University of Bergen
Department of Informatics

Master thesis in Secure and Reliable Communication

On self-equivalences of APN functions

Author: Vegard Jensløkken
Supervisor: Nikolay Stoyanov Kaleyski

UNIVERSITETET I BERGEN
Det matematisk-naturvitenskapelige fakultet

Abstract

In this thesis we investigate the structure of what we call extended linear self-equivalences for vectorial Boolean functions. That is, (L_1, L_2, L) such that $L_1 \circ F \circ L_2 + L = F$ for some vectorial Boolean function F, where L_1 and L_2 are linear permutations and L is a linear function.

We implement a parallel version of an algorithm for testing EA equivalence in the programming language Rust. This allows us to compare the performance of implementations in C and Rust for similar problems and to conclude that our Rust implementation is comparable in efficiency while being significantly easier to write and maintain.

Using our implementation we calculate the self-equivalences for all known quadratic APN functions up to CCZ equivalence in dimensions 6, 8 and 10. We discover functions with trivial linear self-equivalence, but with nontrivial EL self-equivalences. Based on this we formulate a search procedure for obtaining new APN functions, which exploits extended linear self-equivalences in the same way that the search of Beierle et al. exploits linear self-equivalences.

From the initial test runs of our new algorithm we discover that the search allows us to start from a given APN function and find APN functions CCZ-inequivalent to it. More interestingly we observe that the search can even find non-quadratic APN functions.

Acknowledgements

First of all I would like to thank my supervisor Nikolay Kaleyski for his guidance and for all the time he set aside for me. I will remember all of our funny and educational zoom meetings.
I would also like to thank all my friends and family for their support.
Last but certainly not least I would like to thank my girlfriend for all her invaluable support, help and for being there for me. I would not have been able to do this without you.

Vegard Jensløkken
October 2023

Contents

Chapter 1

Introduction

Modern block ciphers rely on vectorial Boolean functions for their security against cryptanalytic attacks. APN functions offer optimal resistance against differential cryptanalysis, which is one of the most efficient methods known today. APN functions are in addition studied from a purely mathematical point of view, because they correspond to optimal objects in combinatorics, algebra and other disciplines. While they have been studied in depth since the early 90's, we still know little about APN functions. One reason for searching for new instances of APN functions is to get more examples and better understand their structure.

Obtaining new APN functions through exhaustive search requires computational resources far beyond what is achievable with current technology. This makes finding alternative methods of obtaining new functions necessary. The huge number of functions also makes it necessary to consider them up to a notion of equivalence. This is typically CCZ-equivalence, EA-equivalence, or linear equivalence. CCZ-equivalence preserves differential uniformity and is

the most general of these equivalence relations. EA-equivalence is more specialized and linear equivalence is the least general. In other words, if two functions are linear equivalent, they are also EA- and CCZ-equivalent, but not vice versa. In this thesis we are going to work with linear and EA equivalence. Two functions F and G are said to be EA equivalent if, $A_1 \circ F \circ A_2 + A = G$; where A_1 and A_2 are affine permutations, and A is an affine function. In the case where A_1 and A_2 are linear and A is 0, we say that F and G are linearly equivalent. In other words, we say that F and G are linearly equivalent of the equation, $L_1 \circ F \circ L_2 = G$, holds.

Any pair of linear permutations, (L_1, L_2), such that $L_1 \circ F \circ L_2 = F$ is called a linear self-equivalence of F. Self-equivalences can be defined in a similar way for EA equivalence, CCZ equivalence and so forth. Self-equivalences have previously been used to optimize the search for APN functions by drastically reducing the size of the search space. Beierle et al. used *linear* self-equivalence to obtain new APN functions [3][2]. Since EA equivalence is strictly more general than linear equivalence, a natural question to pose is whether a different set of functions can be obtained from *EA self-equivalences*. As part of the work conducted in this thesis, we show that there exist APN functions which only have a trivial linear self-equivalence, but have nontrivial EA self-equivalences.

1.1 The problem

Kaleyski proposed an algorithm for recovering EA equivalence between two functions F and G [13]. This was later implemented in the programming language, C, by Heggebakk, where she greatly improved the performance of the algorithm compared to the original Magma implementation [10]. C is known to be a highly efficient language, however, it is also known to have a steep learning curve. Recently Rust has been introduced as an alternative to C which promises to provide a more user friendly experience, while preserving the efficiency benefit of C [1]. For this reason we chose Rust as the programming

language for the work in this thesis. This allowed us to compare the performance of C and Rust for the particular application of implementing algorithms similar to the one presented in [10].

We use our Rust implementation to compute the self-equivalences of all known quadratic APN functions in dimension 6, 8 and 10 up to CCZ equivalence. To simplify the problem we choose to only consider the cases where A_1 and A_2, from the definition of EA equivalence, $A_1 \circ F \circ A_2 + A = G$, are linear. This gives us the equation $L_1 \circ F \circ L_2 + A = G$, and we say the triple (L_1, L_2, A) is an *extended linear equivalence*, or *EL equivalence*, between F and G. Similarly, any triple (L_1, L_2, A) where $L_1 \circ F \circ L_2 + A = F$ is called the *extended linear self-equivalence*, or simply *EL self-equivalence* of F.

It is known that the linear self equivalences (L_1, L_2) of any vectorial Boolean function form a group with the operation $(L_1, L_2) \circ (K_1, K_2) = (L_1 \circ K_1, K_2 \circ L_2)$. In this thesis we define an operation for composing EL self-equivalences as $(L_1, L_2, A) \circ (K_1, K_2, B) = (L_1 \circ K_1, K_2 \circ L_2, L_1 \circ B \circ L_2 + A)$. We show that the EL self-equivalences of any vectorial Boolean function form a group under this operation.

Based on the EL self-equivalences we propose a search procedure similar to that of Beierle et al., except that instead of linear self-equivalences it is based on EL self-equivalences. Unfortunately, we were not able to obtain any new APN functions, however we show that it is possible to go from quadratic APN functions to non-quadratic APN functions and to go between different CCZ classes. This shows that exploring this computational method further may be worthwhile and may yield new examples of APN functions that may not be obtainable by other searches.

To summarize, in this thesis we do the following:

1. Implement the algorithm from [13] in Rust to test linear and extended linear equivalence. In addition, we compare the performance of this Rust implementation with a previous implementation in C.
2. Define an operation under which the EL self-equivalences of any given function have a group structure.
3. Compute all the EL self-equivalences of all known quadratic APN functions up to CCZ equivalence in dimension 6, 8 and 10.
4. Formulate a search procedure for obtaining new APN functions based on EL self-equivalences.
5. Implement the search procedure and observe that it can (**a**) go from the self-equivalences of quadratic APN functions to non-quadratic APN functions, and (**b**) to go between different CCZ classes.

Chapter 2

Background

Let $\mathbb{F}_2$ denote the finite field consisting of the elements 0 and 1 and let $\mathbb{F}_2^n$ denote the vector space of dimension n. We denote the extension field of degree n over $\mathbb{F}_2$ by $\mathbb{F}_{2^n}$. The elements of the finite field $\mathbb{F}_{2^n}$ can naturally be identified with the elements of the vector space $\mathbb{F}_2^n$; we will therefore use them interchangeably throughout this thesis. We define a Boolean function as a mapping $f : \mathbb{F}_2^n \to \mathbb{F}_2$ that takes n binary inputs and outputs a single binary value. These functions are also known as $(n, 1)$-functions. Similarly (n, m)-functions are functions $\mathbb{F}_2^n \to \mathbb{F}_2^m$, that take n binary inputs and output m binary values. These functions are also known as *vectorial Boolean functions* and can be seen as being composed of m Boolean functions. In this thesis, however, we solely focus on the case where $n = m$, that is to say, for the remainder of this thesis we only consider (n, n)-functions.

Any (n, n)-function F, can be written as $F(x) = (f_1(x), f_2(x), ..., f_n(x))$ for $(n, 1)$-functions $f_1(x), f_2(x), ..., f_n(x)$. These functions are called the *coordinate functions* of F. The nonzero combinations of the coordinate functions F_b of F, where $b \in \mathbb{F}_2^n$ for $b \neq (0, 0, ..., 0)$ are called the *component functions* of F. More precisely we define F_b to be $F_b = \sum_{i=1}^{n} b_i f_i$, for $b = b_1, b_2, ..., b_n$. To give an example: Suppose we have the coordinate functions (f_1, f_2, f_3, f_4) of F. Based on

this we define a component function F_b of F and choose a $b = (1, 0, 0, 1)$. This yields the component function $F_b = f_1 + f_4$.

2.1 Representations

There are many ways of representing vectorial Boolean functions. In this section we will explain the two representations used throughout this thesis, truth table form and univariate representation.

2.1.1 Truth table form

Truth table form (sometimes referred to as a lookup table) is arguably the easiest when programming. A truth table is essentially just a list of all the values of a function. When using truth tables to represent vectorial Boolean functions, one typically uses the binary representation like in Table 2.1. In practice however, using binary strings like in Table 2.1 is rarely convenient when programming. When implementing vectorial Boolean functions in truth table form one typically uses integers to represent the binary strings like shown in Table 2.2. When using truth tables as input files for programs one typically has one row with an integer representing the dimension n followed by a line of 2^n, space separated, integers representing the output of the function, as shown in Figure 2.1.

x_1	x_2	x_3	x_4	$f_1(x_1, x_2, x_3, x_4)$	$f_2(x_1, x_2, x_3, x_4)$	$f_3(x_1, x_2, x_3, x_4)$
0	0	0	0	0	0	0
0	0	0	1	1	0	1
0	0	1	0	0	1	0
0	0	1	1	1	1	1
0	1	0	0	1	0	0
0	1	0	1	0	0	0
0	1	1	0	1	1	0
0	1	1	1	0	0	1
1	0	0	0	1	1	0
1	0	0	1	0	0	1
1	0	1	0	1	0	1
1	0	1	1	0	1	0
1	1	0	0	1	1	1
1	1	0	1	0	0	0
1	1	1	0	1	1	0
1	1	1	1	1	0	0

Table 2.1: Truth table representation of a $(4, 3)$-function.

(x_1, x_2, x_3, x_4)	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$F(x_1, x_2, x_3, x_4)$	0	5	2	7	4	0	6	1	6	1	5	2	7	0	6	4

Table 2.2: Integer truth table representation of Table 2.1.

```
4
0 4 1 3 4 9 2 14 2 1 0 0 15 1 2 11
```

Figure 2.1: Example of a truth table file.

2.1.2 Univariate representation

Another representation of an (n, n)-function, F, is the *univariate representation*. This is the polynomial:

$$F(x) = \sum_{i=0}^{2^n-1} c_i x^i,$$

for some coefficients $c_i \in \mathbb{F}_{2^n}$.
Many important functions, for instance APN-functions and differentially 4-uniform functions, have a very simple univariate representation.

2.2 Cryptography

Vectorial Boolean functions are widely used in symmetric ciphers and are often referred to as *substitution-boxes* or *S-boxes*. They are usually the only non-linear part of the algorithms and it is hence important to have a cryptographically strong function. In fact, the security of the cipher heavily depends on the properties of the function. For every possible cryptanalytic attack that exploits weaknesses in the S-boxes, we can define a property that measures how well the functions resist that attack. In this section we will list some of the most important cryptographic properties.

2.2.1 Algebraic degree

An important concept is the algebraic degree. This can be used to define functions, such as, affine and linear functions and is also used as a measure of how resistant a function is to so-called *higher-order differential attacks* [14].
The algebraic degree can be defined using the univariate representation,

$F(x) = \sum_{i=0}^{2^n-1} c_i x^i$. The largest binary weight of any exponent i in the univariate representation, with $c_i \neq 0$, is the algebraic degree or simply the *degree* of the function. To give an example: Suppose we have a $(4, 4)$-function $F = x^{13} + x^6 + x^5$. We can see that the exponents in binary are 1101, 0110 and 0101 respectively. Now it is easy to see that the binary weights of the exponents are 3, 2 and 2. Consequently, we see that F has an algebraic degree of 3.

Using algebraic degree we can define *affine* and *linear* functions. A function with degree of at most one is called *affine*. An affine (n, n)-function, A, has the property that $A(x) + A(y) + A(z) = A(x + y + z)$, for any $x, y, z \in \mathbb{F}_2^n$. A linear function, L, is simply an affine function satisfying $L(0) = 0$. Similarly to affine functions, linear functions satisfy the property that: $L(x) + L(y) = L(x + y)$ for any $x, y \in \mathbb{F}_2^n$. Any function with degree exactly 2 is *quadratic*.

2.2.2 Differential uniformity

Differential uniformity is used for measuring the resistance against *differential attacks* [4]. To define differential uniformity we first need to define the notion of *derivative* $D_a F$ of F in direction $a \in \mathbb{F}_{2^n}$:

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

Recall that in binary fields subtraction is the same as addition. Therefore, we can rewrite this as:

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

In addition, we need to define $\delta_F(a, b)$, for any $a, b \in \mathbb{F}_{2^n}$, which is the number of solutions $x \in \mathbb{F}_{2^n}$ to the equation $D_a F(x) = b$:

$$\delta_F(a,b) = \#\{x \in \mathbb{F}_{2^n} : D_aF(x) = b\}.$$

We can now define the *differential uniformity* of F, Δ_F, which is the largest value of $\delta_F(a,b)$:

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

2.2.3 APN functions

In order to prevent differential attack, Δ_F should be as low as possible. The lowest possible differential uniformity is 2. The functions that attain this optimal value are known as *almost perfect nonlinear functions*, usually referred to as *APN functions*.

2.2.4 Nonlinearity

Another important property measures a vectorial Boolean function's resistance to so called *linear attack* [16]. A linear attack occurs when an attacker tries to approximate an (n,n)-function by using a linear or an affine function. The nonlinearity $\mathcal{NL}(F)$ is defined as the minimum Hamming distance between any component function of F and any affine Boolean function. To define this formally we first need to introduce Hamming distance. This is the number of inputs where two functions disagree.

We can define the Hamming distance as:

$$d_H(F, G) = \#\{x : F(x) \neq G(x)\}.$$

The definition of nonlinearity is:

$$\mathcal{NL}(F) = min\{d_H(F_b, l) : b \in \mathbb{F}_2^n, b \neq (0, 0, ..., 0), l \in \mathcal{A}\},$$

where $\mathcal{A}$ is the set of all affine $(n, 1)$-functions.
To prevent linear attacks, one should choose functions with high nonlinearity.

2.2.5 AB functions

The nonlinearity of any (n, n)-function F satisfies

$$\mathcal{NL}(F) \leq 2^{n-1} - 2^{(n-1)/2}. \text{ [6]}$$

When a function satisfies this equation with equality we say that it is *almost bent*, abbreviated *AB*. These functions provide the best resistance to linear cryptanalysis, however only functions where the dimension n is odd can be AB. In addition, any AB function must be APN as well [8]. This means that any AB function provides the best resistance to both linear attacks and differential attacks. APN functions are not necessarily almost bent, but quadratic APN functions in odd dimensions are AB [7].

2.3 Equivalence relations

When n increases, the number of (n, n)-functions, and in particular APN functions, increase exponentially. Thus, it is necessary to reduce the number of functions to consider. We do this by only considering functions up to certain relations of *equivalence*, that preserve APN-ness. There are different equivalence relations and we will cover some of the most useful ones in this section.

2.3.1 CCZ-equivalence

The most general known equivalence relation that preserve APN-ness is *Carlet-Charpin-Zinoviev-equivalence* or *CCZ-equivalence* for short [7]. CCZ-equivalence preserves both differential uniformity and nonlinearity. This makes the relation very useful when classifying functions. The CCZ-equivalence between two functions F and G is defined by taking the graph of the functions.

We define the graph of a function F, Γ_F, as

$$\Gamma_F = \{(x, F(x)) : x \in \mathbb{F}_2^n\}.$$

If there exists an affine permutation A of $\mathbb{F}_2^n \times \mathbb{F}_2^n$ that maps Γ_F to Γ_G, then F and G are CCZ-equivalent.

2.3.2 EA-equivalence

Suppose we have two functions, F and G. They are said to be EA equivalent if there exist two affine permutations A_1 and A_2, and an affine function A such that:

$$A_1 \circ F \circ A_2 + A = G.$$

EA-equivalence is a special case of CCZ-equivalence, although In the case of monomials and quadratic APN functions two functions are CCZ-equivalent if and only if they are EA-equivalent [17].

2.3.3 Affine and linear equivalence

The functions, F and G, are *affine equivalent* if we have two affine permutations, A_1 and A_2, such that:

$$A_1 \circ F \circ A_2 = G.$$

This is one of the most specialized equivalence relations that preservers APN-ness. We can see that it is a special case of EA-equivalence where A from the equation $A_1 \circ F \circ A_2 + A = G$, is zero.

In the special case where A_1 and A_2 are linear, we say that F and G are *linearly equivalent*. This is defined as:

$$L_1 \circ F \circ L_2 = G.$$

2.3.4 Self-equivalence

For any of the above relations, an equivalence between a function and itself is called a self-equivalence. For example, a *linear self-equivalence* of F is any pair of linear permutations, (L_1, L_2), such that $L_1 \circ F \circ L_2 = F$.

2.4 Known methods for obtaining new APN functions

Searching for new APN functions is known to be hard due to the huge number of functions to consider. As n grows, the number of functions grows super exponentially. In practice, it is infeasible to perform an exhaustive search over all of these functions. Another approach for finding new functions is needed.

With the help of some optimizations it is possible to search for functions using truth tables but this only works up to dimension $n = 5$. Brinkmann and Leander classified all functions using this approach [5]. This means that we only consider searches of dimension $n > 5$.

2.4.1 Polynomial search

One way of reducing the search space is by only considering functions with a short univariate representation. First, one would go through all monomials, that is, all possible functions of the form $F = x^A$ for all possible A. One then proceed to go through all binomials of the form $F = x_1^A + c_1 x_2^B$, for all possible A, c_1, B, such that F is APN. Of course this is only possible for relatively short polynomials.

2.4.2 Matrix approach

Yuyin et al. introduced a new matrix construction they refer to as *quadratic APN matrix*, *QAM* [18]. With this new approach, they managed to discover 2,252 new quadratic APN functions up to CCZ equivalence in dimension $n = 8$ and 471 new functions in dimension $n = 7$.

2.4.3 Self-equivalence search

Beierle et al. illustrated that it is possible to use linear self-equivalences to obtain new functions in dimension 6, 7 and 8 [3].

First we take a fixed linear self-equivalence (L_1, L_2). We search for all functions F such that $L_1 \circ F \circ L_2 = F$. We proceed to guess one value of F, say $F(1) = 3$. From here we can obtain $F(L_2(1)) = L_1^{-1}(F(1))$. Then, we apply L_2 again to obtain $F(L_2(L_2(1))) = L_1^{-1}(L_1^{-1}(F(1)))$. We continue by applying L_2 on the left hand side of the equation and L_1^{-1} on the right hand side, until L_2 loops around and we get the original first guess. This means that a single guess can derive multiple values of F.

With their search, Beierle et al. were able to reduce the search space. Later, Beierle and Leander applied this search to higher dimensions and obtained $12,956$ new quadratic APN. Of these functions, $12,921$ was in dimension $n = 8$, and 35 new APN functions in dimension $n = 9$. Two of the functions in dimension $n = 9$ were permutations [2].

As shown in Chapter 3, there exist APN functions having only trivial linear self-equivalence, but nontrivial EL self-equivalence. These functions can not be found by Beierle's search, but may potentially be discovered by the search that we propose in Section 3.5. This search is similar to that of Beierle et al., except that we use *extended linear self-equivalences*.

2.5 Algorithm for recovering EA equivalence

Recall that there are an exponentially increasing amount of functions to consider when the dimension n increases. As a consequence, we need to use equivalence relations to classify them. When searching for new functions any new function needs to be tested for equivalence against the known ones. This, however, is not always easy in practice. Since CCZ is the most general equivalence relation, a function is only considered to be new if it is not CCZ equivalent to any of the known ones. Due to the fact that the majority of the known APN functions up to CCZ equivalence are quadratic, we only have to test for EA equivalence. This is because, like stated earlier, quadratic APN functions are CCZ equivalent if and only if they are EA equivalent.

Kaleyski has proposed an algorithm for recovering EA-equivalence [13]. This algorithm uses invariants to check whether two functions, F and G, are EA-equivalent. The algorithm reconstructs the affine permutations A_1 and A_2, from the definition of EA equivalence, in two steps. Without loss of generality one can assume that A_1 is linear, i.e. L_1. For quadratic functions one can also assume that A_2 is linear. We rewrite the equivalence relation as $L_1 \circ F \circ L_2 + A = G$.

In our case this algorithm is used for quadratic APN functions, so we will only consider this case. To obtain L_1 we calculate the multiset, $\mathcal{M}_F$, containing the elements $\mathcal{M}_F = \{F(x_1) + F(x_2) + F(x_3) + F(x_1 + x_2 + x_3) : x_1, x_2, x_3 \in \mathbb{F}_{2^n}\}$. With this multiset we consider the multiplicities of each element.

We calculate the multiset $\mathcal{M}_F$ for F and $\mathcal{M}_G$ for G. We know that L_1 must map any element with a given multiplicity in $\mathcal{M}_F$, to some element with the same multiplicity in M_G. There are very few linear permutations that satisfy this condition and this greatly reduces the search space.

After we guess L_1 we can obtain L_2: By applying L_1^{-1} to A and G we get $F \circ L_2 = G' + A'$. This can also be formulated as $F \circ L_2 = G''$.

Kaleyski denote by $\mathcal{T}_k(0)$ the set of all k-tuples of elements from $\mathbb{F}_2^n$ that add up to zero. This can be expressed as: $\mathcal{T}_k(0) = \{(x_1, x_2, ..., x_k) \in (\mathbb{F}_2^n)^k \mid \sum_{i=1}^{k} x_i = 0\}$. Using this definition, Kaleyski goes on to show that if there is an element in $\mathbb{F}_2^n$ that is part of a k-tuple that sums up to some element t under G'', then the image under L_2 has to be part of a k-tuple whose sum under F is t. This can be expressed mathematically as $L_2(x) \in \cap_{t \in \sum_k^G(0,x)}(\cup O_k^F(0,t)$, where $O_k^F(0,t) = \{(x_1, x_2, ..., x_k) \in \mathcal{T}_k(0) \mid F(x_1)+F(x_2)+...+F(x_k) = t\}$. He further explains that it is sufficient to use $k = 3$ to restrict the number of L_2 to consider. Ultimately the search space for L_1 and L_2 is reduced to such an extent that it is feasible to go through all possible choices and check whether one of them is an equivalence. Lastly we obtain A' by solving the equation $A' = F \circ L_2 + G'$ and verify that A' is indeed affine. For an exact description of the algorithm, we refer the reader to [13].

Chapter 3

Results

Beierle et al. show how linear self-equivalence can be used to obtain new APN functions. This raises the question: can we obtain other functions by using a more general equivalence relation? We decide to investigate EA equivalence, however, in order to simplify the study, we restrict to the special case where A_1 and A_2 from the equation $A_1 \circ F \circ A_2 + A = G$ are linear. We call this extended linear equivalence, and write it as $L_1 \circ F \circ L_2 + A = G$. This is in fact the most practically relevant case because, for quadratic APN functions, we can assume that the permutations A_1 and A_2 are linear without loss of generality.

We implement Kaleyski's algorithm in Rust and compute all the EL self-equivalences for every known quadratic APN function up to CCZ equivalence. We observe that some functions only have trivial linear self-equivalences, but have nontrivial extended linear self-equivalences. This motivates us to define a new search similar to that of Beierle et al., except that instead of using linear self-equivalences, it uses extended linear self-equivalences. Our first step is to prove that the EL self-equivalences form a group. We do this in the following section.

3.1 Extended linear equivalence group

We introduce the following operation on the set of EL self-equivalences:

Definition 1 Let $\mathcal{SE}_{EL}(F)$ denote the set of all EL self-equivalences of $F \in \mathbb{F}_{2^n}$ and let $(L_1, L_2, a), (K_1, K_2, b) \in \mathcal{SE}_{EL}(F)$.
We define the operation $\circ_{EL}$ as:

$$(L_1, L_2, a) \circ_{EL} (K_1, K_2, b) = (L_1 \circ K_1, K_2 \circ L_2, L_1 \circ b \circ L_2 + a).$$

From here on we will write $\circ$ instead of $\circ_{EL}$, and it should be clear from the context which operation the symbol refers to.

Recall that a group, is a set, G, with a binary operation $*$ that satisfies the following axioms [15]:

1. It is $associative$; for any $A, B, C \in G$
   $(A * B) * C = A * (B * C)$.
2. It has an $identity\ element\ I$ such that for any $A \in G$
   $A * I = I * A = A$.
3. Every element has an $inverse$; for all $A \in G$ there exists $A^{-1} \in G$ such that
   $A * A^{-1} = A^{-1} * A = I$.

We now show that for any vectorial Boolean function $F \in \mathbb{F}_{2^n}$, the set $\mathcal{SE}_{EL}(F)$ together with the operation from Definition 1, form a group. We verify that each axiom from the definition of a group holds.

1. First we prove that the group is associative:
   Let $A = (L_1, L_2, a), B = (K_1, K_2, b), C = (H_1, H_2, c)$, then
   $A \circ B = (L_1 \circ K_1, K_2 \circ L_2, L_1 \circ b \circ L_2 + a)$ and
   $B \circ C = (K_1 \circ H_1, H_2 \circ K_2, K_1 \circ c \circ K_2 + b)$.

   We can compute $A \circ (B \circ C)$ like this:
   $A \circ (B \circ C) = (L_1 \circ K_1 \circ H_1, H_2 \circ K_2 \circ L_2, L_1 \circ K_1 \circ c \circ K_2 \circ L_2 + L_1 \circ b \circ L_2 + a)$.

   Then we compute $(A \circ B) \circ C$ as such:
   $(A \circ B) \circ C = (L_1 \circ K_1 \circ H_1, H_2 \circ K_2 \circ L_2, L_1 \circ K_1 \circ c \circ K_2 \circ L_2 + L_1 \circ b \circ L_2 + a)$.

   Since $A \circ (B \circ C) = (A \circ B) \circ C$ we know that the group is associative.

2. We proceed to prove that the group has an identity element:
   Let $I = (id, id, 0)$, where id is the identity mapping. Let $A = (L_1, L_2, a)$, then:
   $A \circ I = (L_1 \circ id, id \circ L_2, L_1 \circ 0 \circ L_2 + a) = (L_1, L_2, a)$
   $= A = I \circ A = (id \circ L_1, L_2 \circ id, id \circ a \circ id + 0) = (L_1, L_2, a) = A$.

   Hence $I = (id, id, 0)$ is the identity element of the group.

3. Lastly we verify that every element has an inverse:
   Let $A = (L_1, L_2, a)$ and $A^{-1} = (L_1^{-1}, L_2^{-1}, L_1^{-1} \circ a \circ L_2^{-1})$. Let id be the identity mapping on $\mathbb{F}_2^n$. Now:
   $A \circ A^{-1} = (L_1 \circ L_1^{-1}, L_2^{-1} \circ L_2, L_1 \circ L_1^{-1} \circ a \circ L_2^{-1} \circ L_2 + a) = (id, id, 0) = I$
   and
   $A^{-1} \circ A = (L_1^{-1} \circ L_1, L_2 \circ L_2^{-1}, L_1^{-1} \circ a \circ L_2^{-1} + L_1^{-1} \circ a \circ L_2^{-1}) = (id, id, 0) = I$.

   We have proved that the group has an inverse.

Using the group axioms we have now proved that any vectorial Boolean function F, the set $\mathcal{SE}_{EL}(F)$, forms a group.

3.2 Rust

The preferred language in programmatic implementation of finite field arithmetic is typically very high-level, ergonomic alternatives with integrated utilities for mathematical calculations. Frequently used are Magma, Python, and Sagemath. While these languages are well-suited to create a 'proof of concept' their computational speed is known to be quite slow. This becomes especially apparent when processing large quantities of data. When trying to speed up algorithms written in these languages, C and C++ are often used.

For our implementation we wanted to utilize a high-level, easy to learn, modern programming language. We wanted a high performing language that can handle large computational data. Rust was a natural choice and to the best of our knowledge there are not many examples of Rust being used in this field. It is therefore interesting to see how it performs in comparison to C.

Languages like Java and Python use a process called garbage collector. A garbage collector is a process that periodically frees up unused memory. This reduces the cognitive load for the programmer, but it also slows down performance. There is no garbage collector in Rust, making it faster and more memory efficient compared to languages like Java and Python [1]. Languages like C do not have a garbage collector either, however one must explicitly free memory, making it easy to make mistakes. Rust implements a borrow system, where memory is dropped as soon as it goes out of scope. This ensures that memory is efficiently freed while the programmer does not have to explicitly handle it. Being a modern language, Rust has data structures and a modern compiler with descriptive error messages and suggestions for when one inevitably makes mistakes. Thus Rust combines a high efficiency comparable to that of C and C++, with a relative ease of use comparable to modern programming languages like Java and Python.

3.3 Implementation of Kaleyski's algorithm

Like described in Section 2.5, Kaleyski proposed an algorithm for computing EA-equivalences. We have implemented this algorithm in Rust and mad it available at [11]. This also allowed us to use Rust's facilities for easy parallel implementation.

3.4 Benchmarking Rust

Like previously mentioned, the C programming language is frequently used to increase performance. Heggebakk showed in her thesis that her C-implementation of Kaleyski's algorithm is up to 300 times faster than the one in Magma [10]. Unlike our implementation, Heggebakk's version did not run in parallel. To compare the two implementations we changed our implementation so that it too runs sequentially. The comparison was done in the same way and on the same server as Heggebakk. For each of the functions to be tested, we created 10 different equivalent functions using randomly generated L_1, L_2 and A. The test was then done on both of the implementations in order to get an average running time.

Table 3.1 is structured in the same way as in [10]. The first column is the dimension n and the second column is the id in the same way as in [9]. The next two columns are the running times of Heggebakk's C implementation and our Rust implementation, respectively, in seconds. As we can see from the table, the running time of our implementation is faster in 21 out of the 27 tested functions. For function 5.1 we observe that the running timer is 17 times faster than the C equivalent and we can calculate that the median speedup of these funcitons is 79%.

		Heggebakk's C implementation	This Rust implementation
n	name	time	time
8	1.1	2.2522	0.7156
	1.2	2.1718	0.6773
	1.3	26.5036	23.2005
	1.4	28.4011	19.6781
	1.5	73.7415	46.6857
	1.6	77.6487	57.3253
	1.7	9.9878	9.7976
	1.8	20.2737	13.3931
	1.9	34.5491	19.2599
	1.10	6.1019	4.4727
	1.11	24.4097	18.4355
	1.12	11.8153	6.5502
	1.13	16.1372	17.0938
	1.14	2.4758	0.6650
	1.15	89.5798	110.0045
	1.16	8.3757	9.4433
	1.17	7.3813	8.1360
	2.1	12.4223	8.5608
	3.1	13.3039	17.0895
	4.1	5.5313	1.1602
	5.1	11.6943	0.6815
	6.1	2.6697	0.6704
	7.1	2.1930	0.6133
10	1.1	194.7185	34.0678
	1.2	189.1036	37.8949
	1.5	1098.5705	1013.1090
	1.6	1225.41360	2335.3506

Table 3.1: Comparison of running times between a C implementation and our Rust implementation.

3.5 Extended linear self-equivalence groups

We have run our implementation of Kaleyski's algorithm for all known APN functions up to CCZ equivalence in dimensions $6, 8$ and 10 to compute all the self-equivalence groups and made it available at [12]. We observe that 3 out of

14 functions in dimension 6 have nontrivial EL self-equivalence and the same is true for 36 out of 26 514 functions in dimension 8. Most interestingly we observe that all three functions with nontrivial EL self-equivalence in dimension 6, and 9 of the functions in dimension 8 have a trivial linear self-equivalence group. Out of 26 514 functions in dimension 8, 13 758 had a trivial EL self-equivalence group. In dimension 6, 7 out of 14 had a trivial EL self-equivalence group.

3.6 Generating APN functions from self-equivalences

Based on the previous section, a natural question is whether a search similar to that of Beierle et al., but using extended linear self-equivalences instead of linear self-equivalences; this may produce functions that would not be obtainable from Beierle's search.

We propose the following search: in our algorithm we assume that $F(0) = 0$. To begin, we fill an empty truth table with 2^n zeros. We then guess the value of the first element and generate multiple elements for free. Step-by-step the search is as follows:

1. We start with the equation: $L_1 \circ F \circ L_2 + A = F$.
2. We can rewrite the equation as: $F \circ L_2 = L_1^{-1} \circ F + L_1^{-1} \circ A$.
3. Due to L_1^{-1} being linear we can rewrite it again as: $F \circ L_2 = L_1^{-1}(F + A)$.
4. Using the equation above we can start by assigning the first element of the first cycle. $F(x) = y$, for some $x, y \in \mathbb{F}_{2^n}$.
5. We apply L_2 to both sides:
   $F(L_2(x)) = L_1^{-1}(F(x) + A(x)) = F(L_2(x)) = L_1^{-1}(y + A(x))$.
   We have now derived the value of $F(L_2(x))$, due to the fact that L_1^{-1}, L_2 and A are all known.

6. Finally we can apply L_2 again to get:
   $F(L_2(L_2(x))) = L_1^{-1}(F(L2(x)) + A(L_2(x))$.
   We continue this until we loop, in other words, until $L_2(L_2(L_2(...(L_2(x))))) = x$, being $F(x) = y$.

We precompute what we call cycles from L_2. A *cycle* under L_2 is all the elements that can be obtained from some x by successive application of L_2.This means that we start by letting the first element of the first cycle to be 1. Then we apply L_2 to 1 to obtain, say 50. We proceed to apply L_2 to 50 to obtain some new number. We continue doing this until we reach the start, i.e 1. If there are elements from $\mathbb{F}_{2^n}$ that do not belong in the cycle, then we start computing the next cycle. We do this in the same way, except we start by assigning the first element of this cycle as the smallest element that is not in the previous cycle.

The benefit of precomputing the cycles for each element of the self-equivalence group of all functions with nontrivial EL self-equivalence is that we are able to choose the best possible element for the search. If we know that for some function F in dimension $n = 8$ we have one L_2 that induces 15 cycles, and another that induces 20 cycles, then we have $(2^8)^5$ less iterations if we chose the L_2 with 15 cycles. Based on this, we compute all these cycles for all APN functions with nontrivial EL self-equivalence in dimension $6, 8$ and 10 up to CCZ equivalence.

3.7 Backwards search

In addition to the search described in the previous section, we can also try to search for new functions by successively replacing the values of cycles. Say the function has, for instance, 15 cycles. We first try to replace the value of the last cycle, and then the last two cycles until we reach the first cycle. Consequently, we might obtain a new function that shares the two first cycles with the original function, resulting in us obtaining the functions sooner than if we were to guess all the cycles. We call this a *backwards search*.

3.8 Computational results

To gauge the possibility of using these searches to obtain new functions, we ran some preliminary experimental computations in dimension 6 and 8. We used both the backwards search and the regular search. We observed that for the functions "$Beierle\_14$" and "$Beierle\_15$" from [2] in dimension 8, we were able to go from one equivalence class to another. In dimension 6, we ran the search for the functions "1.3" and "1.7" from [9] and we observed that we can go from quadratic functions to non-quadratic functions. Unfortunately, none of these functions proved to be new, regardless we have clearly demonstrated that the search can produce inequivalent functions and indeed functions of a different algebraic degree. In light of this, this is a computational approach that deserves further investigation.

Chapter 4

Conclusion and future work

4.1 Conclusion

We studied extended linear self-equivalences and defined an operation under which EL self-equivalences form a group. We implemented the EL equivalence algorithm of Kaleyski in Rust. We benchmarked our implementation and were able to show that it is comparable to a previous implementation written in C. We extended our implementation to run in parallel.

We calculated all the self-equivalences for all known CCZ-inequivalent quadratic APN functions in dimensions 6, 8, and 10 using our parallel implementation. Based on these computational results we observe that there exist functions with non-trivial EL self-equivalence that only have trivial linear self-equivalences.

Based on the results from our computation we formulated an example of a search similar to that of Beierle et al. This search can find functions having a nontrivial EL self-equivalence. Using this new search, we ran some preliminary experimental searches that showed that searches of this type can transition between CCZ classes, and can even find functions of a different algebraic degree.

4.2 Future work

In our work we were only able to compute self-equivalences for quadratic APN functions in even dimensions, because Kaleyski's algorithm does not work for quadratic APN functions in odd dimensions. One potential direction for future work would be to use a different algorithm to calculate EL self-equivalences for quadratic APN functions in odd dimensions and to continue the searches there.

A natural next step would be to take the search and use it to its full potential. By doing a dedicated computational work, one can try to classify all EL self-equivalences and run the search in parallel for an extended period of time to see how many new functions can be obtained.

Bibliography

[1] The Rust Programming Language - The Rust Programming Language.
URL: https://doc.rust-lang.org/book/.

[2] Christof Beierle and Gregor Leander. New Instances of Quadratic APN Functions. *IEEE Transactions on Information Theory*, 68(1):670–678, January 2022. ISSN 1557-9654. doi: 10.1109/TIT.2021.3120698. Conference Name: IEEE Transactions on Information Theory.

[3] Christof Beierle, Marcus Brinkmann, and Gregor Leander. Linearly Self-Equivalent APN Permutations in Small Dimension. *IEEE Transactions on Information Theory*, 67(7):4863–4875, July 2021. ISSN 1557-9654. doi: 10.1109/TIT.2021.3071533. Conference Name: IEEE Transactions on Information Theory.

[4] Eli Biham and Adi Shamir. Differential cryptanalysis of DES-like cryptosystems. *Journal of Cryptology*, 4(1):3–72, January 1991. ISSN 0933-2790, 1432-1378. doi: 10.1007/BF00630563.
URL: http://link.springer.com/10.1007/BF00630563.

[5] Marcus Brinkmann and Gregor Leander. On the classification of APN functions up to dimension five. *Designs, Codes and Cryptography*, 49(1): 273–288, December 2008. ISSN 1573-7586. doi: 10.1007/s10623-008-9194-6.
URL: https://doi.org/10.1007/s10623-008-9194-6.

[6] Claude Carlet. *Boolean Functions for Cryptography and Coding Theory*. Cambridge University Press, 1 edition, November 2020. ISBN 978-1-108-60680-6 978-1-108-47380-4. doi: 10.1017/9781108606806.
URL: https://www.cambridge.org/core/product/identifier/9781108606806/type/book.

[7] Claude Carlet, Pascale Charpin, and Victor Zinoviev. Codes, Bent Functions and Permutations Suitable For DES-like Cryptosystems. *Des. Codes Cryptography*, 15:125–156, November 1998. doi: 10.1023/A:1008344232130.

[8] Florent Chabaud and Serge Vaudenay. Links between differential and linear cryptanalysis. In Alfredo De Santis, editor, *Advances in Cryptology — EUROCRYPT'94*, Lecture Notes in Computer Science, pages 356–365, Berlin, Heidelberg, 1995. Springer. ISBN 978-3-540-44717-7. doi: 10.1007/BFb0053450.

[9] Yves Edel and Alexander Pott. A new almost perfect nonlinear function which is not quadratic. *Advances in Mathematics of Communications*, 3(1): 59–81, December 2008. ISSN 1930-5346. doi: 10.3934/amc.2009.3.59.
URL: https://www.aimsciences.org/en/article/doi/10.3934/amc.2009.3.59. Publisher: Advances in Mathematics of Communications.

[10] Marie Heggebakk. An efficient implementation of a test for EA-equivalence. Master's thesis, The University of Bergen, June 2022.
URL: https://bora.uib.no/bora-xmlui/handle/11250/3003709. Accepted: 2022-07-07T23:40:57Z.

[11] Vegard Jensløkken. jenslokken/EA-equivalence-Rust, October 2023.
URL: https://github.com/jenslokken/EA-equivalence-Rust.

[12] Vegard Jensløkken. jenslokken/Extended-linear-APN-search, October 2023.
URL: https://github.com/jenslokken/Extended-linear-APN-search.

[13] Nikolay Kaleyski. Deciding EA-equivalence via invariants. *Cryptography and Communications*, 14(2):271–290, March 2022. ISSN 1936-2455. doi: 10.1007/s12095-021-00513-y.
URL: https://doi.org/10.1007/s12095-021-00513-y.

[14] Lars R. Knudsen. Truncated and higher order differentials. In Bart Preneel, editor, *Fast Software Encryption*, Lecture Notes in Computer Science, pages 196–211, Berlin, Heidelberg, 1995. Springer. ISBN 978-3-540-47809-6. doi: 10.1007/3-540-60590-8_16.

[15] Rudolf Lidl and Harald Niederreiter. Algebraic Foundations. In *Finite Fields*, Encyclopedia of Mathematics and its Applications, pages 1–46. Cambridge University Press, Cambridge, 2 edition, 1996. ISBN 9780521392310. doi: 10.1017/CBO9780511525926.003.
URL: https://www.cambridge.org/core/books/finite-fields/algebraic-foundations/97B0BD50368571B6BA2EBB0AED8DC6A1.

[16] Mitsuru Matsui. Linear Cryptanalysis Method for DES Cipher. In Tor Helleseth, editor, *Advances in Cryptology — EUROCRYPT '93*, Lecture Notes in Computer Science, pages 386–397, Berlin, Heidelberg, 1994. Springer. ISBN 978-3-540-48285-7. doi: 10.1007/3-540-48285-7_33.

[17] 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.
URL: http://link.springer.com/10.1007/s10801-011-0309-1.

[18] Yuyin Yu, Mingsheng Wang, and Yongqiang Li. A Matrix Approach for Constructing Quadratic APN Functions, 2013.
URL: https://eprint.iacr.org/2013/007. Report Number: 007.