University of Bergen
Department of informatics

An efficient implementation of a test for EA-equivalence

Marie Heggebakk

UNIVERSITETET I BERGEN
Det matematisk-naturvitenskapelige fakultet

May, 2022

Acknowledgments

First of all, I would like to thank my supervisors, Lilya Budaghyan and Nikolay Kaleyski at the Selmer Center for their guidance and support throughout my thesis.

A special thanks to Nikolay for helping me formulate my thesis, always being available and for answering all my questions. His involvement and encouragement have been highly appreciated and invaluable. I will never forget all our epic debugging sessions.

I would also want to thank my family and friends for their support and believing in me.

Last, but not least, I would like to thank my boyfriend, Ståle, for always being there for me, and his endless support and encouragement throughout the last year.

Marie Heggebakk
Bergen, 2022

Abstract

We implement an algorithm for testing EA-equivalence between vectorial Boolean functions proposed by Kaleyski in the *C* programming language, and observe that it reduces the running time (as opposed to the original *Magma* implementation of the algorithm) necessary to decide equivalence up to 300 times in many cases. Our implementation also significantly reduces the memory usage, and makes it possible to run the algorithms for dimensions from 10 onwards, which was impossible using the original implementation due to its memory consumption. Our approach allows us to reconstruct the exact form of the equivalence and to prove that two given functions are equivalent (for comparison, computing invariants for the functions, which is the approach typically used in practice, only allows us to show that two functions are not equivalent). Furthermore, our approach works for functions of any algebraic degree, while most existing approaches (such as invariants and other algorithms for EA-equivalence) are restricted to the quadratic case.

We then adapt Kaleyski's algorithm to test for linear and affine equivalence instead of EA-equivalence. We supply an implementation in *C* of this procedure as well. As an application, we show how this method can be used to test quadratic APN functions for EA-equivalence through the linear equivalence of their orthoderivatives. We observe that by taking this approach, we can reduce the time necessary for deciding EA-equivalence up to 20 times (as compared with our efficient *C* implementation from the previous paragraph). The downside compared to Kaleyski's original algorithm is that this faster method makes it difficult to recover the exact form of the EA-equivalence between the tested APN functions. We confirm this by running some computational experiments in dimension 6, and observing that only one out of all possible linear equivalences between the orthoderivatives corresponds to the EA-equivalence between the APN functions in question. To the best of our knowledge, this is the first investigation into the exact relationship between the EA-equivalence of quadratic APN functions and the affine equivalence of their orthoderivatives given in the literature.

Contents

Chapter 1

Introduction

Cryptographically optimal functions (such as APN functions) are typically classified up to equivalence relations such as CCZ-equivalence and EA-equivalence. This reduces the number of functions that have to be considered and makes searching for them easier. However, this raises the practical problem of testing whether a given pair of functions is equivalent. This is crucial to e.g. constructing new APN functions, since an APN function is only considered to be new if it is inequivalent to all currently known ones.

Testing CCZ- and EA-equivalence of vectorial Boolean functions is a very hard computational problem. At present, no efficient way exists for testing CCZ-equivalence from first principles. It is possible to test CCZ-equivalence through linear codes, but this only works reliably below dimension 10; for dimensions 10 and above, this approach can give false negatives. In fact, this method cannot be used for dimensions greater than 10 at all due to its huge memory consumption (even on our department server having around 500 GB of memory). In practice, it can be shown that a given pair of functions are inequivalent using invariants such as the Γ- and Δ-rank, or the differential spectrum of the orthoderivative. One issue with this approach, is that invariants can only be used to show that functions are inequivalent (if they have distinct values of a given invariant); it is not possible to prove that two functions are equivalent in this way. Furthermore, the known invariants can be restrictive with respect to their use cases: the Γ- and Δ-rank, for instance, are only usable up to dimension 10, and are quite slow to compute; the orthoderivatives are only defined in the case of quadratic APN functions.

In many cases (such as for quadratic APN functions), testing CCZ-equivalence can be reduced to testing other equivalence relations such as EA-equivalence. Testing EA-equivalence is possible using linear codes as well, but this approach has the same problem as in the case of CCZ-equivalence. Recently, an algorithm for testing EA-equivalence from first principles (without going through linear codes) was proposed by Kaleyski. Realizing an efficient implementation of this algorithm was left as a problem for future work.

In this thesis, we implement Kaleyski's algorithm in *C*, and observe that this implementation is significantly faster than the existing proof of concept implementation in *Magma* provided by Kaleyski. More precisely, our implementation is up to 300 times faster, and more memory efficient than the *Magma* one. Furthermore, our implementation has a significantly lower memory consumption, and can be used in dimensions $n \geq 10$ which was impossible using the *Magma* implementation. Besides testing whether two functions are EA-equivalent, our approach can recover the exact form of the equivalence. We conduct computational experiments and summarize the running times in order to give a clear picture of how much more efficient our *C* implementation is.

We also know that if two quadratic APN functions are EA-equivalent, then their orthoderivatives are linear equivalent. To date, there is no efficient algorithm for testing affine or linear equivalence in the general case. We adapt Kaleyski's algorithm to the case of testing linear and affine equivalence, and implement this new procedure in *C*. We test its efficiency by applying it to the orthoderivatives of some of the known quadratic APN functions. We observe that by doing so, we can cut down the computation time for deciding EA-equivalence by another factor of up to 20 times (as opposed to the *C* implementation of Kaleyski's algorithm discussed above). Unfortunately, we also observe that reconstructing the exact form of the EA-equivalence is very difficult in this way, and if this is necessary, a different algorithm should be used. Nonetheless, the most frequent use case involves merely testing whether two given functions are equivalent or not, and then our newly developed procedure can significantly reduce the running times.

We provide open source implementations of both algorithm online.

Chapter 2

Background

2.1 Vectorial Boolean functions

Let us denote by $\mathbb{F}_2 = \{0, 1\}$ the finite field of two elements, and by $\mathbb{F}_2^n$ the vector space of dimension n over $\mathbb{F}_2$. We consider functions that take n binary inputs (that is, zeros and ones) and produce a single binary output, i.e. functions from $\mathbb{F}_2^n$ to $\mathbb{F}_2$. Such a function is called an **$(n, 1)$-function**, also known as a **Boolean function**. Any information can be represented in binary, i.e. as a sequence of binary values. Therefore, any data can be expressed using Boolean functions. Because of this, Boolean functions are widely used in many areas within mathematics and computer science.

A Boolean function f gives only one binary output. If we need to output more than a single bit of data, it is necessary to use several Boolean functions. For instance, if we need to output m bits of data, we would need m Boolean functions $f_1, f_2, \ldots, f_m$. By combining these into a vector $F = (f_1, f_2, \ldots, f_m)$, we obtain what is called an **(n, m)-function**, taking n binary inputs and producing m binary outputs. In other words, we obtain a function from $\mathbb{F}_2^n$ to $\mathbb{F}_2^m$. This is also known as a **vectorial Boolean function**, and the Boolean functions $f_1, f_2, \ldots, f_m$ are called the **coordinate functions** of F.

The nonzero linear combinations of the coordinate functions are called the **component functions** of F. Thus, every component is a coordinate but not every coordinate is a component. The component functions of a vectorial Boolean function are needed for the definition of nonlinearity which we will give later. For example: if $F = (f_1, f_2, f_3)$ has the coordinates f_1, f_2 and f_3, then the components would be $f_1, f_2, f_3, f_1 + f_2, f_1 + f_3, f_2 + f_3$ and $f_1 + f_2 + f_3$. Since every component function is the sum of a subset of the coordinate functions of F and there are n coordinate functions in total, we can represent each component function by an n-dimensional binary vector. For instance, $f_1 + f_3$ for $F = (f_1, f_2, f_3)$ can be represented as $b = (1, 0, 1)$ since the first an last coordinate occur in the sum, but the second one does not. We denote the component functions of F by F_b with $b \in \mathbb{F}_2^n$ for $b \neq (0, 0, \ldots, 0)$.

The easiest way to represent a Boolean function or a vectorial Boolean function is with a **Truth Table**. The Truth Table explicitly lists the output for every possible input to the function. An example of a Truth Table of a Boolean function is given in Table 2.1 and of a vectorial Boolean function in Table 2.2. Table 2.1 represents the Boolean function $f(x_1, x_2, x_3) = x_1 + x_2 + x_3$ and Table 2.2 represents the $(4, 2)$-function $F = (f_1, f_2)$ with $f_1(x_1, x_2, x_3, x_4) = x_1 + x_2$ and $f_2(x_2, x_2, x_3, x_4) = x_3 + x_4$.

When working with (n, m)-functions on a computer, we can represent their truth table as a list of integers between 0 and $2^m - 1$. To do this, we identify each vector of $\mathbb{F}_2^m$ with the integer that is its binary expansion. For instance, the vector $(1, 0, 0, 1, 1)$ would be represented by the integer $2^4 + 2^1 + 2^0 = 19$. The truth table can then be written in a file as a sequence $s_0, s_1, s_2, \ldots, s_{2^n-1}$ of integers, where s_i is the value of the function for the input vector corresponding to the integer i.

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

Table 2.1: Truth Table of an $(3, 1)$-function

x_1	x_2	x_3	x_4	$f_1(x_1,x_2,x_3,x_4)$	$f_2(x_1,x_2,x_3,x_4)$
0	0	0	0	0	0
0	0	0	1	0	1
0	0	1	0	0	1
0	0	1	1	0	0
0	1	0	0	1	0
0	1	0	1	1	1
0	1	1	0	1	1
0	1	1	1	1	0
1	0	0	0	1	0
1	0	0	1	1	1
1	0	1	0	1	1
1	0	1	1	1	0
1	1	0	0	0	0
1	1	0	1	0	1
1	1	1	0	0	1
1	1	1	1	0	0

Table 2.2: Truth Table of an $(4,2)$-function

The downside with this kind of representation is that the size of the truth table becomes large even for relatively small values of n. For $n = 20$, the table would end up with $2^{20} = 1\,048\,576$ entries. Because of this, it may be better to use other representations that can be more compact than the truth table, such as the Algebraic normal form and the univariate representation.

2.1.1 Algebraic normal form (ANF)

The **Algebraic normal form (ANF)** of an $(n,1)$-function f is the polynomial

$$\begin{aligned}F(x_1,\dots,x_n) =& a_\emptyset + a_1x_1 + \cdots + a_nx_n + \\ & a_{1,2}x_1x_2 + a_{1,3}x_1x_3 + \cdots + a_{n-1,n}x_{n-1}x_n + \\ & a_{1,2,3}x_1x_2x_3 + a_{1,2,4}x_1x_2x_4 + \cdots + a_{n-2,n-1,n}x_{n-2}x_{n-1}x_n + \\ & \cdots + a_{1,2,3,\dots,n}x_1x_2x_3\cdots x_n,\end{aligned}$$

with n binary variables $x_1, x_2, \dots, x_n$, where the coefficients $a_\emptyset, a_1, \dots, a_{1,2,\dots,n}$ are binary. Any $(n,1)$-function has a uniquely defined ANF.

We can see for example that the $(n, 1)$-function given in Table 2.1 can be represented as $f(x_1, x_2, x_3) = x_1 + x_2 + x_3$. Since this function has three inputs, the ANF consists of $2^3 = 8$ terms, but only three of them have non-zero coefficients, and so the ANF is quite short.

Vectorial Boolean functions can be represented in a similar way using the ANF, except that the coefficients are vectors of bits. For instance, the function given in Table 2.2 will be represented as $F(x_1, x_2, x_3, x_4) = (1, 0)x_1 + (1, 0)x_2 + (0, 1)x_3 + (0, 1)x_4$. Note that in the worst case, the ANF of a $(4, 2)$-function could have $2^4 = 16$ terms, but in this case only 4 of them have non-zero coefficients. Thus, in many cases, the ANF can be more compact than the Truth Table, which always has 2^n entries. Just like in the case of $(n, 1)$-functions, any (n, m)-function has a uniquely defined ANF.

Another benefit of the ANF is that it is easy to compute important properties such as the algebraic degree, which is defined in the next section.

2.1.2 Univariate representation

In the particular case when $n = m$, (n, n)-functions can be conveniently expressed using univariate polynomials over a finite field. Let $\mathbb{F}_{2^n}$ denote the finite field with 2^n elements for some positive integer n. Recall that $\mathbb{F}_2^n$ can be identified with $\mathbb{F}_{2^n}$, so that the vectors of $\mathbb{F}_2^n$ can be interpreted as finite field elements. Then any (n, n)-function can be seen as a function from $\mathbb{F}_{2^n}$ to itself, and this allows it to be uniquely represented as a univariate polynomial over $\mathbb{F}_{2^n}$ of the form

$$\sum_{i=0}^{2^n-1} a_i x^i, \quad a_i \in \mathbb{F}_{2^n}.$$

Some important classes of (n, n)-functions, such as APN functions have a very simple univariate representation; in fact, almost all known constructions of APN functions use the univariate representation.

2.1.3 Walsh Transform

Vectorial Boolean functions can also be represented with the Walsh transform. The **Walsh transform** of a vectorial Boolean function F is the function W_F that takes a pair of elements from $\mathbb{F}_{2^n} \times \mathbb{F}_{2^m}$ as input, and outputs an integer in $\mathbb{Z}$, i.e. $W_F : \mathbb{F}_{2^n} \times \mathbb{F}_{2^m} \to \mathbb{Z}$. The formula for computing the values of the Walsh transform of an (n, m)-function F is

$$W_F(a, b) = \sum_{x \in \mathbb{F}_{2^n}} (-1)^{\mathrm{Tr}_m(bF(x)) + \mathrm{Tr}_n(ax)},$$

where $\mathrm{Tr}_n(x)$ is the absolute trace function $\mathrm{Tr}_n(x) = x + x^2 + x^{2^2} + \cdots + x^{2^{n-1}}$.

The multiset $\{|W_F(a, b)| \ : \ a, b \in \mathbb{F}_{2^n}\}$ of the absolute values of W_F is called the **extended Walsh spectrum** of F.

The Walsh transform is invertible, i.e. if we know all the values $W_F(a, b)$ for all a and b, then we can uniquely reconstruct F; this is why the values of W_F can be seen as a representation of F. The Walsh transform can also be used to express many of the important properties of vectorial Boolean functions such as their differential uniformity and nonlinearity, which we will see later.

2.2 Cryptographic Properties

Boolean functions and vectorial Boolean functions play an important role in the design and analysis of cryptographic primitives. In cryptography, the importance of secure functions that are resilient to different types of cryptanalytic techniques is extremely high. There are different types of attacks that exploit weaknesses in functions used in block ciphers. If it is possible to exploit patterns and regularities in the behavior of a function, an attacker can be able to crack the cipher. This is why the design and analysis of cryptographic primitives crucially relies on (n, m)-functions with high cryptographic security, and researchers in the area work on finding new functions with optimal values of various cryptographic properties. In this section, we introduce some of the most important cryptographic properties for (n, m)-functions that are also relevant to our work.

2.2.1 Algebraic degree

The **algebraic degree**, denoted $\deg(F)$, is one of the essential properties of a vectorial Boolean function and it measures the resistance of the function to higher-order differential attacks. If a function F is given in ANF, we can quickly compute $\deg(F)$, which is the largest number of variables in any term with a non-zero coefficient. Suppose that we have the ANF

$$F(x_1, x_2, x_3, x_4) = (1,0,1,0)x_1x_3x_4+(1,1,1,1)x_3+(0,1,1,0)x_3x_4+(1,1,0,1).$$

We can see that there are four terms with non-zero coefficients: $x_1x_3x_4$, x_3, x_3x_4 and the constant term 1. To find the degree of these terms, we look at the number of variables that they contain. Therefore, the degree of the first term, $x_1x_3x_4$, is equal to 3; the degree of the second term x_3 is 1, and the degree of the third term x_3x_4 is 2, while the degree of the constant term is, of course, 0. The algebraic degree of F is then $\deg(F) = 3$.

It is also easy to compute the algebraic degree from the univariate representation, in which case it is expressed in terms of the binary weights of the exponents. For instance, the number 19 can be represented in binary as 10011. Since it has three non-zero bits, its binary weight is 3. The algebraic degree of a function $F(x) = \sum a_i x^i$ given in univariate form is then the largest binary weight of an exponent i with $a_i \neq 0$.

The algebraic degree can be used to define some important classes of functions that are used throughout the literature. An **affine** function is an (n, m)-function where $\deg(F) \leq 1$. Equivalently, it has the property that

$$F(x) + F(y) + F(z) = F(x + y + z)$$

for all $x, y, z \in \mathbb{F}_2^n$. An affine function F where $F(0) = 0$ is called **linear** and has the property that

$$F(x) + F(y) = F(x + y)$$

for all $x, y \in \mathbb{F}_2^n$. A **quadratic** function is a function where $\deg(F) = 2$.

It is easy to see that an affine function is, in fact, a linear function plus a constant. Linear and affine functions play an essential role in various transformations such as equivalence relations. However, they act predictably, and so they are not suitable for cryptographic purposes by themselves since ciphers should not have any obvious patterns or regularities that the attackers can exploit.

2.2.2 Differential uniformity

One of the most important cryptographic properties of an (n, m)-function is the differential uniformity, which describes how strong the correlation between the input and output differences to a function is. The differential uniformity determines the resistance against differential cryptanalysis, which is a powerful attack applied to block ciphers [1]. The **derivative** D_aF of F, in the **direction** $a \in \mathbb{F}_2^n$ is the (n, m)-function

$$D_aF(x) = F(a + x) - F(x).$$

Since addition and subtraction are the same operation over $\mathbb{F}_2^n$, this is usually written as

$$D_aF(x) = F(a + x) + F(x).$$

Let $\delta_F(a, b)$ denote the number of solutions $x \in \mathbb{F}_2^n$ to the equation $D_aF(x) = b$, i.e.

$$\delta_F(a, b) = \#\{x : D_aF(x) = b\}.$$

The **differential uniformity**, Δ_F, of an (n, m)-function F is the largest value of $\delta_F(a, b)$, i.e.

$$\Delta_F = \max\{\delta_F(a, b) : a \in \mathbb{F}_2^n, b \in \mathbb{F}_2^m | a \neq 0\}.$$

The multiset of all possible values of $\delta_F(a, b)$ for all $a \in \mathbb{F}_2^n, b \in \mathbb{F}_2^m$ where $a \neq 0$ is called the **differential spectrum** of F.

To provide good resistance against differential attacks, δ_F should be as low as possible. We are mostly interested in the case when $n = m$. The smallest possible value in this case is $\delta_F = 2$. Suppose a function has $\delta_F = 2$. Then it is called an **almost perfect nonlinear (APN)** function, which provides the best possible resistance against differential attacks.

2.2.3 Nonlinearity

Another powerful attack against block ciphers is the linear attack. The linear attack attempts to approximate an (n, n)-function F used in a block cipher by using a linear or an affine function. As mentioned above, linear and affine functions behave predictably and can be easily analyzed.

Nonlinearity measures the distance between any component function F_b of an (n, n)-function F and any affine $(n, 1)$-function A. The distance between any choice of F_b and A should be large in order to make approximation difficult. To measure this distance, we use the notion of **Hamming distance**, which counts the number of inputs that two functions disagree on. The Hamming distance is defined as

$$d_H(F, G) = \#\{x \in \mathbb{F}_2^n : F(x) \neq G(x)\}.$$

We consider that the functions F and G are close to one another if the Hamming distance is low. Using the Hamming distance, we can define the nonlinearity as follows. The nonlinearity of a Boolean function f with n variables is the minimum Hamming distance between f and any affine Boolean function with n variables. Let us denote the set of all affine Boolean functions on n variables by $\mathcal{A}_n$. For a vectorial Boolean function F the nonlinearity $\mathcal{NL}(F)$ is the minimum distance between any of its component functions F_b and any affine Boolean function in $\mathcal{A}_n$, i.e.

$$\mathcal{NL}(F) = \min\{d_H(F_b, l) : b \in \mathbb{F}_{2^n}, \quad b \neq 0, l \in \mathcal{A}_n\}.$$

It is known from [2] that the nonlinearity of any (n, n)-function F satisfies

$$\mathcal{NL}(F) \leq 2^{n-1} - 2^{(n-1)/2}.$$

If a function attains this optimal value, it is called **Almost Bent (AB)**. An AB function provides the best resistance against linear cryptanalysis. This optimal nonlinearity can only be achieved if n is odd, which makes AB functions only exist for odd dimensions n. Any AB function is also APN, which makes these function not only good against linear cryptanalysis, but also against differential cryptanalysis. Even though all AB functions are APN, not all APN functions are AB, but it is known that all quadratic APN functions for odd n are AB [3].

2.3 Equivalence Relations

The number of (n, m)-functions grows exponentially as n increases, so that it becomes extremely large even for small values of n. Because of this, the number of functions needs to be reduced with the help of equivalence relations. There are many ways to define equivalence relations, but for them to be useful in practice, they need to preserve the cryptographic properties that we study, such as differential uniformity and nonlinearity. If two functions F and G are equivalent, they are considered to be "the same", which helps to reduce the size of the search space and the number of functions that we need to consider.

In this section, we introduce the most frequently used equivalence relations on vectorial Boolean functions. As we shall see, these relations form a hierarchy of increasing generality. The relationship between the individual equivalence relations is visualized in Figure 2.1.

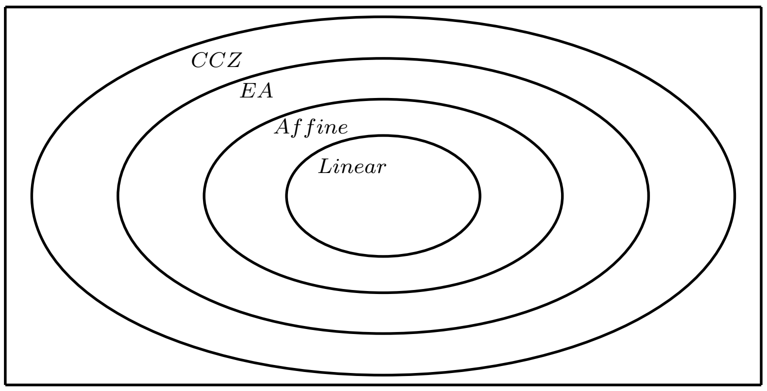

Figure 2.1: Equivalence relations

2.3.1 Linear and Affine equivalence

The linear equivalence relation is one of the simplest equivalence relations on (n, m)-functions. If a function F satisfies $F(x) \neq F(y)$ for all $x \neq y$ it is called a permutation. Suppose that L_1 and L_2 are linear permutations of $\mathbb{F}_2^n$ and $\mathbb{F}_2^m$, respectively, and that $L_1 \circ F \circ L_2 = G$ for two (n, m)-functions F and G, where the symbol $\circ$ denotes functional composition. Then F and G are said to be **linear equivalent**. If L_1 and L_2 are affine, then F and G are called **affine equivalent**. Since linear functions are a special case of affine functions, linear equivalence is a special case of affine equivalence, so that if F and G are linear equivalent, then they are also affine equivalent, but not necessarily vice-versa.

Linear and affine equivalence preserve a number of important cryptographic properties, such as the differential uniformity, nonlinearity and the algebraic degree. They also preserve the property of a function being a permutation.

2.3.2 EA-equivalence

Extended Affine equivalence, or EA-equivalence, is rather similar to affine equivalence, but it is more general because it allows for the addition of an affine function. More precisely, we say that F and G are **EA-equivalent** if $A_1 \circ F \circ A_2 + A = G$, where A_1, A_2, A are affine functions, and A_1, A_2 are permutations.

EA-equivalence preserves most of the important cryptographic properties, such as the differential uniformity, nonlinearity and algebraic degree, but unlike affine equivalence it does not preserve the property of the function being a permutation. EA-equivalence is one of the most frequently used relations together with CCZ-equivalence which is described below. CCZ-equivalence is strictly more general than EA-equivalence, and it is currently the most general known equivalence relation that preserves differential uniformity and nonlinearity. Because of this, APN functions are typically classified up to CCZ-equivalence. However, in many important cases, CCZ-equivalence coincides with EA-equivalence.

For instance, we know by [4] that two quadratic APN functions are CCZ-equivalent if and only if they are EA-equivalent. Most of the APN functions that are known are quadratic, and because of this, EA-equivalence is almost as important as CCZ-equivalence in the classification of APN functions.

2.3.3 CCZ-equivalence

Carlet-Charpin-Zinoviev equivalence, or CCZ-equivalence is the most general equivalence relation that is used on APN functions. Two vectorial Boolean functions F and G are called **CCZ-equivalent**, if there exists an affine permutation that maps the graph of F, Γ_F, to the graph of G, Γ_G, where $\Gamma_F = \{(x, F(x)) : x \in \mathbb{F}_2^n\}$ and $\Gamma_G = \{(x, G(x)) : x \in \mathbb{F}_2^n\}$. CCZ-equivalence is more general than EA-equivalence so that any two functions that are EA-equivalent are also CCZ-equivalent, but not necessarily vice-versa. In fact, CCZ-equivalence is known to be strictly more general than EA-equivalence combined with taking inverses of permutations [5].

2.4 Testing equivalence relations

In the process of searching for new APN functions, it is important to check that any newly found APN function is not equivalent to any of the previously known APN functions. Since APN functions are classified up to CCZ-equivalence, a newly discovered function is only considered to be genuinely new if it is CCZ-inequivalent to all currently known ones. However, testing whether two given functions are CCZ-equivalent is a very hard problem. The only currently known approach is by testing the permutation equivalence of linear codes. Invariants can also sometimes be used to show that two given functions are inequivalent. We outline the currently available methods for testing and deciding equivalences below.

2.4.1 Known approaches for testing equivalence

Linear codes

Given an (n,n)-function F, one can construct a matrix M_F of the form

$$M_F = \begin{pmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 0 & 1 & \alpha & \alpha^2 & \cdots & \alpha^{2^n-2} \\ F(0) & F(1) & F(\alpha) & F(\alpha^2) & \cdots & F(\alpha^{2^n-2}) \end{pmatrix},$$

where α is a primitive element of $\mathbb{F}_{2^n}$. We can then define the linear code $\mathcal{C}_F$ by using the matrix M_F as its parity-check matrix. Two linear codes C_1 and C_2 are called permutation equivalent if there is a permutation π of $\{1,2,\ldots,2^n\}$ such that $(x_1,x_2,\ldots,x_{2^n})$ is a codeword of $\mathcal{C}_1$ if and only if $(x_{\pi(1)},x_{\pi(2)},\ldots,x_{\pi(2^n)})$ is a codeword of $\mathcal{C}_2$. It can be shown that F and G are CCZ-equivalent if and only if their associated linear codes $\mathcal{C}_F$ and $\mathcal{C}_G$ are permutation equivalent [6].

The advantage of linear codes is that coding theory is an old and more developed discipline than the study of APN functions and therefore more approaches and algorithms are known. However, testing permutation equivalence takes a lot of time for dimensions $n > 8$, and testing for dimensions $n \geq 9$ can result in false negatives when the implementation runs out of memory. This means that testing functions above $n = 10$ may not be possible to do reliably using our current resources, and so these functions cannot be classified up to CCZ-equivalence directly.

One can also test EA-equivalence with linear codes, where the approach is the same, but the associated codes have a different form. Unfortunately this method has the same problems with memory, which can result in false negatives. There are two other known approaches for testing EA-equivalence. These do not go trough linear codes, but use invariants to restrict and reduce the search space when trying to find the equivalence between the two functions. Recalling that the vast majority of the known APN functions are quadratic, and that EA-equivalence is the same as CCZ-equivalence in the case of quadratic APN functions, this means that these two algorithms can effectively be used to classify quadratic APN functions up to CCZ-equivalence.

The two known algorithms use invariants to restrict the number of guesses for the functions A_1, A_2 and A from the definition of EA-equivalence. The first algorithm is due to Kaleyski, and is introduced in [7]. It is efficient for APN functions in even dimensions n. A proof of concept implementation in *Magma* is provided by Kaleyski, but an optimized implementation in a low-level programming language (such as *C*) is left as a problem for future work. Efficiently implementing this algorithm is the main focus of this thesis. Through writing a more sophisticated implementation of the algorithm in *C*, we are able to reduce the running times more than 300 times. This also has the advantage of making the implementation accessible since it does not require proprietary software such as *Magma*.

The other algorithm, due to Canteaut et al., is introduced in [8]. Canteaut's algorithm is based on an invariant called the Jacobian matrix which is used to restrict the search space; however, while it is usable for both even and odd dimensions n, it works only for quadratic functions. This shows that, while similar in principle, the two algorithms handle distinct use cases.

2.4.2 Invariants

An invariant is a property that is preserved by an equivalence relation e.g. if F and G are CCZ-equivalent, then both of them have the same value for this property. The differential uniformity Δ_F is an invariant for CCZ-equivalence, which is why CCZ-equivalence is used to classify APN functions.

The most useful aspect of invariants is that they can show that two given functions F and G are inequivalent, e.g. if a property P is an invariant under CCZ-equivalence, and if $P(F) \neq P(G)$, we can immediately conclude that F and G are not CCZ-equivalent. However, if $P(F) = P(G)$, this does not give us any useful information.

One of the benefits of invariants is that they are concrete values, so that we can pre-compute the invariants of given functions and store them. This way it is only necessary to compute the value once for each function, and then if one wants to compare a newly found function for equivalence against the known ones, one can start by computing the value of an invariant for it. If the new value is not among the known values, the function is not equivalent to any of the currently known functions. If the invariant matches some known value, one needs to conduct further equivalence tests only for those functions with the same value of the invariant.

Other algorithms

Efficient algorithms that do not use linear codes are known for some special cases of EA- and linear equivalence. For instance, the algorithm due to Biryukov et al. [9] can test linear and affine equivalence, but only if the tested functions are bijective. A suite of algorithms for testing so-called "restricted EA-equivalence" is described in [10]; unfortunately, these algorithms cannot handle the general case of linear equivalence, or any of the more general relations such as EA-equivalence or CCZ-equivalence. Finally, the algorithm due to Canteaut [8] mentioned above can test EA-equivalence between vectorial Boolean functions of any dimension, but only when the tested functions are quadratic.

Orthoderivatives

An important invariant under EA-equivalence is the so-called orthoderivative. The differential spectrum of the orthoderivative is an EA-invariant that almost always distinguishes EA-inequivalent functions [8]. An orthoderivative of an (n, m)-function F is an (n, m)-function π_F such that $\pi_F(0) = 0$ and for any $\alpha \in \mathbb{F}_{2^n} \backslash \{0\}$ and any $x \in \mathbb{F}_{2^n}$, we have $\pi_F(\alpha) \neq 0$ and

$$\pi_F(\alpha) \cdot (F(x) + F(\alpha + x) + F(\alpha) + F(0)) = 0,$$

where "$\cdot$" denotes a scalar product on $\mathbb{F}_{2^n}$.

Suppose F is a quadratic function, then it has a unique orthoderivative if and only if F is APN. If F and G are EA-equivalent quadratic APN (n, n)-functions, then their orthoderivatives, π_F and π_G, are affine equivalent but not necessarily APN [8]. Because of this, invariants that are not very useful for classifying APN functions by themselves, such as the extended Walsh spectrum and the differential spectrum, become useful for distinguishing EA-inequivalent APN functions if applied to their orthoderivatives.

The Walsh spectra and the differential spectra of the orthoderivatives take distinct values on almost all EA-inequivalent classes of quadratic APN functions, which means that the orthoderivatives can be used as an EA-equivalence test in practice. In our work, we exploit the fact that the orthoderivatives of EA-equivalent functions must be affine equivalent and use this to design an algorithm for testing EA-equivalence through the affine equivalence of the orthoderivatives. This allows us to cut down the computation time significantly (as opposed to the approach of directly testing EA-equivalence), although it does have the drawback of making it difficult to recover the exact form of the equivalence.

Γ- and Δ-rank

One drawback of the orthoderivatives is that they can only be used for quadratic APN functions. In the case of functions of higher algebraic degree, or functions that are not APN, we cannot apply orthoderivatives in order to disprove equivalence. The Γ- and Δ-rank are defined in [11] as the ranks of the incidence matrices of certain combinatorial designs that can be associated with vectorial Boolean functions. We do not go into details here since the definition is rather technical. The two ranks, however, can be computed for any vectorial Boolean function, and are invariant under CCZ-equivalence. However, computing these invariants is resource intensive: as indicated in e.g. [12], computing a Γ-rank for dimension $n = 10$ can take up to a week in some cases, while Δ-ranks tend to be even harder to compute. Furthermore, there are instances of functions having the same Γ- and Δ-rank that are not CCZ-equivalent. Thus, an equivalence test is necessary in this case to test such functions for equivalence.

For instance, functions 1.8 and 1.11 over $\mathbb{F}_{2^8}$ from the switching class representatives from [11] have the same value of the both the Γ-rank and the Δ-rank, and hence cannot be distinguished in this way. Running our EA-equivalence test, however, can establish their inequivalence in around 30 seconds.

Chapter 3

Implementation

The main goal of this thesis is to write an efficient implementation of [7], referred to as **Kaleyski's algorithm**. A proof of concept implementation in *Magma* was already developed alongside [7], but an efficient implementation of the algorithm was left as a problem for future work. As part of this master project, we realize such an implementation in *C*, and, as we observe from our computational experiments, the running times decrease more than 300 times (in some cases) as compared to the *Magma* implementation. The reduced memory requirements of our efficient implementation also make it usable in dimensions 10 and above, which was not possible with the *Magma* program. Furthermore, in the following chapter we propose a modification of Kaleyski's algorithm for testing linear and affine equivalence. We observe that by applying it to the orthoderivatives of quadratic APN functions, we can further reduce the computation time needed for deciding EA-equivalence up to 20 times (in some cases) as compared to the *C* implementation.

3.1 The original implementation

The original implementation of Kaleyski's algorithm is in the *Magma* algebra system[13] as a proof of concept. The benefit of using a high-level programming language such as *Magma* is that it includes implementations of different useful procedures and structures, making the implementation easier. One downside with *Magma* is that it is not readily available since it is proprietary. A more serious problem is that because of its high-level philosophy, the running times of the implementation are quite long; indeed, *Magma* is good for prototyping and testing ideas, but in order to obtain optimal efficiency, a lower-level programming language has to be used.

By implementing Kaleyski's algorithm in *C*, we can avoid many of the problems that occur in the original implementation, such as accessibility (the implementation can now be run on any computer and does not require special software) and memory consumption (the *Magma* implementation can run out of memory in some cases for dimensions $n \geq 10$), and we can significantly reduce the running times.

Based on our experimental results, we can see that our implementation is much more efficient in time and memory than the original *Magma* implementation. This allows it to be used efficiently for dimensions such as 10 and 12, which was impossible using the *Magma* version, and is still outside the capabilities of other algorithms such as the code isomorphism test. We give a detailed discussion and comparison of the running times for the computation of Kaleyski's implementation in *Magma* and our implementation in *C* in Section 3.2.4.

3.2 About the implementation

To be able to write an efficient implementation of Kaleyski's algorithm, an important first consideration is the programming language. As mentioned above, *Magma* is not accessible, and results in longer running times.

By choosing the *C* programming language [14], we gain more control of the memory allocation and end up with faster running times. Also, *C* is freely distributed software. Finite field arithmetic can be performed extremely quickly in *C* since we only need finite field addition, which can be implemented as bitwise XOR. However, programming in *C* does come with certain trade-offs. Since *C* is not an Object-oriented programming (OOP) language, it does not provide objects such as lists, dynamically allocated arrays, sets, and other structures that can be very useful and simplify the implementation. The language does not provide any heap or garbage collection. In practice, all of these structures, objects, and features need to be implemented from scratch. All of this has the consequence that writing an implementation in C is a much harder and more laborious process than doing so in e.g. *Magma*. Nonetheless, we believe that the improvement in running times is ultimately worth it.

In the following, we give an overview of the structure of our implementation, including a brief description of the most important auxiliary structures and functions. For more details, the reader can refer directly to our implementation which is available at [15].

3.2.1 Structures

Since C is not an OOP language, most of the structures (such as sets, multi-sets, linked lists, etc.) need to be implemented from scratch. All of the implemented structures are defined in the header file *structures.h*. Here one can find implementations of the structures *TruthTable*, *Partition*, *WalshTransform* and *Linkedlist*.

One of the most fundamental and widely used structures in our implementation is the *TruthTable* which is used to represent a vectorial Boolean function. The *TruthTable* needs to store two things: the *elements* of a function F, and the dimension, n, of F. The definition of this structure is given in Listing 3.1 below.

Listing 3.1: Structure *TruthTable*

```
typedef struct TruthTable {
  size_t n; // Dimension of the function
  size_t *elements; // All the elements of the function
} TruthTable;
```

The *TruthTable* needs to allocate enough memory so that it can hold all the 2^n elements. When memory is allocated, it is important to deallocate it at some point, preferably as soon as possible. In addition, truth tables will be typically stored in text files, and need to be converted into the internal representation shown in Listing 3.1. In order to make all of this easier, the same header file contains the following auxiliary functions:

- *initTruthTable*: allocates enough memory to store 2^n elements and sets the dimension, n;

- *parseTruthTable*: reads a file which contains a truth table and stores the information in a structure of the type *TruthTable*. In this case, the file has to be of the following form: the first line of the file contains a single number, which is the dimension n. The second line contains 2^n integers separated by spaces which represent the values of the function F on $F(0), F(1), F(2), \ldots, F(2^n - 1)$. We recall that the elements of $\mathbb{F}_{2^n}$ can be represented as integers between 0 and $2^n - 1$ by identifying their coordinate vectors with the binary expansion of integers.
- *printTruthTable*: given a truth table, prints it out to the console. The first line represents the dimension n and the second line contains all 2^n values.
- *destroyTruthTable*: frees the memory allocated to a *TruthTable*.

Another structure that we implement is *Partition* which stores a partition of $\mathbb{F}_{2^n}$. Recall that in order to compute the list of possible outer permutations L_1, we partition the field $\mathbb{F}_{2^n}$ into "buckets" with respect to the multiplicities of certain elements. The *Partition* structures must therefore contain the number of elements in each "bucket", the multiplicities of the elements of $\mathbb{F}_{2^n}$, and the total number of "buckets". For instance, for the Gold function in dimension 6, *Partition* holds 3 "buckets", with the multiplicities 12160, 4992 and 3456. The number of elements in each "bucket" is 1, 21 and 42, respectively. The definition of this structure is given below in Listing 3.2.

Listing 3.2: Structure *Partition*

```c
typedef struct Partition {
  size_t numberOfBuckets;
  size_t *multiplicities;
  size_t *bucketSizes;
  size_t **buckets;
} Partition;
```

We define several auxiliary functions for this structure:

- *initPartition*: allocates the memory needed to represent the structure;
- *partitionFunction* (described in more detail below): given a function F (represented as a truth table), computes the partition of $\mathbb{F}_{2^n}$ into "buckets" and stores it in the structure;

- *printPartition*: given a partition, prints it out to the console; the first line is the number of "buckets" n, and each of the following n lines lists all elements in the corresponding bucket;

- *destroyPartition*: deallocates the memory used by the structure.

The program also uses linked lists, and since C does not include its own implementation, we also need to implement these. The program uses two different kinds of linked lists: *Node* with *integers* as values, and *TtNode* with structures of the type *TruthTable* as values. Because of this, there are two different linked list implementations with different functions. The definition of the structures *Node* and *TtNode* is given below in Listing 3.3 and Listing 3.4, respectively.

Listing 3.3: Structure *Node*

```
typedef struct Node {
  size_t data;
  struct Node *next;
} Node ;
```

Listing 3.4: Structure *TtNode*

```
typedef struct TtNode {
  TruthTable *data;
  struct TtNode *next;
} TtNode ;
```

Node and *TtNode* contain different auxiliary functions, which essentially do the same, except for different structures:

- *initNode*/*initTtNode*: allocates the memory needed to represent the structure;

- *addNode*/*addTtNode*: given the head of a linked list, and the data (either an integer or a truth table), adds a new node to the linked list;

- *countNodes*/*countTtNodes*: given the head of a linked list, counts all nodes and returns their number;

- *printNodes*/*printTtNodes*: given the head of linked list, prints the values to the console. Each line represents a node;

- *destroyNode*/*destroyTtNode*: deallocates the memory used by the structure.

3.2.2 Functions

The implementation of Kaleyski's algorithm needs several functions. These include *partitionFunction*, *mappingOfBuckets*, *outerPermutation* and *innerPermutation*. All of these functions are described below.

partitionFunction

The *partitionFunction* takes a *TruthTable* F and an integer k as input. To partition a function F, we find the multiplicity of all elements of $\mathbb{F}_{2^n}$ where the multiplicity of an element $e \in \mathbb{F}_{2^n}$ is defined as the number of k-tuples $(x_1, x_2, \ldots, x_k) \in \mathbb{F}_{2^n}^k$ such that

$$x_1 + x_2 + \cdots + x_k = 0$$

and

$$F(x_1) + F(x_2) + \cdots + F(x_k) = e.$$

The multiplicities are then grouped into "buckets", such that all the elements e with the same multiplicity are in the same bucket. Since the size of k is not fixed, the calculation of the multiplicities is implemented as a recursive function of depth k. When the recursion is done, it returns a *Partition* with all the data that has been computed.

Obviously, the time needed for computing the multiplicities as described above grows exponentially with k. In order to avoid this exponential growth, we also implement a variant of the above procedure which computes the multiplicities of the elements $e \in \mathbb{F}_{2^n}$ using the Walsh transform, as described in Proposition 4 of [7]. More precisely, the multiplicity M of an element $e \in \mathbb{F}_{2^n}$ can be expressed as

$$2^{2n} M = \sum_{a,b \in \mathbb{F}_{2^n}} (-1)^{\mathrm{Tr}(be)} W_F^k(a, b).$$

mappingOfBuckets

We know that if F and G are EA-equivalent via $L_1 \circ F \circ A_2 + A = G$, then L_1 must map a bucket from the partition under F to a bucket of the same size from the partition under G. However, it is possible that the partitions contain multiple buckets of the same size, and then it is not clear a priori which bucket should be the image of a given bucket under L_1. Therefore, after the program has calculated the partitions of two functions F and G, it is necessary to guess the mapping of the buckets B_F of F to the buckets B_G of G. For each of these mappings, we find all linear permutations L_1 that respect the bucket partition, and map buckets under F to buckets under G in the specified manner. If for some such L_1, we manage to find a valid A_2 (that is, an A_2 which correctly defines an EA-equivalence between F and G), we can immediately terminate with success. Otherwise, we proceed to the next choice of L_1 under the current mapping of buckets; or to the next mapping of buckets, if we have exhausted all choices of L_1. An example of two functions F and G with multiple buckets of the same size in their partitions is given in Figure 3.1, with F being the function $F(x) = x^3 + \alpha^{17}(x^{17} + x^{18} + x^{20} + x^{24})$ in dimension $n = 6$, where α is a primitive element of $\mathbb{F}_{2^6}$. The arrows in the figure illustrate all the possible mappings for any given bucket; for instance, the first bucket in the partition under F can map to the first, the fourth, or the last bucket in the partition under G.

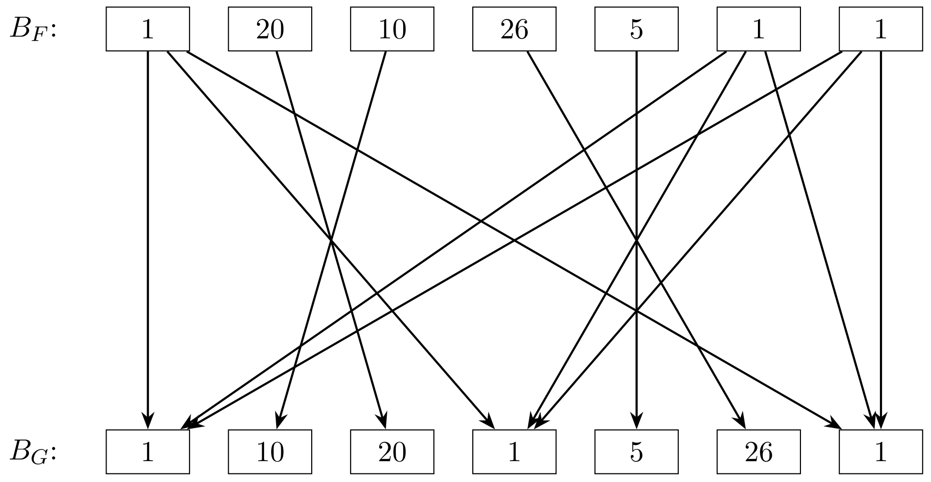

Figure 3.1: Map between B_F and B_G with multiple buckets of same sizes.

outerPermutation

Having computed the *Partition* of F and G, and selected a *mappingOfBuckets*, we now find all outer permutations, L_1, of $\mathbb{F}_{2^n}$ that respect them. Since L_1 is a linear function, it is enough to guess its values on a basis of $\mathbb{F}_{2^n}$. All the permutations L_1 are therefore found by a recursive search which guesses their values on a *basis*. We simply use the standard basis, $\{2^0, 2^1, 2^2, \dots, 2^{n-1}\}$. The search procedure is essentially a depth first search (DFS) with backtracking. When guessing the value $L_1(b)$ of a basis element b, we first find the bucket under F to which b belongs; then we know that $L_1(b)$ must belong to its corresponding bucket under G, although we do not know which element from that bucket it is. We thus make guesses, and try out all possibilities. Having guessed the values of e.g. b and b' under L_1, we can also derive the value $L_1(b + b')$ of $b + b'$ using the linearity of L_1. This can lead to contradiction, if $b + b'$ belongs to a certain bucket under F, while $L_1(b + b')$ does not belong to its corresponding bucket under G. This means that when we reach a point where such a contradiction occurs, we backtrack and try a different guess, until all possible combinations have been tested. Once all basis elements have been assigned a value, we can reconstruct the entire truth table of L_1, and either store it, or directly search for a corresponding inner permutation as described below.

In our main implementation, we do not store the outer permutations L_1 in memory, but instead process each one of them using the *innerPermutation* procedure described below until an equivalence is found, or all possibilities have been exhausted. In this way, our implementation ends up with negligible memory requirements since practically no information has to be stored.

We note that the original implementation in *Magma* takes a different approach, and first computes a list of all the outer permutations L_1, before processing each one separately. This might be useful if one is interested in the exact number or form of the outer permutations, but is not optimal from the point of view of efficiency: for instance (as we observe in our experimental results), it is possible that already the first few outer permutations L_1 that we find will provide an inner permutation giving an EA-equivalence between the tested functions; computing other outer permutations is then wasteful. Furthermore, this is the reason that the *Magma* implementation would often run out of memory when testing EA-equivalence for dimensions 10 and above.

Nonetheless, we implement a variation of the algorithm in which we do compute all outer permutations first (just like in the *Magma* implementation), and then process them one by one. In this case, instead of being directly processed by *innerPermutation*, the outer permutations found by the DFS are stored in a linked list. This version allows us to give a detailed breakdown of the running times, and an exact comparison of the efficiency as compared to the *Magma* implementation. We discuss this in more detail in Section 3.2.4.

innerPermutation

For each L_1 found by *outerPermutation* we apply the inverse L_1^{-1} to both sides of the relation $L_1 \circ F \circ A_2 + A = G$ and obtain $F \circ A_2 + A' = G'$. For each L_1, the goal is now to find an A_2. Once we have guessed A_2, we can compute A' as $A' = F \circ A_2 + G'$; if A' is affine, then the pair (L_1, A_2) that we have selected gives us a valid EA-equivalence between F and G.

The function *innerPermutation* takes in some parameters that are needed, such as a function F, a function G', an empty *TruthTable* A_2 and an empty *TruthTable* A. Once again, we guess A_2 by guessing its values on the standard basis. In order to do this efficiently, we first compute restricted *domains* for the basis elements; the *domain* of an element b is the set of possible values that $A_2(b)$ can take. To reduce the domain of $x \in \mathbb{F}_{2^n}$ (which is initially set to $\mathbb{F}_{2^n}$), we compute (using the notation from [7]) the intersection of all the sets $O_3^F(t)$ of all triples (x_1, x_2, x_1+x_2) with $F(x_1)+F(x_2)+F(x_1+x_2) = t$, for all values of t that can be expressed as $G(x_1) + G(x_2) + G(x_1 + x_2)$ with $x \in \{x_1, x_2, x_1 + x_2\}$ (see the discussion in [7, pp. 284-285]). In order to compute intersections, we use Boolean maps, i.e. a Boolean array $W[x]$ indexed by $x \in \mathbb{F}_{2^n}$, such that $W[x] = 1$ if and only if x belongs to the set under consideration. Computing the intersection of two sets can then be naturally expressed using bitwise "and".

Having computed the reduced domains, we use a DFS to guess the inner permutation A_2 and reconstruct $A = F \circ A_2 + G$ and check if A is affine. If the reconstructed A is affine, we have found an equivalence between F and G, and can terminate with success.

When the program is done, it prints the functions L_1, A_2 and A to the console. If the program has not found an affine function such that $L_1 \circ F \circ A_2 + A = G$, the output will remain empty. Finally, the program frees all the allocated memory before it terminates.

3.2.3 How to use the program

Our implementation is provided on GitHub [15]. The easiest way to use the program is to clone the GitHub project and run the *compile.sh* script which will create the executable, *ea*.

The *ea* program tests and recovers the EA-equivalence between two given functions F and G. It expects at least one argument, the function F given as a truth table. In the case that only a single function is given, the program generates a random function EA-equivalent to F and compares it against F for equivalence (for testing purposes). By running the program with the flag "-h" the program will output all options to run the program, which will look something like this:

Listing 3.5: Running *./ea* with help

```
> ./ea -h
EA-equivalence test
Usage: ea [ea_options] [filename_F] [filename_G]
Ea options:
  -h   - Print help
  -k   - Size of k
  -t   - Add this for printing running times
        for different functions.

  filename_F: path to the file of a function F
  filename_G: path to the file of a function G
```

When giving the program the path to two functions F and G it will search for EA-equivalence between these two. If the search is successful, the program will print L_1, A_2 and A.

An example of running the program is given below:

Suppose that the file *f.tt* contains the function F:

```
6
0 1 8 15 27 14 35 48 53 39 43 63 47 41 1 1 41 15 15 47 52 6
34 22 20 33 36 23 8 41 8 47 36 52 35 53 35 39 20 22 33 34 48
53 39 48 6 23 22 33 63 14 23 52 14 43 27 63 36 6 27 43 20 34
```

and the file *g.tt* contains the function G:

```
6
0 31 12 49 26 49 4 13 45 26 58 47 59 56 62 31 42 48 62 6 27 53
29 17 23 37 24 8 42 44 55 19 44 14 30 30 34 52 2 54 18 24 59
19 16 46 43 55 0 39 42 47 37 54 29 44 46 33 31 50 7 60 36 61
```

Then the EA-equivalence test between these two functions can be run as follows:

Listing 3.6: Example of output from the program

```
>./ea f.tt g.tt
L1:
6
0 3 1 2 38 37 39 36 32 35 33 34 6 5 7 4 9 10 8 11 47 44
    46 45 41 42 40 43 15 12 14 13 58 57 59 56 28 31 29
    30 26 25 27 24 60 63 61 62 51 48 50 49 21 22 20 23
    19 16 18 17 53 54 52 55

A2:
6
0 52 50 6 40 28 26 46 4 48 54 2 44 24 30 42 43 31 25 45
    3 55 49 5 47 27 29 41 7 51 53 1 39 19 21 33 15 59
   61 9 35 23 17 37 11 63 57 13 12 56 62 10 36 16 22
   34 8 60 58 14 32 20 18 38

A:
6
0 50 59 9 35 17 24 42 6 52 61 15 37 23 30 44 60 14 7 53
    31 45 36 22 58 8 1 51 25 43 34 16 2 48 57 11 33 19
    26 40 4 54 63 13 39 21 28 46 62 12 5 55 29 47 38
   20 56 10 3 49 27 41 32 18
```

3.2.4 Computational results

As discussed above, we have implemented the EA-equivalence test in two different ways. In the first one, we compute all outer permutations L_1 first, and then process them one by one. This is done so that it will exactly mirror the *Magma* implementation, and allow us to objectively compare the improvements in running time. This also allows us to estimate the worst-case running times by noting the number of outer permutations and the time necessary for processing each of them. Our observations are presented in Table 3.1.

The first column of the table gives the dimension n of $\mathbb{F}_{2^n}$. The functions are indexed in the second column in the same way as in [7]. The next 4 columns under “Kaleyski’s implementation” and the next 4 columns under “This implementation” give the time in seconds for computing the partitions of $\mathbb{F}_{2^n}$ according to the quadruple sums as explained in [7] and using the Walsh transform, respectively. The following column gives the time for computing all outer permutations L_1 preserving the corresponding partition. The last column gives the time for reconstructing the inner permutation L_2.

We observe that the efficient implementation in *C* as a whole is much faster than the proof of concept implementation in *Magma*. The running time for computing the partition using the quadruples as described in [7] is much faster in *C*. For example, for function 1.3 for $n = 8$, we observe that the runtime for partitioning in *C* is over 500 times faster than in *Magma*. We also observe that using the Walsh transform, the runtime for the same function is over 200 times faster (using the Walsh transform should be faster if we are using k-tuples with $k > 4$ instead of quadruples in the partition, or if we have the Walsh transform precomputed since the running times given here include the time for computing the Walsh transform from the truth table of the function). For finding all outer permutations L_1, we observe that for most of the cases the runtime is much faster than in *Magma*. For instance, for function 1.2 for $n = 8$, we observe that the runtime is about 16 times faster in *C*. However, in some cases we observe that *Magma* is faster, such as for functions 1.5 and 1.6 for $n = 8$, where the runtime in *Magma* is almost 2 times faster. It is also worth mentioning that we have been able to run the implementation for functions with $n = 10$, where *Magma* runs out of memory (this is because the proof of concept implementation was designed in such a way that it would compute all outer permutations first, storing everything in memory, and only then processing them one by one).

However, in practice it is not necessary (and not optimal) to compute and store all outer permutations in advance. Instead, a more time and memory efficient approach is to try and find an inner permutation for an outer permutation L_1 as soon as L_1 is found by the DFS. In this way, we do not need to allocate any memory for storing the outer permutations, and if

		Kaleyski's implementation				This implementation			
n	ID	Sums	Walsh	L_1	L_2	Sums	Walsh	L_1	L_2
6	1.1	1.650	1.250	1.030	0.308	0.011	0.028	0.076	0.006
	1.2	1.510	1.390	0.300	0.337	0.008	0.017	0.050	0.005
	2.1	1.390	1.450	0.010	0.134	0.006	0.017	0.004	0.008
	2.2	1.250	1.250	0.380	0.419	0.006	0.017	0.041	0.006
	2.3	1.240	1.450	0.970	0.405	0.009	0.017	0.070	0.006
	2.4	1.260	1.250	0.010	0.283	0.010	0.018	0.010	0.007
	2.5	1.300	1.310	0.050	0.317	0.007	0.022	0.001	0.084
	2.6	1.260	1.290	0.010	0.316	0.007	0.022	0.006	0.008
	2.7	1.310	1.290	0.010	0.352	0.007	0.028	0.003	0.008
	2.8	1.310	1.310	0.010	0.326	0.007	0.021	0.002	0.010
	2.9	1.300	1.310	0.010	0.262	0.008	0.019	0.010	0.008
	2.10	1.580	1.300	0.010	0.317	0.009	0.018	0.030	0.008
	2.11	1.290	1.290	0.000	0.302	0.012	0.019	0.001	0.010
	2.12	2.450	2.470	0.030	0.524	0.007	0.017	0.002	0.004
8	1.1	103.580	74.910	23.090	0.935	0.426	1.015	3.603	0.374
	1.2	92.140	86.570	206.830	0.955	0.439	1.013	12.674	0.338
	1.3	244.540	238.560	78.180	1.432	0.430	1.022	15.736	0.376
	1.4	146.520	140.710	12.530	1.160	0.431	1.019	16.736	0.377
	1.5	112.860	107.580	58.300	1.036	0.438	1.020	112.924	0.326
	1.6	111.810	106.920	62.580	1.015	0.439	1.020	116.904	0.327
	1.7	127.330	121.320	10.020	1.084	0.436	1.011	7.316	0.360
	1.8	126.210	121.740	26.670	1.065	0.433	1.011	36.422	0.351
	1.9	127.250	121.730	40.370	1.007	0.442	1.017	54.936	0.348
	1.10	127.090	121.270	10.400	1.068	0.443	1.019	6.379	0.384
	1.11	127.410	122.560	50.560	1.083	0.432	1.014	24.540	0.370
	1.12	127.950	121.240	46.520	1.075	0.438	1.019	18.709	0.355
	1.13	127.850	122.320	10.530	1.083	0.454	1.019	16.742	0.366
	1.14	132.900	127.100	0.010	1.070	0.439	1.015	0.001	0.754
	1.15	126.410	121.940	22.580	1.086	0.430	1.014	106.083	0.353
	1.16	127.020	121.040	9.970	1.070	0.439	1.023	9.615	0.360
	1.17	126.860	120.790	69.860	1.027	0.432	1.027	6.843	0.394
	2.1	99.690	94.340	27.380	56.611	0.435	1.011	59.920	0.374
	3.1	118.870	112.990	57.480	1.042	0.438	1.018	29.217	0.335
	4.1	115.700	110.040	0.070	40.200	0.437	1.016	0.003	0.696
	5.1	102.470	96.640	0.030	1.016	0.430	1.024	3.699	0.669
	6.1	110.940	105.610	0.040	0.980	0.434	1.013	0.002	0.520
	7.1	98.650	93.330	49.350	132.942	0.432	1.022	12.565	0.320

Table 3.1: Observed running times for *Magma* and *C*

we find an outer permutation which allows us to reconstruct an equivalence between the tested functions, then we do not even need to search for the rest of the outer permutations. One would almost always use this more efficient version in practice, except possibly if the exact list of outer permutations is needed for some reason, or if the computation needs to be carried out in separate phases.

Running times for this more efficient implementation are given in Table 3.2. The third and fourth column, respectively, give the total running time for the experimental implementation described above (which computes all outer permutations first) and the observed running times for our main implementation, while the last column gives the number of all outer permutations. As we can see, the running times are significantly faster, and the procedure can even be used for dimensions such as $n = 12$, where verifying that the Gold function x^3 is EA-equivalent to a randomly generated function equivalent to it takes around 4535 seconds. We note that the running times for this implementation may depend a bit on how "lucky" we are when searching for the outer permutations; in other words, if we find an outer permutation L_1 first which does lead to an EA-equivalence between the tested functions, then the running time will obviously be quite short since we will find the EA-equivalence almost immediately. On the other hand, if we have to go through many outer permutations before we find one that corresponds to an EA-equivalence, the running time may be much longer. As Table 3.2 illustrates, however, the running times in practice appear to be quite satisfactory. As already discussed above, this improved implementation also has minimalistic memory requirements, and so can be used in high dimensions such as 12 or even 14, where none of the currently available methods for testing EA-equivalence can be used.

n	ID	Finding all L_1	Main implementation	Permutations
8	1.1	13.403	1.004	680
	1.2	13.451	0.954	680
	1.3	16.542	4.668	8
	1.4	17.545	0.952	8
	1.5	113.688	0.935	4
	1.6	117.670	6.623	4
	1.7	8.113	7.495	1
	1.8	37.206	29.939	4
	1.9	55.727	29.870	4
	1.10	7.206	3.602	2
	1.11	25.342	16.125	4
	1.12	19.503	15.994	4
	1.13	17.563	3.404	2
	1.14	1.194	11.034	2
	1.15	106.866	9.197	1
	1.16	10.414	1.977	2
	1.17	7.669	1.192	2
	2.1	61.892	40.525	360
	3.1	29.991	5.150	4
	4.1	2.011	3.888	16
	5.1	4.799	9.405	8
	6.1	0.957	7.801	8
	7.1	13.317	2.057	680
10	1.1	17225.140	62.031	3410
	1.2	17114.651	73.068	3410
	1.5	35158.835	436.070	155
	1.6	39467.687	495.447	155

Table 3.2: Observed running times for finding all L_1 by computing all L_1 first, and by processing each L_1 directly

Chapter 4

Testing Linear and Affine equivalence

In this chapter we adapt Kaleyski's algorithm to a procedure for deciding linear equivalence between two vectorial Boolean functions. We also describe how the approach can be generalized to handle affine equivalence. Finally, we provide an efficient C implementation of the algorithm.[1]

4.1 Introduction

Recall that an orthoderivative of an (n, m)-function F is an (n, m)-function π_F such that $\pi_F(0) = 0$ and for any $\alpha \in \mathbb{F}_{2^n} \backslash \{0\}$ and any $x \in \mathbb{F}_{2^n}$, we have $\pi_F(\alpha) \neq 0$ and $\pi_F(\alpha) \cdot (F(x) + F(\alpha + x) + F(\alpha) + F(0)) = 0$. Recall also that any quadratic APN function has a unique orthoderivative [8]. If two quadratic APN (n, n)-functions F and G are EA-equivalent via $A_1 \circ F \circ A_2 + A = G$, then $(L_1^*)^{-1} \circ \pi_F \circ L_2 = \pi_G$, where $L_1(x) = A_1(x) + A_1(0)$, $L_2(x) = A_2(x) + A_2(0)$, and L_1^* is the adjoint of L_1. By using this observation we investigate how to test EA-equivalence between F and G through the linear equivalence between π_F and π_G.

[1]An extended abstract describing this work has been submitted to the 7th International Workshop on Boolean Functions and Their Applications (BFA) 2022.

The varied structure of the orthoderivatives that is observed in [8] suggest that comparing the orthoderivatives of F and G for equivalence (rather than F and G themselves), might be more efficient as a test for EA-equivalence. Since there is currently no known efficient method for testing linear or affine equivalence, we design a natural algorithm for this and show that by applying it to the orthoderivatives, the time needed for verifying EA-equivalence is cut down significantly (up to twenty times in some cases) as compared to our main C implementation described in the previous chapter.

4.2 Algorithm for testing linear equivalence

Recall that two (n, m)-functions F and G are linear equivalent if $L_1 \circ F \circ L_2 = G$. Similarly to the algorithm from [7], we first restrict the set of possible L_1, and then, for each guess of L_1 we compose both sides of the relation $L_1 \circ F \circ L_2 = G$ with the inverse, L_1^{-1}, and try to reconstruct L_2. We reduce the possible choices of L_1 by partitioning $\mathbb{F}_{2^n}$ into "buckets" with respect to the pre-images under F and under G. In this way, we write $\mathbb{F}_2^n = B_0^F \cup B_1^F \cup \cdots \cup B_{2^n}^F$, where $B_i^F = \{x \in \mathbb{F}_2^n : \#F^{-1}(x) = i\}$; and similarly, $\mathbb{F}_2^n = B_0^G \cup B_1^G \cup \cdots \cup B_{2^n}^G$. The sizes of B_i^F and B_i^G for any i must be the same, otherwise we already have reached a contradiction, and can conclude that F and G are linear-inequivalent. Otherwise, we know that the image of any B_i^F under L_1 must be B_i^G. By using a DFS we find all L_1 that satisfy this condition. The number of L_1 is sufficiently small for us to test all of them. For instance, for $n = 8$ we only get several hundreds possibilities for L_1 in the worst case. The pseudo-code for this procedure is given in Algorithm 1.

This algorithm can be easily adapted to test affine equivalence, i.e $A_1 \circ F \circ A_2 = G$ between functions F and G. Here $A_1 = L_1 + c_1$ and $A_2 = L_2 + c_2$ for some constants c_1, c_2 and L_1, L_2 linear, and so it suffices to go trough all choices of c_1 and c_2, and run the linear-equivalence algorithm for each of them. With the following observation, we can reduce the number of choices.

Algorithm 1: Reconstructing the outer permutation L_1 in $L_1 \circ F \circ L_2 = G$

```
Input  : Two $(n, m)$-functions $F$ and $G$
Output: All linear permutations $L_1$ of $\mathbb{F}_2^m$ respecting the partitions
        induced by $F$ and $G$
Partition $\mathbb{F}_2^m = B_0^F \cup \cdots \cup B_{2^m}^F = B_0^G \cup \cdots \cup B_{2^m}^G$ ;
if $(\exists i)(\#B_i^F \neq \#B_i^G)$ then
    return $\emptyset$
end
Let $\mathcal{B} = \{b_1, b_2, \ldots, b_m\}$ be a basis of $\mathbb{F}_2^m$ ;
return Guess ($\mathcal{B}$, 1, $\emptyset$) # recursively guess the values of $L_1$ on $\mathcal{B}$
Function Guess($\mathcal{B}$, $i$, $\mathcal{L}$):
    if $i = n + 1$ then
        reconstruct $L_1$ from its values on $\mathcal{B}$ ;
        return $\mathcal{L} \cup \{L_1\}$ ;
    end
    Let $j$ be such that $b_i \in B_j^F$ ;
    for $y \in B_j^G$ do
        $L_1(b_i) \leftarrow y$ ;
        # Check all currently known values of $L_1$ for contradiction
        $contradiction \leftarrow$ false ;
        for $x \in \mathrm{Span}(b_1, b_2, \ldots, b_i)$ do
            let $j, k$ be such that $x \in B_j^F$, $L_1(x) \in B_k^G$ ;
            if $k \neq j$ then
                $contradiction \leftarrow$ true ;
                break ;
            end
        end
        if $contradiction =$ false then
            $\mathcal{L} \leftarrow \mathcal{L} \cup \mathrm{Guess}(\mathcal{B}, i + 1, \mathcal{L})$ ;
        end
    end
    return $\mathcal{L}$ ;
```

We assume that $F(0) = G(0)$ for simplicity. First, we observe that $\#F^{-1}(F(c_2)) = \#G^{-1}(0)$. Indeed, from $c_1 + L_1(F(L_2(x) + c_2)) = G(x)$, by substituting $x = 0$ we get $L_1(F(c_2)) + c_1 = 0$; then if $x \in \mathbb{F}_2^n$ with $F(x) = F(c_2)$, then $G(x) = L_1(F(c_2)) + c_1 = 0$ as well. Thus, it is enough to consider c_2 with $\#F^{-1}(c_2) = \#G^{-1}(0)$.

4.3 Implementation and experimental results

We have implemented the modified algorithm in C and made it available on GitHub [16]. The implementation allows for several use cases:

(i) Testing two functions for linear equivalence;

(ii) Testing two functions for affine equivalence;

(iii) Testing two quadratic APN functions for EA-equivalence through the linear-equivalence of their orthoderivative.

We give some running times for case (iii) and compare them with the running times for the efficient implementation of Kaleyski's algorithm.

Since our linear equivalence algorithm is an adaptation of Kaleyski's algorithm, we can reuse the structures and functions from [15] and simply recombine them in order to implement this new algorithm. In fact, this procedure is simpler to implement; there is no need for the k-tuple sums for partitioning the functions (and so no need for a recursive function). We partition the field into buckets according to the multiplicities of the images of F, which is not only simpler but also faster (since instead of k nested loops we only need one). There is no need to create maps of the buckets since we know a priori which bucket under F will map to which bucket under G (these are uniquely defined by the corresponding multiplicities). After we have created the partitions, we proceed in the same way as in [7], finding all possible outer permutations L_1 and trying to find an inner permutation A_2.

For testing the efficiency of the approach for deciding EA-equivalence through the orthoderivatives, we generate a random triple (A_1, A_2, A) for some of the known APN functions F from [11] for $n = \{6, 8\}$ and from the CCZ-inequivalent representatives for $n = 10$. We then construct $G = A_1 \circ F \circ A_2 + A$ and apply our algorithm to π_F and π_G. For each choice of F, we generate 10 triples (A_1, A_2, A) and give the average running time for finding an EA-equivalence relation. We compare the running time of this procedure with the efficient implementation of Kaleyski's algorithm. The results are summarized in Table 4.1 given in seconds.

n	ID	Main implementation	This implementation
8	1.1	1.004	0.252
	1.2	0.954	0.289
	1.3	4.668	1.138
	1.4	0.952	1.794
10	1	62.031	20.338
	2	73.068	15.728
	5	436.070	17.977
	6	495.447	91.324

Table 4.1: Running times of Kaleyski's algorithm and this algorithm.

As we can see from Table 4.1, this approach always cuts down the computation time (except for function 1.4 in $n = 8$), and the effect is particularly pronounced in high dimensions such as $n = 10$, where the running time is approximately 24 times faster than using the efficient implementation of [7] for function 5.

However, the downside of this method is that knowing the linear equivalence $L'_1 \circ \pi_F \circ L'_2 = \pi_G$ between π_F and π_G does not allow us to easily reconstruct the EA-equivalence $A_1 \circ F \circ A_2 + A = G$ between F and G. As observed in Proposition 36 of [8], if F and G are EA-equivalent via $A_1 \circ F \circ A_2 + A = G$ then their orthoderivatives are linear equivalent via $(L_1^*)^{-1} \circ \pi_F \circ L_2 = \pi_G$, where L_1 and L_2, are the linear parts of A_1 and A_2, respectively. However, this is not the only one possible equivalence between π_F and π_G; in fact there are many other such pairs of (L'_1, L'_2) satisfying the equation $L'_1 \circ \pi_F \circ L'_2 = \pi_G$. In the worst case, we will have to run the algorithm until it find all pairs (L'_1, L'_2) with $L'_1 \circ \pi_F \circ L'_2 = \pi_G$ before we are able to recover the EA-equivalence between F and G. We have tested this computationally for some APN functions in $n = 6$, and we observe that we do find the pair (L_1, L_2) originating from the EA-equivalence between F and G, but this is the only pair corresponding to an EA-equivalence. Therefore, if the exact form of the EA-equivalence between F and G is needed, it would be better to compute it using Kaleyski's algorithm or Canteaut's algorithm from [8].

Chapter 5

Conclusion

In this thesis we have efficiently implemented in C a test for EA-equivalence between vectorial Boolean functions, and showed that it can be used to reduce the running time for testing EA-equivalence significantly (as opposed to the existing implementation of the same algorithm). We have showed that the new implementation can check for EA-equivalence for dimension $n = 10$ and above, overcoming some of the memory issues of the existing implementation. This approach can not only decide whether two given functions are EA-equivalent, but it can recover the exact form of the EA-equivalence between them. The proposed method can work for any pair of functions (of any algebraic degree and any differential uniformity), although as observed in [7], it is not efficient in the case of e.g. AB functions. We provide the implementation as open source software online.

We have also developed a new algorithm for deciding linear-equivalence and affine equivalence between vectorial Boolean functions, and have shown that it can be used to reduce the computation time for checking EA-equivalence between quadratic APN functions significantly. We have provided an efficient implementation in C of this algorithm as well, and made it available online as open source software.

Unlike the approach of using invariants, our algorithms can prove that two functions are EA- or linear-equivalent (as opposed to only being able to disprove it), and in the case of Kaleyski's original algorithm, we can recover the exact form of the equivalence (if it exists).

Bibliography

[1] Eli Biham and Adi Shamir. Differential cryptanalysis of DES-like cryptosystems. *Journal of Cryptology*, 4(1):3–72, January 1991. ISSN 1432-1378. doi: 10.1007/BF00630563.

[2] Claude Carlet. *Boolean Functions for Cryptography and Coding Theory*. Cambridge University Press, first edition, November 2020. ISBN 978-1-108-60680-6 978-1-108-47380-4. doi: 10.1017/9781108606806.

[3] Claude Carlet, Pascale Charpin, and Victor Zinoviev. Codes, bent functions and permutations suitable for des-like cryptosystems. *Designs, Codes and Cryptography*, 15(2):125–156, 1998.

[4] Satoshi Yoshiara. Equivalences of quadratic APN functions. *Journal of Algebraic Combinatorics*, 35(3):461–475, May 2012. ISSN 0925-9899, 1572-9192. doi: 10.1007/s10801-011-0309-1.

[5] L. Budaghyan, C. Carlet, and A. Pott. New classes of almost bent and almost perfect nonlinear polynomials. *IEEE transactions on information theory*, 52(3):1141–1152, 2006. ISSN 0018-9448. doi: 10.1109/TIT.2005.864481.

[6] Yves Edel and Alexander Pott. On the Equivalence of Nonlinear Functions. *Enhancing Cryptographic Primitives with Techniques from Error Correcting Codes*, pages 87–103, 2009. doi: 10.3233/978-1-60750-002-5-87.

[7] Nikolay Kaleyski. Deciding EA-equivalence via invariants. *Cryptography and communications*, 14(2):271–290, 2021. ISSN 1936-2447. doi: 10.1007/s12095-021-00513-y.

[8] Anne Canteaut, Alain Couvreur, and Léo Perrin. Recovering or Testing Extended-Affine Equivalence. *IEEE Transactions on Information Theory*, pages 1–1, 2022. ISSN 1557-9654. doi: 10.1109/TIT.2022.3166692.

[9] Alex Biryukov, Christophe De Cannière, An Braeken, and Bart Preneel. A Toolbox for Cryptanalysis: Linear and Affine Equivalence Algorithms. In *Advances in Cryptology — EUROCRYPT 2003*, Lecture Notes in Computer Science, pages 33–50, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-14039-9. doi: 10.1007/3-540-39200-9_3.

[10] Lilya Budaghyan and Oleksandr Kazymyrov. Verification of restricted ea-equivalence for vectorial boolean functions. In *International Workshop on the Arithmetic of Finite Fields*, pages 108–118. Springer, 2012.

[11] Yves Edel and Alexander Pott. A new almost perfect nonlinear function which is not quadratic. *Advances in Mathematics of Communications*, 3(1):59–81, 2009. ISSN 1930-5338. doi: 10.3934/amc.2009.3.59.

[12] Nikolay S. Kaleyski. Invariants for EA- and CCZ-equivalence of APN and AB functions. *Cryptography and Communications*, 13(6):995–1023, 2021.

[13] Wieb Bosma, John Cannon, and Catherine Playoust. The Magma Algebra System I: The User Language. *Journal of Symbolic Computation*, 24(3-4):235–265, September 1997. ISSN 07477171. doi: 10.1006/jsco.1996.0125.

[14] Brian W. Kernighan. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, N.J, 1978. ISBN 978-0-13-110163-0.

[15] Marie Heggebakk. heggebakk/ea-equivalence: Computationally testing ea-equivalence. https://github.com/heggebakk/ea-equivalence, 2022.

[16] Marie Heggebakk. heggebakk/affine: Testing Affine functions. https://github.com/heggebakk/affine, 2022.