Measuring and Mapping the Sensitivity of Rotation Symmetric Boolean Functions

Sindre Tornes Steinsvik

Master's thesis in Software Engineering at

Department of Computing, Mathematics and Physics,
Bergen University College

Department of Informatics,
University of Bergen

December 2021

Western Norway
University of
Applied Sciences

Acknowledgements

I would like to acknowledge and thank both my supervisors Pål Ellingsen and Constanza Susana Riera for the help throughout this master thesis. I would also like to thank family and friends for their support, and a special appreciation to my girlfriend for allowing me to rant about this project to a certain extent.

Abstract

This thesis investigate the sensitivity of rotational symmetric Boolean functions up to five variables, this means the difference in output created by complementing one of the input entries. The programs designed for creating Boolean functions, filtering out rotational symmetric Boolean functions and testing for sensitivity are all explained in the thesis. The complete analysis of the result are shown in tables, and the functions with sensitivity close to $\frac{1}{2}$ are listed in its short algebraic normal form (SANF).

Contents

List of Figures

List of Tables

Chapter 1

Introduction

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). Today Boolean functions are relevant and central in several cryptographic algorithms, both in stream and block ciphers – represented by combinations of LFSRs and substitution boxes (S-boxes, cf. AES, DES, and more). Some fundamental concepts in achieving secure communication today are confusion and diffusion. It has been shown that Boolean functions provide both confusion and diffusion. Confusion is achieved by the complexity of the related functions, which can be described by several properties, but the two most important are: the algebraic degree and the nonlinearity of said functions. These criteria describe, in their own way, the difference between any given function and the set of affine functions – that is, linear functions with or without a constant – as Claude Carlet phrased it, in [1]. 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. Through this thesis, an additional known complexity criteria is going to be discussed, the concept of *sensitivity*.

1.1 Research question and expected results

Research question

- The goal of this master thesis is to measure and map the sensitivity of rotation symmetric Boolean functions for highest possible values of n.

Expected Results

The obtained results from this master thesis will shed some light on the above research question. The searching space for the different values of n when talking about Boolean functions is 2^{2^n}. For $n = 2$ the search space is 16. The first expected result will be to find all rotation symmetric function in $n = 2$, then map and measure the sensitivity of these functions. Next will be to increase n as much as possible and try to find a good method to find the rotation symmetric functions and check the functions' sensitivity.

For $n = 4$, the space has size 2^6; for $n = 5$, the space has size 2^8; for $n = 6$, the space has size 2^{14}; for $n = 7$ the space has size 2^{20}.

1.2 Structure of thesis

A summary of the thesis structure.

Chapter 1 - Introduction:

Introduces the circumstances of the research and questions.

Chapter 2 - Background:

Describes the context of the research and the fundamental mathematics used to accomplish it.

Chapter 3 - Calculating sensitivity on rotation symmetric functions for n = 2,3,4:

Describing the method, programs and code used to calculating Rots for $n = 2, 3, 4$

Chapter 4 - Calculating sensitivity on rotation symmetric functions for n = 5:

Describing the method, programs and code used to calculating Rots for $n = 5$.

Chapter 5 - Results and Discussion:

Results and analysis of $n \leq 5$.

Chapter 6 - Conclusion:

Conclusion of the work, its contribution and future work.

Chapter 2

Background

This chapter shows an overview of some of the mathematical concepts used in this thesis. In particular, rotation symmetric Boolean functions, and sensitivity for Boolean functions are two key factors in this thesis and will be explained in this section.

2.1 Mathematical Foundations

2.1.1 Set theory

Set theory is widely used for providing a language for describing concepts in both mathematics and computer science. A *set* – in simple terms – is a collection of objects of any kind, which are referred to as the *elements* of the set; e.g. $A = \{1, 2, 3\}$ is a set referenced to set A. Set A contains the numbers 1, 2 and 3. Although sets can contain elements of all sorts of characters, most sets used in this thesis will be comprised of numbers, or other mathematical elements. Elements in a set are separated with commas, and the elements are contained within braces { }.

The order of the elements is irrelevant. If we define $B = \{2, 1, 3\}$, B contains numbers 2, 1 and 3, and therefore sets A and B contains exactly the same values. Therefore they are the same set (i.e. $A = B$).

Claiming that an object is contained within a set is one of the fundamental statements in set theory, and is represented by $\in$. For set A used above, $2 \in A$

is true, but $4 \in A$ is not true. If an element is not in the set, this is represented by $\notin$; i.e. $4 \notin A$.

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. $\pi, \sqrt{2}$, etc.). Note that some texts may include 0 in $\mathbb{N}$.

Two sets can be merged into a third set by the use of different operators. The union $\cup$ operator, also called the *or* operator, forms a new set that contains all elements in the first set and all the elements in the second set. The intersection operator $\cap$, also known as the *and* operator, combines two sets to create a set with all elements included in both sets.

If we have two sets A and B, and all elements of A are in B, then A is a *subset* of B. This is written as $A \subseteq B$. Note that, if $A \subseteq B$ and $B \subseteq A$, then $A = B$. If not all elements of A are in B, then we write $A \nsubseteq B$. Given two sets A and B, if A is not equal to B, we denote this by $A \neq B$.

Example 2.2. Let a, b, c be elements of some set S, and let $A = \{x, y\}$, $B = \{y, z\}$, and $C = \{z\}$ be subsets of S. Then,

$$\begin{aligned} A \cup B &= \{x, y, z\}, \\ A \cap B &= \{y\}, \\ A \times C &= \{(x, z), (y, z)\}. \end{aligned}$$

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

An *ordered pair* is an object of the form (a, b), where a is an element of a set A and b is an element of a set B. The set of all ordered pairs of this form is called the *Cartesian product* of two sets A and B and is denoted by $A \times B$. Therefore, $A \times B = (a, b) : a \in A$ and $b \in B$. This operation is particularly important since it lead to the concept of *relations* and *functions*, that play a

important role in computer science. A *binary relation* between A and B is an association between elements $a \in A$ and $b \in B$, and is given by a subset of $A \times B$. We have then that a is related to b if and only if the pair (a, b) is in the set that defines the relation.

Definition 2.1. (Function) [2]
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 the set that defines the relation.

A function f from a set A to a set B is denoted 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, the range is denoted by $f(A)$, defined $f(A) = \{f(x) \mid x \in A\}$.

There are some properties of functions that are important to note. A function $f : A \to B$ is called *injective* (or *one-to-one*) if, for $a_1, a_2 \in A$, $f(a_1) = f(a_2)$ 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 $(g \circ f)(x) = g(f(x))$ [2].

2.1.2 Boolean algebra

The simplest Boolean algebra consist of the set $B = \{0, 1\}$ together with the operations of *disjunction* ($\vee$), *conjunction* ($\wedge$) and *negation* ($\neg$). The effect of

the operations ($\vee$) and ($\wedge$) on the symbols 0 and 1 is given in Table 2.1, which represents the *truth table* of these operations (which shows a column for every input element, and a column for the output of each operation).

p	q	$\neg p$	$p \vee q$	$p \wedge q$
0	0	1	0	0
0	1	1	1	0
1	0	0	1	0
1	1	0	1	1

Table 2.1: Examples operations *disjunction* ($\vee$), *conjunction* ($\wedge$) and *negation* ($\neg$).

The process of negation is defined by $\neg 0 = 1$ and $\neg 1 = 0$. If p and q are two Boolean variables, in other words each of p and q can take the value 0 or 1, then the Table shown in 2.1 can be constructed [2].Truth tables can be used to establish the following equivalences which are called the laws of Boolean algebra:

Commutative laws

$p \wedge q = q \wedge p$

$p \vee q = q \vee p$

Associative laws

$p \wedge (q \wedge r) = (p \wedge q) \wedge r$

$p \vee (q \vee r) = (p \vee q) \vee r$

Distributive laws

$p \wedge (q \vee r) = (p \wedge q) \vee (p \wedge q)$

$p \vee (q \wedge r) = (p \vee q) \wedge (p \vee q)$

Idempotent laws

$p \wedge p = p$

$p \vee p = p$

Absorption laws

$p \wedge (p \vee q) = p$

$p \vee (p \wedge q) = p$

De Morgan's laws

$\neg(p \wedge q) = \neg p \vee \neg q$

$\neg(p \vee q) = \neg p \wedge \neg q$

Minterm in Boolean algebra is a product term, in which each variable appears once. Minterms can be used to express any Boolean functions of Boolean variables uniquely as a a disjunction of minterms. This is called the *disjunctive normal form* of the Boolean expression. Consider the Boolean expression $f(p, q, r)$ whose truth table is given in Table 2.2. The ones in this table correspond to the three minterms: $\neg p \wedge \neg q \wedge r$, $\neg p \wedge q \wedge r$ and $p \wedge \neg q \wedge \neg r$. The truth table of f can be obtained by overlaying the truth table for these three minterms.

p	q	r	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

Table 2.2: Example Boolean expression

Since disjunction with 1 has the effect of overwriting any zero, f is the disjunction of these three miniterms, and so $f(p, q, r) = \neg p \wedge \neg q \wedge r \vee \neg p \wedge q \wedge r \vee p \wedge \neg q \wedge \neg r$. This is the disjunctive normal form of f. Clearly, the same reasoning can be applied to a Boolean expression of any number of variables. We have seen that every Boolean function has a unique representation as a disjunction of miniterms. Hence any Boolean function can be represented in terms of the operators $\vee$, $\wedge$ and $\neg$. [2]

2.2 Boolean functions

Boolean functions are named after George Boole (1815-1864), who laid the foundation for what is now called *Boolean Algebra* [3], and are usable in a range of fields not only in mathematics, cf. *logic gates* in Electrical Engineering. This thesis revolves around the complexity of Boolean functions as cryptographic tools, the following section serves to define such functions.

Let $\mathbb{V}_n$ be the vector space of dimension n over the two-element field $\mathbb{F}_2$. For two vectors in $\mathbb{V}_n$, say $a = (a_1, ..., a_n)$ and $b = (b_1, ..., b_n)$, we define the scalar product $a \cdot b = a_1 b_1 \oplus ... \oplus a_n b_n$, where the multiplication and addition $\oplus$ (called xor) are over $\mathbb{F}_2$ (it should not be confused with the direct product of vector spaces, but that will be clear from context). We also define the operation $\star$ by a $\star$ $b = (a_1 b_1, ..., a_n b_n)$.[3]

Definition 2.2. Boolean functions [3]
A Boolean function f in n variables is a map from $\mathbb{V}_n$ to $\mathbb{F}_2$. The $(0,1)-$sequence defined by $(f(v_0), f(v_1), ..., f(v_{2^n-1}))$ is called the truth table of f , where $v_0 = (0, ..., 0, 0), v_1 = (0, ..., 0, 1), ..., v_{2^n-1} = (1, ..., 1, 1)$, written in lexicographical order. The $(1,-1)-$sequence of f (or simply sequence) is defined by $((-1)^{f(v_0)}, ..., (-1)^{f(v_{2^n-1})})$. The algebra of all Boolean functions on $\mathbb{V}_n$ will be denoted by $\mathcal{B}_n$.

Definition 2.3. Algebraic Normal Form [4]
Among the most common representations of Boolean functions is the Algebraic Normal Form (in brief the ANF), most usually used in cryptography and coding. ANF is an n-variable polynomial representation over $\mathbb{F}_2$, of the form:

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

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

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

$$\ell_{u,c}(x) = u \cdot x \oplus c = u_1 x_1 \oplus \cdots \oplus u_n x_n \oplus c, \tag{2.1}$$

where $u = (u_1, \ldots, u_n) \in \mathbb{F}_2^n$ and $c \in \mathbb{F}_2$. If $c = 0$, such that $\ell_{u,0}$ only consists of monomials of algebraic degree 1, and no constant, then it is a *linear* function [3].

Every Boolean function f has a unique representation in its algebraic normal form. Most Boolean functions discussed in this thesis will be presented in its corresponding ANF. For the particular set of Boolean functions known as rotation symmetric Boolean functions (Rots), defined in Section 2.3, the functions will be sometimes shown and discussed in their short algebraic normal form (SANF), explained also in Section 2.3.

Definition 2.4. Affine Transformation [5]
An *affine transformation* $T : \mathbb{F}_2^n \to \mathbb{F}_2^n$ is a transformation of the form $T(x) = Ax + b$, with A an $n \times n$ matrix, and b in $\mathbb{F}_2^n$.

By matrix multiplication, the affine transformations map each x_i in $= (x_1, \ldots, x_n)$ 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.

When discussing affine transformations, only matrices that are invertible are included, meaning no information is lost in the transformation.

2.3 Rotation symmetric Boolean functions

In this section, we introduce a crucial concept for this thesis, the concept of rotation symmetric Boolean functions.

Let $\mathbb{V}_n (= \mathbb{F}_2^n)$ be the vector space of dimension n over two-element field $\mathbb{F}_2$. Let $x_i \in (0, 1)$ for $1 \leq i \leq n$. For $1 \leq k \leq n$, we define

$P_n^k(x_i) = x_{i+k}$ if $i + k \leq n, = x_{i+k-n}$ if $i + k < n$.

Let $(x_1, x_2, ..., x_{n-1}, x_n) \in \mathbb{V}_n$. Then we extend the definition as

$P_n^k(x_1, x_2, ..., x_{n-1}, x_n) = (P_n^k(x_1), P_n^k(x_2), ..., P_n^k(x_{n-1}), P_n^k(x_n))$.

A Boolean function on n variables may be viewed as a mapping from $\mathbb{V}_n$ into $\mathbb{V}_1$. A Boolean function $f(x_1, ..., x_n)$ can be explained as the

output column of its *truth table,* i.e., a binary string of length 2^n, $f = [f(0,0,...,0), f(1,0,...,0), f(0,1,...,0)f(1,1,...,1)]$. The table below shows a truth table of 3-variable Boolean functions.

Definition 2.5. Rotation Symmetric Boolean functions [6]
A Boolean function f is *Rotation symmetric (RotS)* if and only if for any $(x_1,...,x_n) \in \mathbb{V}_n$

$f(P_n^k(x_1,...,x_n)) = f(x_1,...,x_n)$ for any $1 \leq k \leq n$.

x_3	x_2	x_1	f	No.	x_3	x_2	x_1	f
0	0	0	1	1	0	0	0	0
0	0	1	0	2	0	0	1	0
0	1	0	0	2	0	1	0	0
0	1	1	0	3	0	1	1	1
1	0	0	1	2	1	0	0	0
1	0	1	1	3	1	0	1	1
1	1	0	0	3	1	1	0	1
1	1	1	0	4	1	1	1	0

Table 2.3: Truth table of Boolean functions

For a Boolean function there are 2^n different input values. From the definition above, a functions that possesses the same value for each of the subsets generated from the rotational symmetry is a RotS function. For $n = 3$ the subsets that needs to yield the same output are:
$\{(0,0,0)\}$
$\{(0,0,1),(0,1,0),(1,0,0)\}$
$\{(0,1,1),(1,0,1),(1,1,0)\}$
$\{(1,1,1)\}$
There are four different subsets which partition the 8 input patterns, and any 3-

variable RotS Boolean function can have a specific value corresponding to each subset. Thus there are $2^4 = 16$ rotation symmetric functions on 3 variables. In Table 2.3, the left one is a function which is not RotS, the right one is a RotS function (each different subset is numbered). Information, formulas and definitions in this section are taken from [6].

2.3.1 Short algebraic normal form (SANF)

Short algebraic normal form will be used later in this thesis for making a more compressed list of sensitivity for RotS functions.

Definition 2.6. SANF [3]
A rotation symmetric function $f(x_1, ..., x_n)$ can be written as

$$a_0 \oplus a_1 x_1 \oplus \bigoplus_{j=1}^{n} a_{1j} x_1 x_j \oplus ... \oplus a_{12...n} x_1 x_2 ... x_n,$$

where the coefficients $a_0, a_1, a_{1j}, ..., a_{12...n} \in \mathbb{F}_2$, and the existence of a representative term $x_1 x_{i2} ... x_{i\ell}$ implies the existence of all the terms from $G_n(x_1 x_{i2} ... x_{il})$ in the algebraic normal form. This representation of f is called the short algebraic normal form (SANF) of f.
As an example, consider the ANF of a 4-variable rotation symmetric Boolean function $x_1 \oplus x_2 \oplus x_3 \oplus x_4 \oplus x_1x_2x_3 \oplus x_2x_3x_4 \oplus x_3x_4x_1 \oplus x_4x_1x_2$. Its SANF is $x_1 \oplus x_1x_2x_3$.

2.4 Sensitivity of Boolean functions

The sensitivity set of a Boolean function at a particular input is the set of input positions where changing that one bit changes the output. The sensitivity of the Boolean function at a particular input is then the cardinality of the sensitivity set, while the sensitivity of the function is defined as the maximum of its sensitivity over all possible inputs.

Definition 2.7. (Sensitivity of Boolean functions) [7]

Let $f : \{0,1\}^n \to \{0,1\}$ be a Boolean function, where n is the input

dimension of f. We define the sensitivity of f at x by

$$s(f, \vec{x}) = |\{i \in \{1, \ldots, n\} / f(\vec{x} \oplus \vec{e_i}) \oplus f(\vec{x}) = 1\}|,$$

where $\vec{e_i}$ i the standard unity vector with 1 in position i. The sensitivity of f at a particular input x is then the number of input positions where changing that one bit changes the output.

Tables 2.4 below shows the method used in this thesis for calculating sensitivity for RotS functions. In the example below function f given by its truth table 0001, which is in the set for rots functions for $n = 2$, is tested for sensitivity. Here $\vec{x} = (x_1, x_2)$. The full set of RotS functions for $n = 2$ and their sensitivity can be found in the Appendix A.

$$f(\vec{x}) = (0001)$$
$$\vec{e}_1 = (1, 0)$$
$$\vec{e}_2 = (0, 1)$$

x_1	x_2	$f(\vec{x})$	$\vec{x} \oplus \vec{e}_1$	$f(\vec{x} \oplus \vec{e}_1)$	$f(\vec{x}) \oplus f(\vec{x} \oplus \vec{e}_1)$
0	0	0	1 0	0	0
0	1	0	1 1	1	1
1	0	0	0 0	0	0
1	1	1	0 1	0	1

x_1	x_2	$f(\vec{x})$	$\vec{x} \oplus \vec{e}_2$	$f(\vec{x} \oplus \vec{e}_2)$	$f(\vec{x}) \oplus f(\vec{x} \oplus \vec{e}_2)$
0	0	0	0 1	0	0
0	1	0	0 0	0	0
1	0	0	1 1	1	1
1	1	1	1 0	0	1

$$\vec{x} = (0, 0) : S(f(0, 0)) = 0$$
$$\vec{x} = (0, 1) : S(f(0, 1)) = 1$$
$$\vec{x} = (1, 0) : S(f(1, 0)) = 1$$
$$\vec{x} = (1, 1) : S(f(1, 1)) = 2$$

Table 2.4: Sensitivity for $f = 0001$

2.5 Methodology

How to measure and map the sensitivity of rotation symmetric functions for different values of n?

This project is a quantitative simulation case study, and concerns analysis and calculations of mathematical concepts. The goal of this thesis is measuring and mapping the sensitivity of rotation symmetric functions for the highest possible value of n, by using mathematics and code. The method used for achieving this goal is to separate the programming parts into three different blocks and working iteratively with these blocks. The first block concerns the method for creating the Boolean functions for the different values of n. These programs are shown in Sections 3.1 and 4.1. The second block covers the way to filter out the RotS functions from the list of Boolean functions created in the first block. The program used for this is shown in Sections 3.2 and 4.1. The last block is to use the result from block two (the RotS functions) and testing the functions for sensitivity; these programs can be seen in Sections 3.3 and 4.2.

By making and following these blocks the goal is also separated into three sub goals, something that made the main goal easier to achieve.

2.5.1 Coding setup

The coding, development, and testing of these technologies is done in Python with some help from other frameworks and softwares. Following is a shortlist of the most important libraries and tools used in these experiments:

- Programming language Python 3.8
- SageMath 9.0
- Essential Libraries: numpy, BooleanFunctions, pbori and itertools
- Workspace: CoCalc and SageMath Notebook

The main reason these technologies is selected is because of SageMath 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* and *pbori* -package. The pbori-package

is used for creating ANF. These packages have been essential for working with Boolean functions in this thesis. SageMath [8] is an open-source mathematics software system, which builds on several existing open-source packages, and runs on Python. So that is why Python are selected as programming language for this thesis. For this thesis two workplaces are used. Cocalc is a online virtual workspace, which is used for collaborating with my supervisors regarding the code. Cocalc is not suitable for heavy calculations, such as creating Boolean functions for $n = 5$, because of Cocalc‘s lack of memory. SageMath Notebook is a offline workspace and the memory and CPU of the computer it runs on is used. Sagemath Notebook is used for the more heavy calculations in this project. Python's Itertool is a module that provides various functions that work on iterators to produce complex iterators. This module works as a fast, memory-efficient tool that is used either by themselves or in combination to form iterator algebra [9].

2.6 Related work

In, [10] written in 1999, Pieprzyk and Qu studied some functions, which they called rotation symmetric (RotS) as components in the rounds of a hashing algorithm. This is a desirable feature when effective evaluation of the function is important, for example in the implementation of MD4, MD5 or HAVAL, since one can reuse evaluations from previous iterations. In 2008 Pantelimon Stănică and Subhamoy Maitra wrote a paper on rotation symmetric Boolean functions [6]. There they provided a complete enumeration result for these functions including the number of such functions with specific degree. Pantelimon Stănică and Subhamoy Maitra studied the rotation symmetric bent functions completely up to 8 variables. Further, they observed that up to 10 variables, and found out there is no homogeneous rotation symmetric bent function of degree > 2. They also checked the cryptographic properties of rotation symmetric functions up to 7 variables. This paper [6] have helped a lot to get a good understanding of rotation symmetric Boolean functions.

Chapter 3

Calculating sensitivity on RotS Boolean functions for n = 2,3,4

In this chapter, the program used for calculating sensitivity on rotation symmetric functions for $n = 2, 3, 4$ is explained and shown in Figures 3.1-3.5. The Chapter is divided into three sections: Constructing all Boolean functions for $n = 2, 3, 4$, filtering out rotation symmetric Boolean functions and calculating sensitivity on rotation symmetric Boolean functions.

```
def generate_lower_functions(n_var, R, x):
    temp_functions = [R(0), R(1), x[0], x[0] + 1]

    for i in range(2, n_var+2):
        new_x = x[i-1]
        new_functions = []
        add_function = new_functions.append
        for f1 in temp_functions:
            for f2 in temp_functions:
                add_function(f1*new_x +f2)
        temp_functions = new_functions

    return temp_functions

n = 2
ring = BooleanPolynomialRing(n, "x")
x = ring.gens()
x_map = Arrangements(x, n)

functions = generate_lower_functions(n-1, ring, x)
new_x = x[n-1]
```

Figure 3.1: Generating functions for $n = 2, 3, 4$

3.1 Constructing all Boolean functions

Method *generate_lower_functions* shown in Fig 3.1 has been used to generate all Boolean functions for $n \leq 4$ in this project. For $n = 5$ the method was modified (see next section). The method in fig 3.1 uses three inputs, n choosing an integer for numbers of variables, *ring* setting up the Boolean polynomial ring with the built-in concept of SageMath and last the x a generated list of $x1, x2, ..$ The method generates all possible Boolean functions in algebraic normal form for n variables, building on 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 $\mathcal{B}_0 = \{0, 1\}$.

The result will be a variable containing a list of every Boolean function in algebraic normal form for the given n. Fig 3.2 shows what the variable *functions* contains for $n = 3$. The variable *functions* is what the program in Fig 3.1 returns as a result.

```
functions
[0,
 1,
 x0,
 x0 + 1,
 x1,
 x1 + 1,
 x0 + x1,
 x0 + x1 + 1,
 x0*x1,
 x0*x1 + 1,
 x0*x1 + x0,
 x0*x1 + x0 + 1,
 x0*x1 + x1,
 x0*x1 + x1 + 1,
 x0*x1 + x0 + x1,
 x0*x1 + x0 + x1 + 1
```

Figure 3.2: Result from generating functions $n = 2, 3, 4$

3.2 Finding rotation symmetric Boolean functions

The method used for filtering out the rotation symmetric Boolean function is based on testing every Boolean function with the method shown below. Fig 3.3 shows the test for $n = 3$.

```
def generate_rots(f):
    rots_functions = []
    for i in range(len(f)):
        s1=f[i](0,0,1)
        s2=f[i](1,0,0)
        s3=f[i](0,1,0)
        b1=f[i](0,1,1)
        b2=f[i](1,0,1)
        b3=f[i](1,1,0)
        if((s1==s2==s3)&(b1==b2==b3)):
            rots_functions.append(f[i])
    return rots_functions
```

Figure 3.3: Filtering out RotS Boolean functions for $n = 2, 3, 4$

The method shown in Fig 3.3 takes a list variable in this case a list of Boolean functions created from the method in 3.1, and tests every Boolean function to find and filter out the rotation symmetric Boolean functions. The list variable is shown in Fig 3.2

This is done by testing if the function f possesses the same value corresponding to each of the subsets generated from the rotational symmetry. The method will filter out the functions of interest by checking which value function f got in every entry in its truth table. Boolean functions have 2^n entries in their truth table. Fig 3.3 shows the test for $n = 3$, where the entries will be $2^3 = 8$. However, it is only necessary to test for 6 entries, because the entries $(0, 0, 0)$ and $(1, 1, 1)$ are two unique subsets and will always have the same corresponding value. The other 6 entries are tested. Entry $(0, 0, 1)$ is stored in `s1`, entry $(1, 0, 0)$ is stored in `s2`, entry $(0, 1, 0)$ is stored in `s3`, entry $(0, 1, 1)$ is stored in `b1`, entry $(1, 0, 1)$ is stored in `b2` and entry $(1, 1, 0)$ is stored in `b3`. Then the *if* sentence tests if `s1`, `s2` and `s3` contains the same value and if `b1`, `b2` and `b3` contains the same value. If one functions passes this test the functions will be added to the list variable *rots_functions*. The method returns a list of rotation symmetric Boolean functions.

3.3 Calculating sensitivity on RotS Boolean functions

The method for calculating the sensitivity takes a list of RotS functions and prints out all the functions with their sensitivity. This is shown in Fig 3.4.

```
def sens(f):
    x=[(0,0,0),(0,0,1),(0,1,0),(0,1,1),(1,0,0),(1,0,1),(1,1,0)
    ,(1,1,1)]
    x_pluss_e1=[(1,0,0),(1,0,1),(1,1,0),(1,1,1),(0,0,0),(0,0,1)
    ,(0,1,0),(0,1,1)]
    x_pluss_e2=[(0,1,0),(0,1,1),(0,0,0),(0,0,1),(1,1,0),(1,1,1)
    ,(1,0,0),(1,0,1)]
    x_pluss_e3=[(0,0,1),(0,0,0),(0,1,1),(0,1,0),(1,0,1),(1,0,0)
    ,(1,1,1),(1,1,0)]


    sum_list = []
    for i in range(len(f)):
        s1=[]
        s2=[]
        s3=[]
        for j in range(len(x_pluss_e1)):
            s1.append(f[i](*x_pluss_e1[j])+f[i](*x[j]))
            s2.append(f[i](*x_pluss_e2[j])+f[i](*x[j]))
            s3.append(f[i](*x_pluss_e3[j])+f[i](*x[j]))


        integer_map1 = map(int, s1)
        integer_map2 = map(int, s2)
        integer_map3 = map(int, s3)

        l1 = list(integer_map1)
        l2 = list(integer_map2)
        l3 = list(integer_map3)

        sum_list.append([a + b + c for a, b, c in zip(l1, l2, l3)])

    for i in range(len(f)):
        print('For function (' + str(f[i]) + ') the sensitivity is:
    ')
        for j in range(len(x)):
            print('x = (' + str(x[j]) +') : S(f('+ str(x[j]) +')) =
     ' + str(sum_list[i][j]))
```

Figure 3.4: Testing for sensitivity for $n = 2, 3, 4$

The sens method for calculating $n = 2, 3, 4$ uses a lot of hard coded values like the x list, `x_pluss_e1`, `x_pluss_e2` and `x_pluss_e3`. For $n = 5$, this will be done automatically, because of the length of the values. When it comes to calculating the sensitivity for each function, a double "for loop" is used to calculate $f(\vec{x}) \oplus f(\vec{x} \oplus \vec{e}_1)$, $f(\vec{x}) \oplus f(\vec{x} \oplus \vec{e}_2)$ and $f(\vec{x}) \oplus f(\vec{x} \oplus \vec{e}_3)$ for each entry the functions have in its corresponding truth table. The result gets stored in three different arrays, that further down in the method gets converted to a two-dimensional list. The last two "for loops" print out the result for each RotS. An example is show in Fig 3.5.

```
For function (x0*x1*x2 + x0 + x1 + x2 + 1) the sensitivity is:
x = ((0, 0, 0)) : S(f((0, 0, 0))) = 3
x = ((0, 0, 1)) : S(f((0, 0, 1))) = 3
x = ((0, 1, 0)) : S(f((0, 1, 0))) = 3
x = ((0, 1, 1)) : S(f((0, 1, 1))) = 2
x = ((1, 0, 0)) : S(f((1, 0, 0))) = 3
x = ((1, 0, 1)) : S(f((1, 0, 1))) = 2
x = ((1, 1, 0)) : S(f((1, 1, 0))) = 2
x = ((1, 1, 1)) : S(f((1, 1, 1))) = 0
```

Figure 3.5: Result for sensitivity $n = 2, 3, 4$

Chapter 4

Calculating sensitivity on RotS Boolean functions for n=5

In this chapter, the program used for calculating sensitivity on rotation symmetric functions for $n = 5$ is explained, and displayed in Figures 4.1-4.3. The chapter is divided into two sections. The first section concerns the construction of all Boolean functions for $n = 5$ and the filtering out of rotation symmetric Boolean functions. The next section covers the calculation of sensitivity on rotation symmetric Boolean functions for $n = 5$.

4.1 Constructing all Boolean functions and finding rotation symmetric Boolean functions

```
from sage.crypto.boolean_function import BooleanFunction
import numpy as np
import itertools

def generate_lower_functions(n_var, R, x):
    f = [R(0), R(1), x[0], x[0] + 1]

    for i in range(2, n_var+2):
        new_x = x[i-1]
        new_functions = []
        new2_functions= []
        add_function = new_functions.append
        add2_function = new2_functions.append
        for f1 in f:
            for f2 in f:
                g= (f1*new_x + f2)
                if(i<5):
                    add_function(g)

                if((g(0,0,0,0,1)==g(1,0,0,0,0)==g(0,1,0,0,0)==g
    (0,0,1,0,0)==g(0,0,0,1,0))and(g(0,0,0,1,1)==g(1,0,0,0,1)==g
    (1,1,0,0,0)==g(0,1,1,0,0)==g(0,0,1,1,0))and(g(0,0,1,0,1)==g
    (1,0,0,1,0)==g(0,1,0,0,1)==g(1,0,1,0,0)==g(0,1,0,1,0))and(g
    (0,0,1,1,1)==g(1,0,0,1,1)==g(1,1,0,0,1)==g(1,1,1,0,0)==g
    (0,1,1,1,0))and(g(1,0,1,0,1)==g(1,1,0,1,0)==g(0,1,1,0,1)==g
    (1,0,1,1,0)==g(0,1,0,1,1))and(g(0,1,1,1,1)==g(1,0,1,1,1)==g
    (1,1,0,1,1)==g(1,1,1,0,1)==g(1,1,1,1,0))):
                    add2_function(g)
        f = new_functions
        func = new2_functions
    return func

n = 5
ring = BooleanPolynomialRing(n, "x")
x = ring.gens()
x_map = Arrangements(x, n)

functions = generate_lower_functions(n-1, ring, x)
new_x = x[n-1]
```

Figure 4.1: Generating functions for $n = 5$

The fundamentals for creating the functions is the same for $n = 2, 3, 4$ and for $n = 5$. The method used for generating Boolean functions for $n = 5$ is shown i Fig 4.1. One of the differences is that this method generates functions and filters out the RotS Boolean functions in the same method. This means that the *generate_rots* method used for $n = 2, 3, 4$ shown in Fig 3.3 is not used here. One other difference is that this method does not append all the $2^{2^5} = 4294967296$ functions created. This is done because when trying to append the functions the program is trying to store all the 4 294 967 296 functions. When trying to store that many functions the program would crash, because of insufficient memory. The functions for $n = 5$ are still created, but not stored. Because the functions for $n = 5$ can not be stored, the RotS test had to be implemented inside this method. The RotS test is done similarly to the test for $n = 2, 3, 4$ shown in Fig 3.3. For $n = 5$ there are $2^5 = 32$ entries, and, after discarding $(0, 0, 0, 0, 0)$ and $(1, 1, 1, 1, 1)$, the method will test 30 entries. These 30 entries are divided into 6 subsets with 5 entries in each subset. For a function to pass the test, it must have the same value for each of the 5 entries in the subsets. After passing the test, the list of RotS Boolean functions is stored. A list of all RotS Boolean functions for $n = 5$, which has 256 functions, is returned.

4.2 Calculating sensitivity on RotS Boolean functions for $n = 5$

```
def sens(f):
    n=5
    t = list(itertools.product([(0), (1)], repeat=n))
    e1=[(1,0,0,0,0)]
    e2=[(0,1,0,0,0)]
    e3=[(0,0,1,0,0)]
    e4=[(0,0,0,1,0)]
    e5=[(0,0,0,0,1)]
    x_p_e1 =[]
    x_p_e2 =[]
    x_p_e3 =[]
    x_p_e4 =[]
    x_p_e5 =[]

    for i in range(len(t)):
        for j in range(n):
            x_p_e1.append(e1[0][j] ^^ t[i][j])
            x_p_e2.append(e2[0][j] ^^ t[i][j])
            x_p_e3.append(e3[0][j] ^^ t[i][j])
            x_p_e4.append(e4[0][j] ^^ t[i][j])
            x_p_e5.append(e5[0][j] ^^ t[i][j])

    x_pluss_e1 =[x_p_e1[i:i + n] for i in range(0, len(x_p_e1), n)]
    x_pluss_e2 =[x_p_e2[i:i + n] for i in range(0, len(x_p_e2), n)]
    x_pluss_e3 =[x_p_e3[i:i + n] for i in range(0, len(x_p_e3), n)]
    x_pluss_e4 =[x_p_e4[i:i + n] for i in range(0, len(x_p_e4), n)]
    x_pluss_e5 =[x_p_e5[i:i + n] for i in range(0, len(x_p_e5), n)]

    sum_list = []
    for i in range(len(f)):
        s1=[]
        s2=[]
        s3=[]
        s4=[]
        s5=[]
        for j in range(len(x_pluss_e1)):
            s1.append(f[i](*x_pluss_e1[j])+f[i](*t[j]))
            s2.append(f[i](*x_pluss_e2[j])+f[i](*t[j]))
            s3.append(f[i](*x_pluss_e3[j])+f[i](*t[j]))
            s4.append(f[i](*x_pluss_e4[j])+f[i](*t[j]))
            s5.append(f[i](*x_pluss_e5[j])+f[i](*t[j]))
```

Figure 4.2: Testing for sensitivity for $n = 5$ (1)

```
        integer_map1 = map(int, s1)
        integer_map2 = map(int, s2)
        integer_map3 = map(int, s3)
        integer_map4 = map(int, s4)
        integer_map5 = map(int, s5)

        l1 = list(integer_map1)
        l2 = list(integer_map2)
        l3 = list(integer_map3)
        l4 = list(integer_map4)
        l4 = list(integer_map4)

        sum_list.append([a + b + c + d + e for a, b, c, d, e in zip
    (l1, l2, l3, l4, l5)])

        my_array = np.array(sum_list)
         #print(sum_list)
    for i in range(len(f)):
        print('For function (' + str(f[i]) + ') the sensitivity is:
    ')
        for j in range(len(t)):
            print('x = (' + str(t[j]) +') : S(f('+ str(t[j]) +')) =
     ' + str(sum_list[i][j]))
```

Figure 4.3: Testing for sensitivity for $n = 5$ (2)

The method used for calculating the sensitivity for the RotS Boolean functions for $n = 5$ is less hard coded than for $n = 2, 3, 4$. The values for the truth table is auto generated with the use of itertools. The truth table values are stored in the variable with name t. The five different unit vectors called $\vec{e_1}, \vec{e_2}, \vec{e_3}, \vec{e_4}$ and $\vec{e_5}$ are still hard coded. The first double for loop in the method calculates the five different `x_pluss_e` needed for finding the sensitivity. In the previous programs `x_pluss_e` was hard coded, this is now done automatically. The rest of the program is done the same way as shown in chapter for $n = 2, 3, 4$. The method is shown in Fig 4.2 and 4.3.

Chapter 5

Result and Discussion

After the completion of the execution of the program explained in Section 3, a full sensitivity test for each rotation symmetric function in $\mathcal{B}_3$, $\mathcal{B}_4$ and $\mathcal{B}_5$ was stored in text, where each file consisted of 144, 1088 and 8448 lines each. The full data set for $n = 3$ is given in Appendix B.

The full data set for $\mathcal{B}_4$ and $\mathcal{B}_5$ are too large to be shown in this thesis – however, an attempt to summarize the interesting details of the result of $n = 5$ are given in the following Sections 5.1).

5.1 Results and analysis

Sensitivity	$n = 3$	$n = 4$	$n = 5$
0	2	2	2
1	-	-	-
2	2	4	-
3	12	4	18
4	-	54	22
5	-	-	214
Sum	16	64	256
Max sensitivity	3	4	5

Table 5.1: Sensitivity result

The full distribution of sensitivity for rotation symmetric functions in $n \leq 5$ variables is given in Table 5.1, summarizing the results of data collection conducted by use of program explained in Chapter 3 and 4. Table 5.1 displays how many functions exist with the different sensitivity for $n = 3, 4, 5$. The Sum-row of Table 5.1 shows the total amount of Rots functions for the different values of n. The columns for variable $n = 3$ is manually calculated from watching the result of the program shown in Fig 3.4.
The columns for variable $n = 4, 5$ are done by making a text file with the results from Fig 4.2 for $n = 5$ and one text file for $n = 4$, then running the text files through a program that counts how many functions have the different sensitivity.

We can see from the results and Table 5.1 that most Rotation Symmetric Boolean functions have the highest possible sensitivity, that is, the sensitivity is equal to the number of variables n. We can conclude that most of these functions are not optimal from the cryptographic point of view, since it is more desirable to use those that have sensitivity close to $\frac{1}{2}$, in order to avoid biases.

For better overview, we present here the distribution of the sensitivity of the functions for the different values of n shown by percentages: for $n = 3$, 12.5%

of the functions have sensitivity 0, while 12.5% have sensitivity 2 and 75% have sensitivity 3. For $n = 4$, we see that 3.125% of the functions have sensitivity 0, 6.25% have sensitivity 2, 6.25% have sensitivity 3 and 84.375% have sensitivity 4. For $n = 5$, we see that 0.78125% of the functions have sensitivity 0, 7.03125% have sensitivity 3, 8.5937% have sensitivity 4 and 83.59375% have sensitivity 5.

Here we present a complete list of Rots functions with sensitivity less than n (the number of variables). We discard the trivial cases $f = 0$ and $f = 1$ which always get sensitivity 0, and discarding $f + 1$ for every f because f and $f + 1$ always have the same sensitivity. We simplify the notation by identifying variable x_j with j. So, for instance, $0123 = x_0x_1x_2x_3$. A SANF (short algebraic normal form) is one of the possible choices of the terms that generate the other terms in the ANF so that f is rotation symmetric. For instance, for $n = 3$, if the term 01 is present, then by rotation the terms 02 and 12 have to be present as well. Hence, a SANF of $01, 02, 12$ for $n = 3$ is 01. More on SANF in Section 2.3.

$n = 3$
Sensitivity 2:
$01, 02, 12$

SANF:
01

$n = 4$
Sensitivity 2:
$0123, 02, 13$
$0123, 012, 013, 023, 123, 01, 03, 12, 23$

SANF:
$0123, 02$
$0123, 012, 013, 023, 123, 01, 03, 12, 23$

Sensitivity 3:

$0123, 01, 02, 03, 12, 13, 23$

$0123, 012, 013, 023, 123$

SANF:

$0123, 01, 02$

$0123, 012$

$n = 5$

Sensitivity 3:

$012, 013, 014, 023, 024, 034, 123, 124, 134, 234, 02, 03, 13, 14, 24$

$012, 013, 014, 023, 024, 034, 123, 124, 134, 234, 01, 04, 12, 23, 34$

$0123, 0124, 0134, 0234, 1234, 012, 013, 014, 023, 024, 034, 123, 124, 134, 234$

$0123, 0124, 0134, 0234, 1234, 012, 013, 014, 023, 024, 034, 123, 124, 134, 234, 02, 03, 13, 14, 24$

$0123, 0124, 0134, 0234, 1234, 012, 013, 014, 023, 024, 034, 123, 124, 134, 234, 01, 04, 12, 23, 34$

$01234, 013, 023, 024, 124, 134, 02, 03, 13, 14, 24$

$01234, 012, 014, 034, 123, 234, 01, 04, 12, 23, 34$

$01234, 0123, 0124, 0134, 0234, 1234, 013, 023, 024, 124, 134$

$01234, 0123, 0124, 0134, 0234, 1234, 012, 014, 034, 123, 234$

SANF:

$012, 013, 02$

$012, 013, 01$

$0123, 012, 013$

$0123, 012, 013, 02$

$0123, 012, 013, 01$

$01234, 013, 02$

$01234, 012, 01$

$01234, 0123, 013$

$01234, 0123, 012$

Sensitivity 4:

$02, 03, 13, 14, 24$

$01, 04, 12, 23, 34$

01, 02, 03, 04, 12, 13, 14, 23, 24, 34

0123, 0124, 0134, 0234, 1234

0123, 0124, 0134, 0234, 1234, 02, 03, 13, 14, 24

0123, 0124, 0134, 0234, 1234, 01, 04, 12, 23, 34

0123, 0124, 0134, 0234, 1234, 01, 02, 03, 04, 12, 13, 14, 23, 24, 34

01234, 013, 023, 024, 124, 134, 01, 02, 03, 04, 12, 13, 14, 23, 24, 34

01234, 012, 014, 034, 123, 234, 01, 02, 03, 04, 12, 13, 14, 23, 24, 34

01234, 0123, 0124, 0134, 0234, 1234, 013, 023, 024, 124, 134, 02, 03, 13, 14, 24

01234, 0123, 0124, 0134, 0234, 1234, 012, 014, 034, 123, 234, 01, 04, 12, 23, 34

SANF:

02

01

01, 02

0123

0123, 02

0123, 01

0123, 01, 02

01234, 013, 01, 02

01234, 012, 01, 02

01234, 0123, 013, 02

01234, 0123, 012, 01

One can investigate theoretically the sensitivity of some Rots. For instance, for any $n \geq 3$, we can see that the Rots with SANF 01 (i.e. x_0x_1) has sensitivity at least $n-2$:

$$f(\vec{x} \oplus \vec{e_0}) + \oplus f(\vec{x}) = x_1 + x_{n-1}$$

$$f(\vec{x} \oplus \vec{e_1}) + \oplus f(\vec{x}) = x_0 + x_2$$

$$\dots$$

$$f(\vec{x} \oplus \vec{e}_{n-3}) + \oplus f(\vec{x}) = x_{n-4} + x_{n-2}$$

$$f(\vec{x} \oplus \vec{e}_{n-2}) + \oplus f(\vec{x}) = x_{n-3} + x_{n-1}$$

This implies that, for $0 \leq i \leq n-2$, we are adding a new variable with each i, so that we get an independent system, and thus, the equation

$$f(\vec{x} \oplus \vec{e_i}) + \oplus f(\vec{x}) = 1$$

has a solution.

Chapter 6

Conclusions

In this thesis we have investigated sensitivity for rotation symmetric Boolean functions. We provided a complete enumeration results for up to $n = 5$. Our results show that most of the rotation symmetric Boolean functions have the highest possible sensitivity of n. This makes a large subset of these functions is not optimal from the cryptographic point of view. When it comes to the expected result, where we indicated it would be possible to find a way to examine rotation symmetric Boolean functions for a relatively high values of n, this was harder then expected because of the difficulty to create all the Boolean functions for $n < 5$.

In summary, the main results and definite conclusions of this thesis, are summarized in Table 5.1.

6.1 Further Work

By having a program that can create all functions, find RotS functions and testing for sensitivity up to $n = 5$, the obvious future project of interest would be to see if the same can be done for $n = 6, 7, 8$.

The first change that is needed is finding a way to store all functions for $n = 5$ such that functions for $n = 6$ can be created. One idea is to distribute the generate_functions method shown in Fig 4.1 into several methods, such that the methods can run on different computers at the same time. Another idea is to just create Rotation symmetric Boolean functions from the start, if possible,

such that there is no need to create all Boolean functions for the given n.

Also, it would be desirable to find (theoretically) a formula for functions whose sensitivity is close to $\frac{1}{2}$, in order to be able to construct these functions for a large value of n.

One idea that would make the code easier to work further with is removing the test for RotS function from inside the method for creating functions for $n = 5$, but rather have a specific method for filtering out RotS functions. The method used for generating functions can be seen in Fig 4.1.

Bibliography

[1] Claude Carlet. "On Cryptographic Complexity of Boolean Functions". In: *in Proc. 6th Conf. Finite Fields With Applications to Coding Theory. Cryptography and Related Areas.* (2002), pp. 53–69.

[2] Rod Haggarty. *Discrete Mathematics for Computing.* Pearson Education Ltd., 2002.

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

[4] C. Carlet. "Boolean Functions for Cryptography and Error Correcting Codes". In: *Chapter of the monography "Boolean Models and Methods in Mathematics, Computer Science, and Engineering"* (2010), pp. 257–397.

[5] David C. Lay. *Linear Algebra and Its Applications (International 4th ed.)* Pearson Education Inc., 1994.

[6] Pantelimon Stănică and Subhamoy Maitra. "Rotation symmetric Boolean functions—Count and cryptographic properties". In: *Discrete Applied Mathematics* 156.10 (2008), pp. 1567–1580. ISSN: 0166-218X. DOI: https://doi.org/10.1016/j.dam.2007.04.029. URL: https://www.sciencedirect.com/science/article/pii/S0166218X07001734.

[7] Nadia Creignou and Hervé Daudé. "Sensitivity of Boolean formulas". In: *European Journal of Combinatorics* 34.5 (2013), pp. 793–805. ISSN: 0195-6698. DOI: https://doi.org/10.1016/j.ejc.2012.12.006. URL: https://www.sciencedirect.com/science/article/pii/S0195669812002041.

[8] Paul Zimmermann. *SageMath- Open-Source Mathematical Software System.* URL: http://www.sagemath.org/.. (accessed 30.11.21.

[9] nikhilaggarwal3. *Python Itertools.* URL: https://www.geeksforgeeks.org/python-itertools/.. (accessed 30.11.21.

[10] C.X. Qu J. Pieprzyk. "Fast hashing and rotation-symmetric functions". In: *Journal of Universal Computer Science, vol. 5, no. 1* (1999), pp. 20–31.

Appendix A

Calculating Sensitivity for RotS functions for $n = 2$

The following is discussed in Chapter 2 in subsection 2.3. Here is a full calculation of sensitivity for Rots functions for $n = 2$

$$f(\vec{x}) = (0001)$$

$$\vec{e}_1 = (1, 0)$$

$$\vec{e}_2 = (0, 1)$$

X_1	X_2	$f(\vec{x})$	$\vec{x} \oplus \vec{e}_1$	$f(\vec{x} \oplus \vec{e}_1)$	$f(\vec{x}) \oplus f(\vec{x} \oplus \vec{e}_1)$
0	0	0	1 0	0	0
0	1	0	1 1	1	1
1	0	0	0 0	0	0
1	1	1	0 1	0	1

X_1	X_2	$f(\vec{x})$	$\vec{x} \oplus \vec{e}_2$	$f(\vec{x} \oplus \vec{e}_2)$	$f(\vec{x}) \oplus f(\vec{x} \oplus \vec{e}_2)$
0	0	0	0 1	0	0
0	1	0	0 0	0	0
1	0	0	1 1	1	1
1	1	1	1 0	0	1

$$\vec{x} = (0,0) : S(f(0,0)) = 0$$
$$\vec{x} = (0,1) : S(f(0,1)) = 1$$
$$\vec{x} = (1,0) : S(f(1,0)) = 1$$
$$\vec{x} = (1,1) : S(f(1,1)) = 2$$

Table A.1: Sensitivity for $f = 0001$

$$f(\vec{x}) = (0110)$$
$$\vec{e}_1 = (1,0)$$
$$\vec{e}_2 = (0,1)$$

X_1	X_2	$f(\vec{x})$	$\vec{x} \oplus \vec{e}_1$	$f(\vec{x} \oplus \vec{e}_1)$	$f(\vec{x}) \oplus f(\vec{x} \oplus \vec{e}_1)$
0	0	0	1 0	1	1
0	1	1	1 1	0	1
1	0	1	0 0	0	1
1	1	0	0 1	1	1

X_1	X_2	$f(\vec{x})$	$\vec{x} \oplus \vec{e}_1$	$f(\vec{x} \oplus \vec{e}_2)$	$f(\vec{x}) \oplus f(\vec{x} \oplus \vec{e}_2)$
0	0	0	0 1	1	1
0	1	1	0 0	0	1
1	0	1	1 1	0	1
1	1	0	1 0	1	1

$$\vec{x} = (0,0) : S(f(0,0) = 2$$
$$\vec{x} = (0,1) : S(f(0,1) = 2$$
$$\vec{x} = (1,0) : S(f(1,0) = 2$$
$$\vec{x} = (1,1) : S(f(1,1) = 2$$

Table A.2: Sensitivity for $f = 0110$

$$f(\vec{x}) = (0111)$$

$$\vec{e}_1 = (1, 0)$$

$$\vec{e}_2 = (0, 1)$$

X_1	X_2	$f(\vec{x})$	$\vec{x} \oplus \vec{e}_1$	$f(\vec{x} \oplus \vec{e}_1)$	$f(\vec{x}) \oplus f(\vec{x} \oplus \vec{e}_1)$
0	0	0	1 0	1	1
0	1	1	1 1	1	0
1	0	1	0 0	0	1
1	1	1	0 1	1	0
X_1	X_2	$f(\vec{x})$	$\vec{x} \oplus \vec{e}_1$	$f(\vec{x} \oplus \vec{e}_2)$	$f(\vec{x}) \oplus f(\vec{x} \oplus \vec{e}_2)$
0	0	0	0 1	1	1
0	1	1	0 0	1	1
1	0	1	1 1	0	0
1	1	1	1 0	0	0

$$\vec{x} = (0, 0) : S(f(0, 0) = 2$$

$$\vec{x} = (0, 1) : S(f(0, 1) = 1$$

$$\vec{x} = (1, 0) : S(f(1, 0) = 1$$

$$\vec{x} = (1, 1) : S(f(1, 1) = 0$$

Table A.3: Sensitivity for $f = 0111$

$$f(\vec{x}) = (1000)$$

$$\vec{e}_1 = (1, 0)$$

$$\vec{e}_2 = (0, 1)$$

X_1	X_2	$f(\vec{x})$	$\vec{x} \oplus \vec{e}_1$	$f(\vec{x} \oplus \vec{e}_1)$	$f(\vec{x}) \oplus f(\vec{x} \oplus \vec{e}_1)$
0	0	1	1 0	0	1
0	1	0	1 1	0	0
1	0	0	0 0	1	1
1	1	0	0 1	0	0

X_1	X_2	$f(\vec{x})$	$\vec{x} \oplus \vec{e}_1$	$f(\vec{x} \oplus \vec{e}_2)$	$f(\vec{x}) \oplus f(\vec{x} \oplus \vec{e}_2)$
0	0	1	0 1	0	1
0	1	0	0 0	1	1
1	0	0	1 1	0	0
1	1	0	1 0	0	0

$$\vec{x} = (0, 0) : S(f(0, 0) = 2$$

$$\vec{x} = (0, 1) : S(f(0, 1) = 1$$

$$\vec{x} = (1, 0) : S(f(1, 0) = 1$$

$$\vec{x} = (1, 1) : S(f(1, 1) = 0$$

Table A.4: Sensitivity for $f = 1000$

$$f(\vec{x}) = (1001)$$

$$\vec{e}_1 = (1,0)$$

$$\vec{e}_2 = (0,1)$$

X_1	X_2	$f(\vec{x})$	$\vec{x} \oplus \vec{e}_1$	$f(\vec{x} \oplus \vec{e}_1)$	$f(\vec{x}) \oplus f(\vec{x} \oplus \vec{e}_1)$
0	0	1	1 0	0	1
0	1	0	1 1	1	1
1	0	0	0 0	1	1
1	1	1	0 1	0	1

X_1	X_2	$f(\vec{x})$	$\vec{x} \oplus \vec{e}_1$	$f(\vec{x} \oplus \vec{e}_2)$	$f(\vec{x}) \oplus f(\vec{x} \oplus \vec{e}_2)$
0	0	1	0 1	0	1
0	1	0	0 0	1	1
1	0	0	1 1	1	1
1	1	1	1 0	0	1

$$\vec{x} = (0,0) : S(f(0,0) = 2$$

$$\vec{x} = (0,1) : S(f(0,1) = 2$$

$$\vec{x} = (1,0) : S(f(1,0) = 2$$

$$\vec{x} = (1,1) : S(f(1,1) = 2$$

Table A.5: Sensitivity for $f = 1001$

$$f(\vec{x}) = (1110)$$

$$\vec{e}_1 = (1, 0)$$

$$\vec{e}_2 = (0, 1)$$

X_1	X_2	$f(\vec{x})$	$\vec{x} \oplus \vec{e}_1$	$f(\vec{x} \oplus \vec{e}_1)$	$f(\vec{x}) \oplus f(\vec{x} \oplus \vec{e}_1)$
0	0	1	1 0	1	0
0	1	1	1 1	0	1
1	0	1	0 0	1	0
1	1	0	0 1	1	1

X_1	X_2	$f(\vec{x})$	$\vec{x} \oplus \vec{e}_1$	$f(\vec{x} \oplus \vec{e}_2)$	$f(\vec{x}) \oplus f(\vec{x} \oplus \vec{e}_2)$
0	0	1	0 1	1	0
0	1	1	0 0	1	0
1	0	1	1 1	0	1
1	1	0	1 0	1	1

$$\vec{x} = (0, 0) : S(f(0, 0) = 0$$

$$\vec{x} = (0, 1) : S(f(0, 1) = 1$$

$$\vec{x} = (1, 0) : S(f(1, 0) = 1$$

$$\vec{x} = (1, 1) : S(f(1, 1) = 2$$

Table A.6: Sensitivity for $f = 1110$

Appendix B

Full data set for $n = 3$

The following is a complete data set for $n = 3$, after using the method shown in chapter 3 and fig 3.4.

```
For function (0) the sensitivity is:
x = ((0, 0, 0)) : S(f((0, 0, 0))) = 0
x = ((0, 0, 1)) : S(f((0, 0, 1))) = 0
x = ((0, 1, 0)) : S(f((0, 1, 0))) = 0
x = ((0, 1, 1)) : S(f((0, 1, 1))) = 0
x = ((1, 0, 0)) : S(f((1, 0, 0))) = 0
x = ((1, 0, 1)) : S(f((1, 0, 1))) = 0
x = ((1, 1, 0)) : S(f((1, 1, 0))) = 0
x = ((1, 1, 1)) : S(f((1, 1, 1))) = 0
For function (1) the sensitivity is:
x = ((0, 0, 0)) : S(f((0, 0, 0))) = 0
x = ((0, 0, 1)) : S(f((0, 0, 1))) = 0
x = ((0, 1, 0)) : S(f((0, 1, 0))) = 0
x = ((0, 1, 1)) : S(f((0, 1, 1))) = 0
x = ((1, 0, 0)) : S(f((1, 0, 0))) = 0
x = ((1, 0, 1)) : S(f((1, 0, 1))) = 0
x = ((1, 1, 0)) : S(f((1, 1, 0))) = 0
x = ((1, 1, 1)) : S(f((1, 1, 1))) = 0
For function (x0 + x1 + x2) the sensitivity is:
x = ((0, 0, 0)) : S(f((0, 0, 0))) = 3
x = ((0, 0, 1)) : S(f((0, 0, 1))) = 3
x = ((0, 1, 0)) : S(f((0, 1, 0))) = 3
x = ((0, 1, 1)) : S(f((0, 1, 1))) = 3
x = ((1, 0, 0)) : S(f((1, 0, 0))) = 3
x = ((1, 0, 1)) : S(f((1, 0, 1))) = 3
x = ((1, 1, 0)) : S(f((1, 1, 0))) = 3
x = ((1, 1, 1)) : S(f((1, 1, 1))) = 3
For function (x0 + x1 + x2 + 1) the sensitivity is:
x = ((0, 0, 0)) : S(f((0, 0, 0))) = 3
x = ((0, 0, 1)) : S(f((0, 0, 1))) = 3
x = ((0, 1, 0)) : S(f((0, 1, 0))) = 3
x = ((0, 1, 1)) : S(f((0, 1, 1))) = 3
x = ((1, 0, 0)) : S(f((1, 0, 0))) = 3
x = ((1, 0, 1)) : S(f((1, 0, 1))) = 3
x = ((1, 1, 0)) : S(f((1, 1, 0))) = 3
x = ((1, 1, 1)) : S(f((1, 1, 1))) = 3
For function (x0*x1 + x0*x2 + x1*x2) the sensitivity is:
x = ((0, 0, 0)) : S(f((0, 0, 0))) = 0
x = ((0, 0, 1)) : S(f((0, 0, 1))) = 2
x = ((0, 1, 0)) : S(f((0, 1, 0))) = 2
x = ((0, 1, 1)) : S(f((0, 1, 1))) = 2
x = ((1, 0, 0)) : S(f((1, 0, 0))) = 2
x = ((1, 0, 1)) : S(f((1, 0, 1))) = 2
x = ((1, 1, 0)) : S(f((1, 1, 0))) = 2
x = ((1, 1, 1)) : S(f((1, 1, 1))) = 0
```


Figure B.1: Data set for $n = 3$ (1)

```
For function (x0*x1 + x0*x2 + x1*x2 + 1) the sensitivity is:
x = ((0, 0, 0)) : S(f((0, 0, 0))) = 0
x = ((0, 0, 1)) : S(f((0, 0, 1))) = 2
x = ((0, 1, 0)) : S(f((0, 1, 0))) = 2
x = ((0, 1, 1)) : S(f((0, 1, 1))) = 2
x = ((1, 0, 0)) : S(f((1, 0, 0))) = 2
x = ((1, 0, 1)) : S(f((1, 0, 1))) = 2
x = ((1, 1, 0)) : S(f((1, 1, 0))) = 2
x = ((1, 1, 1)) : S(f((1, 1, 1))) = 0
For function (x0*x1 + x0*x2 + x0 + x1*x2 + x1 + x2) the sensitivity
     is:
x = ((0, 0, 0)) : S(f((0, 0, 0))) = 3
x = ((0, 0, 1)) : S(f((0, 0, 1))) = 1
x = ((0, 1, 0)) : S(f((0, 1, 0))) = 1
x = ((0, 1, 1)) : S(f((0, 1, 1))) = 1
x = ((1, 0, 0)) : S(f((1, 0, 0))) = 1
x = ((1, 0, 1)) : S(f((1, 0, 1))) = 1
x = ((1, 1, 0)) : S(f((1, 1, 0))) = 1
x = ((1, 1, 1)) : S(f((1, 1, 1))) = 3
For function (x0*x1 + x0*x2 + x0 + x1*x2 + x1 + x2 + 1) the
    sensitivity is:
x = ((0, 0, 0)) : S(f((0, 0, 0))) = 3
x = ((0, 0, 1)) : S(f((0, 0, 1))) = 1
x = ((0, 1, 0)) : S(f((0, 1, 0))) = 1
x = ((0, 1, 1)) : S(f((0, 1, 1))) = 1
x = ((1, 0, 0)) : S(f((1, 0, 0))) = 1
x = ((1, 0, 1)) : S(f((1, 0, 1))) = 1
x = ((1, 1, 0)) : S(f((1, 1, 0))) = 1
x = ((1, 1, 1)) : S(f((1, 1, 1))) = 3
For function (x0*x1*x2) the sensitivity is:
x = ((0, 0, 0)) : S(f((0, 0, 0))) = 0
x = ((0, 0, 1)) : S(f((0, 0, 1))) = 0
x = ((0, 1, 0)) : S(f((0, 1, 0))) = 0
x = ((0, 1, 1)) : S(f((0, 1, 1))) = 1
x = ((1, 0, 0)) : S(f((1, 0, 0))) = 0
x = ((1, 0, 1)) : S(f((1, 0, 1))) = 1
x = ((1, 1, 0)) : S(f((1, 1, 0))) = 1
x = ((1, 1, 1)) : S(f((1, 1, 1))) = 3
```

Figure B.2: Data set for $n = 3$ (2)

```
For function (x0*x1*x2 + 1) the sensitivity is:
x = ((0, 0, 0)) : S(f((0, 0, 0))) = 0
x = ((0, 0, 1)) : S(f((0, 0, 1))) = 0
x = ((0, 1, 0)) : S(f((0, 1, 0))) = 0
x = ((0, 1, 1)) : S(f((0, 1, 1))) = 1
x = ((1, 0, 0)) : S(f((1, 0, 0))) = 0
x = ((1, 0, 1)) : S(f((1, 0, 1))) = 1
x = ((1, 1, 0)) : S(f((1, 1, 0))) = 1
x = ((1, 1, 1)) : S(f((1, 1, 1))) = 3
For function (x0*x1*x2 + x0 + x1 + x2) the sensitivity is:
x = ((0, 0, 0)) : S(f((0, 0, 0))) = 3
x = ((0, 0, 1)) : S(f((0, 0, 1))) = 3
x = ((0, 1, 0)) : S(f((0, 1, 0))) = 3
x = ((0, 1, 1)) : S(f((0, 1, 1))) = 2
x = ((1, 0, 0)) : S(f((1, 0, 0))) = 3
x = ((1, 0, 1)) : S(f((1, 0, 1))) = 2
x = ((1, 1, 0)) : S(f((1, 1, 0))) = 2
x = ((1, 1, 1)) : S(f((1, 1, 1))) = 0
For function (x0*x1*x2 + x0 + x1 + x2 + 1) the sensitivity is:
x = ((0, 0, 0)) : S(f((0, 0, 0))) = 3
x = ((0, 0, 1)) : S(f((0, 0, 1))) = 3
x = ((0, 1, 0)) : S(f((0, 1, 0))) = 3
x = ((0, 1, 1)) : S(f((0, 1, 1))) = 2
x = ((1, 0, 0)) : S(f((1, 0, 0))) = 3
x = ((1, 0, 1)) : S(f((1, 0, 1))) = 2
x = ((1, 1, 0)) : S(f((1, 1, 0))) = 2
x = ((1, 1, 1)) : S(f((1, 1, 1))) = 0
For function (x0*x1*x2 + x0*x1 + x0*x2 + x1*x2) the sensitivity is:
x = ((0, 0, 0)) : S(f((0, 0, 0))) = 0
x = ((0, 0, 1)) : S(f((0, 0, 1))) = 2
x = ((0, 1, 0)) : S(f((0, 1, 0))) = 2
x = ((0, 1, 1)) : S(f((0, 1, 1))) = 3
x = ((1, 0, 0)) : S(f((1, 0, 0))) = 2
x = ((1, 0, 1)) : S(f((1, 0, 1))) = 3
x = ((1, 1, 0)) : S(f((1, 1, 0))) = 3
x = ((1, 1, 1)) : S(f((1, 1, 1))) = 3
```

Figure B.3: Data set for $n = 3$ (3)

```
For function (x0*x1*x2 + x0*x1 + x0*x2 + x1*x2 + 1) the sensitivity
    is:
x = ((0, 0, 0)) : S(f((0, 0, 0))) = 0
x = ((0, 0, 1)) : S(f((0, 0, 1))) = 2
x = ((0, 1, 0)) : S(f((0, 1, 0))) = 2
x = ((0, 1, 1)) : S(f((0, 1, 1))) = 3
x = ((1, 0, 0)) : S(f((1, 0, 0))) = 2
x = ((1, 0, 1)) : S(f((1, 0, 1))) = 3
x = ((1, 1, 0)) : S(f((1, 1, 0))) = 3
x = ((1, 1, 1)) : S(f((1, 1, 1))) = 3
For function (x0*x1*x2 + x0*x1 + x0*x2 + x0 + x1*x2 + x1 + x2) the
    sensitivity is:
x = ((0, 0, 0)) : S(f((0, 0, 0))) = 3
x = ((0, 0, 1)) : S(f((0, 0, 1))) = 1
x = ((0, 1, 0)) : S(f((0, 1, 0))) = 1
x = ((0, 1, 1)) : S(f((0, 1, 1))) = 0
x = ((1, 0, 0)) : S(f((1, 0, 0))) = 1
x = ((1, 0, 1)) : S(f((1, 0, 1))) = 0
x = ((1, 1, 0)) : S(f((1, 1, 0))) = 0
x = ((1, 1, 1)) : S(f((1, 1, 1))) = 0
For function (x0*x1*x2 + x0*x1 + x0*x2 + x0 + x1*x2 + x1 + x2 + 1)
    the sensitivity is:
x = ((0, 0, 0)) : S(f((0, 0, 0))) = 3
x = ((0, 0, 1)) : S(f((0, 0, 1))) = 1
x = ((0, 1, 0)) : S(f((0, 1, 0))) = 1
x = ((0, 1, 1)) : S(f((0, 1, 1))) = 0
x = ((1, 0, 0)) : S(f((1, 0, 0))) = 1
x = ((1, 0, 1)) : S(f((1, 0, 1))) = 0
x = ((1, 1, 0)) : S(f((1, 1, 0))) = 0
x = ((1, 1, 1)) : S(f((1, 1, 1))) = 0
```

Figure B.4: Data set for $n = 3$ (4)

Appendix C

Calculating sensitivity on RotS functions for $n = 5$

The following is a list of Rots functions and there Sensitivity for $n = 5$ after discarding $f = 0$, $f = 1$ and $f + 1$.

$n = 5$ Sensitivity 3:

- $x0 * x1 * x2 + x0 * x1 * x3 + x0 * x1 * x4 + x0 * x2 * x3 + x0 * x2 * x4 + x0 * x2 + x0 * x3 * x4 + x0 * x3 + x1 * x2 * x3 + x1 * x2 * x4 + x1 * x3 * x4 + x1 * x3 + x1 * x4 + x2 * x3 * x4 + x2 * x4$
- $x0 * x1 * x2 + x0 * x1 * x3 + x0 * x1 * x4 + x0 * x1 + x0 * x2 * x3 + x0 * x2 * x4 + x0 * x3 * x4 + x0 * x4 + x1 * x2 * x3 + x1 * x2 * x4 + x1 * x2 + x1 * x3 * x4 + x2 * x3 * x4 + x2 * x3 + x3 * x4$
- $x0 * x1 * x2 * x3 + x0 * x1 * x2 * x4 + x0 * x1 * x2 + x0 * x1 * x3 * x4 + x0 * x1 * x3 + x0 * x1 * x4 + x0 * x2 * x3 * x4 + x0 * x2 * x3 + x0 * x2 * x4 + x0 * x3 * x4 + x1 * x2 * x3 * x4 + x1 * x2 * x3 + x1 * x2 * x4 + x1 * x3 * x4 + x2 * x3 * x4$
- $x0 * x1 * x2 * x3 + x0 * x1 * x2 * x4 + x0 * x1 * x2 + x0 * x1 * x3 * x4 + x0 * x1 * x3 + x0 * x1 * x4 + x0 * x2 * x3 * x4 + x0 * x2 * x3 + x0 * x2 * x4 + x0 * x2 + x0 * x3 * x4 + x0 * x3 + x1 * x2 * x3 * x4 + x1 * x2 * x3 + x1 * x2 * x4 + x1 * x3 * x4 + x1 * x3 + x1 * x4 + x2 * x3 * x4 + x2 * x4$
- $x0 * x1 * x2 * x3 + x0 * x1 * x2 * x4 + x0 * x1 * x2 + x0 * x1 * x3 * x4 + x0 *$

$x1 * x3 + x0 * x1 * x4 + x0 * x1 + x0 * x2 * x3 * x4 + x0 * x2 * x3 + x0 * x2 * x4 + x0 * x2 * x4 + x0 * x3 * x4 + x0 * x4 + x1 * x2 * x3 * x4 + x1 * x2 * x3 + x1 * x2 * x4 + x1 * x2 + x1 * x3 * x4 + x2 * x3 * x4 + x2 * x3 + x3 * x4$

- $x0 * x1 * x2 * x3 * x4 + x0 * x1 * x3 + x0 * x2 * x3 + x0 * x2 * x4 + x0 * x2 + x0 * x3 + x1 * x2 * x4 + x1 * x3 * x4 + x1 * x3 + x1 * x4 + x2 * x4$
- $x0 * x1 * x2 * x3 * x4 + x0 * x1 * x2 + x0 * x1 * x4 + x0 * x1 + x0 * x3 * x4 + x0 * x4 + x1 * x2 * x3 + x1 * x2 + x2 * x3 * x4 + x2 * x3 + x3 * x4$
- $x0 * x1 * x2 * x3 * x4 + x0 * x1 * x2 * x3 + x0 * x1 * x2 * x4 + x0 * x1 * x3 * x4 + x0 * x1 * x3 + x0 * x2 * x3 * x4 + x0 * x2 * x3 + x0 * x2 * x4 + x1 * x2 * x3 * x4 + x1 * x2 * x4 + x1 * x3 * x4$
- $x0 * x1 * x2 * x3 * x4 + x0 * x1 * x2 * x3 + x0 * x1 * x2 * x4 + x0 * x1 * x2 + x0 * x1 * x3 * x4 + x0 * x1 * x4 + x0 * x2 * x3 * x4 + x0 * x3 * x4 + x1 * x2 * x3 * x4 + x1 * x2 * x3 + x2 * x3 * x4$

$n = 5$ Sensitivity 4:

- $x0 * x2 + x0 * x3 + x1 * x3 + x1 * x4 + x2 * x4$
- $x0 * x1 + x0 * x4 + x1 * x2 + x2 * x3 + x3 * x4$
- $x0 * x1 + x0 * x2 + x0 * x3 + x0 * x4 + x1 * x2 + x1 * x3 + x1 * x4 + x2 * x3 + x2 * x4 + x3 * x4$
- $x0 * x1 * x2 * x3 + x0 * x1 * x2 * x4 + x0 * x1 * x3 * x4 + x0 * x2 * x3 * x4 + x1 * x2 * x3 * x4$
- $x0 * x1 * x2 * x3 + x0 * x1 * x2 * x4 + x0 * x1 * x3 * x4 + x0 * x2 * x3 * x4 + x0 * x2 + x0 * x3 + x1 * x2 * x3 * x4 + x1 * x3 + x1 * x4 + x2 * x4$
- $x0 * x1 * x2 * x3 + x0 * x1 * x2 * x4 + x0 * x1 * x3 * x4 + x0 * x1 + x0 * x2 * x3 * x4 + x0 * x4 + x1 * x2 * x3 * x4 + x1 * x2 + x2 * x3 + x3 * x4$
- $x0 * x1 * x2 * x3 + x0 * x1 * x2 * x4 + x0 * x1 * x3 * x4 + x0 * x1 + x0 * x2 * x3 * x4 + x0 * x2 + x0 * x3 + x0 * x4 + x1 * x2 * x3 * x4 + x1 * x2 + x1 * x3 + x1 * x4 + x2 * x3 + x2 * x4 + x3 * x4$

- $x0 * x1 * x2 * x3 * x4 + x0 * x1 * x3 + x0 * x1 + x0 * x2 * x3 + x0 * x2 * x4 + x0 * x2 + x0 * x3 + x0 * x4 + x1 * x2 * x4 + x1 * x2 + x1 * x3 * x4 + x1 * x3 + x1 * x4 + x2 * x3 + x2 * x4 + x3 * x4$

- $(x0 * x1 * x2 * x3 * x4 + x0 * x1 * x2 + x0 * x1 * x4 + x0 * x1 + x0 * x2 + x0 * x3 * x4 + x0 * x3 + x0 * x4 + x1 * x2 * x3 + x1 * x2 + x1 * x3 + x1 * x4 + x2 * x3 * x4 + x2 * x3 + x2 * x4 + x3 * x4$

- $x0 * x1 * x2 * x3 * x4 + x0 * x1 * x2 * x3 + x0 * x1 * x2 * x4 + x0 * x1 * x3 * x4 + x0 * x1 * x3 + x0 * x2 * x3 * x4 + x0 * x2 * x3 + x0 * x2 * x4 + x0 * x2 + x0 * x3 + x1 * x2 * x3 * x4 + x1 * x2 * x4 + x1 * x3 * x4 + x1 * x3 + x1 * x4 + x2 * x4$

- $x0 * x1 * x2 * x3 * x4 + x0 * x1 * x2 * x3 + x0 * x1 * x2 * x4 + x0 * x1 * x2 + x0 * x1 * x3 * x4 + x0 * x1 * x4 + x0 * x1 + x0 * x2 * x3 * x4 + x0 * x3 * x4 + x0 * x4 + x1 * x2 * x3 * x4 + x1 * x2 * x3 + x1 * x2 + x2 * x3 * x4 + x2 * x3 + x3 * x4$