-	-	240	31997952
ABCDE + ABC + ABD + ACE + BC + DE + B	5	11	0	-	-	3780	53329920
ABCDE + ABC + ABD + ACE + BC + DE + A	5	11	0	-	-	180	31997952
ABCDE + ABCD + ABE + AC + DE + C + 1	5	11	0	-	-	360	6666240
ABCDE + ABCD + ABE + CDE + AC + B + 1	5	9	0	-	-	360	79994880
ABCDE + ABCD + ABC + ADE + AB + CE + 1	5	11	0	-	-	270	6666240
ABCDE + ABCD + ABC + ADE + AB + CE + D	5	11	0	-	-	120	19998720
ABCDE + ABCD + ABC + ADE + AB + CE + B	5	11	0	-	-	420	19998720
ABCDE + ABCD + ABCE + ADE + AB + CD + E	5	11	0	-	-	120	19998720
Thickness 8							
ABCDE + ABC + ADE + AB + DE + A + D + 1	5	9	0	-	-	12960	19998720
ABCDE + ABC + ABD + ACE + BC + DE + B + 1	5	11	0	-	-	4260	53329920
ABCDE + ABC + ABD + ACE + BC + DE + A + 1	5	11	0	-	-	2160	31997952
ABCDE + ABC + ABD + ACE + BC + DE + A + B	5	11	0	-	-	6060	53329920

Functions	4294967296
Homogeneous functions	2111
Rigid functions	211259
Balanced functions	601080390
Semi-bent functions	27387136
Orbits / Representatives	382
Semi-bent orbits	14
Balanced orbits	38