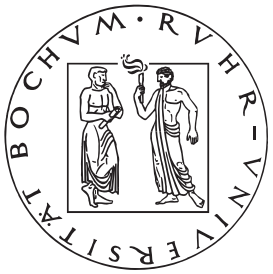
Classification of Almost Perfect Nonlinear Functions up to Dimension Five

Marcus Brinkmann

Diplomarbeit

Fakultät für Mathematik
Ruhr-Universiät Bochum



März 2007

Hiermit erkläre ich, dass ich die vorliegende Diplomarbeit selbständig verfasst und dabei keine anderen als die hier angegebenen Hilfsmittel benutzt habe.

Marcus Brinkmann
Bochum, im März 2007

Contents

1 Introduction **1**
- 1.1 State of the Art . . . 1
- 1.2 Overview . . . 2
  - 1.2.1 Main contributions . . . 3
  - 1.2.2 Notation . . . 4
- 1.3 Future Work . . . 4

2 Backtrack Algorithms **5**
- 2.1 Partial functions . . . 5
  - 2.1.1 Function templates . . . 6
  - 2.1.2 Compatible functions . . . 7
  - 2.1.3 Examples . . . 8
- 2.2 Index function . . . 8
- 2.3 Backtrack . . . 9
  - 2.3.1 Basic backtrack algorithm . . . 9
  - 2.3.2 Problem representation and representation problems . . . 12
  - 2.3.3 Backtrack with stateful filters . . . 13
- 2.4 Isomorph rejection in backtrack algorithms . . . 15
  - 2.4.1 Canonicity filter . . . 15
  - 2.4.2 Weak canonicity filter . . . 16
- 2.5 Estimating the efficiency of backtrack algorithms . . . 18

3 Affine subspaces in $\mathbb{F}_2^n$ of dimension 2 **19**
- 3.1 The case $\mathcal{A}(\mathbb{F}_2^n)$ . . . 19
- 3.2 Decomposition Lemma . . . 20
- 3.3 The case $\mathcal{A}([0;k-1])$ . . . 21
- 3.4 The case $\mathcal{A}(M)$, M arbitrary . . . 22
  - 3.4.1 The case $\mathcal{A}(M)$, M saturated . . . 23
  - 3.4.2 Saturation of arbitrary sets . . . 24
  - 3.4.3 Upper bound on $\mathcal{A}(M)$, M arbitrary . . . 26

4 Classification of APN functions **27**
- 4.1 APN functions . . . 27
  - 4.1.1 APN functions and affine subspaces . . . 27
  - 4.1.2 APN filter predicate . . . 29

4.1.3 Evaluation of ϕ_{APN} . . . 32
4.2 Affine equivalence . . . 35
4.2.1 Affine functions and refinements . . . 35
4.2.2 Affine canonicity filter . . . 37
4.2.3 Evaluation of the affine canonicity filter . . . 37
4.2.4 EA canonicity filter . . . 39
4.2.5 Evaluation of the EA canonicity filter . . . 40
4.3 CCZ equivalence . . . 41
4.3.1 CCZ equivalence test . . . 42
4.3.2 Weak CCZ canonicity filter . . . 44

A Implementation notes **47**
A.1 Efficient data structures . . . 48
A.2 Global state and rollback functions . . . 48
A.3 Static compilation for a fixed problem size . . . 49
A.4 Reordered execution flow . . . 49
A.5 Exploitation of machine characteristics . . . 49
A.6 Engineering and mathematics . . . 51

Chapter 1

Introduction

In this thesis we classify all almost perfect nonlinear (APN) vectorial boolean functions in dimension 4 and 5 up to affine and CCZ equivalence using backtrack programming and give a partial model for the complexity of such a search. In particular, we demonstrate that up to dimension 5 any APN function is CCZ equivalent to a power function, while in dimension 4 and 5 there exist APN functions which are not extended affine equivalent to any power function.

A function $s : \mathbb{F}_2^n \to \mathbb{F}_2^n$ is APN if for every non-zero $c \in \mathbb{F}_2^n$ and every $a \in \mathbb{F}_2^n$ the equation $s(x)+s(x+c) = a$ has at most two solutions (see Definition 4.1 for more details). APN functions were introduced by Nyberg [Nyb94] to disable differential attacks on block ciphers [BS91]. Since then they have been studied extensively, see [Car06] for a comprehensive bibliography.

The APN property as well as other cryptographically interesting properties are invariant under affine transformations [Nyb94]. Let $\alpha, \beta : \mathbb{F}_2^n \to \mathbb{F}_2^n$ be affine bijections and $\gamma : \mathbb{F}_2^n \to \mathbb{F}_2^n$ be any affine function, then with

$$t = \alpha s \beta + \gamma \tag{1.1}$$

the function t is APN if and only if s is APN, and t is said to be *extended affine (EA) equivalent* to s. If $\gamma \equiv 0$ and s is a bijection, t is a bijection and we say t is *affine equivalent* to s. Also, if s is an APN bijection the inverse s^{-1} is APN.

In [CCZ98], Carlet, Charpin and Zinoviev introduced a general affine equivalence relation that includes EA and inverse equivalence as special cases. Let $\mathcal{G}(s) := \{(x, s(x)) \mid x \in \mathbb{F}_2^n\} \subseteq \mathbb{F}_2^n \times \mathbb{F}_2^n$ be the graph of the function s. Then a function t is *CCZ equivalent* to s if $\mathcal{G}(t)$ is affine equivalent to $\mathcal{G}(s)$ in $\mathbb{F}_2^n \times \mathbb{F}_2^n$, that means if there exists an affine function $\lambda \in \mathbb{F}_2^{2n}$ such that:

$$\mathcal{G}(t) = \lambda(\mathcal{G}(s)) \tag{1.2}$$

It was proven in [CCZ98] that this equivalence relation stabilises the APN property.

1.1 State of the Art

Until recently, all known constructions of APN functions happened to be equivalent to power functions on $\mathbb{F}_{2^n}$, see Table 1.1, and it was an open question if this is true for all

Table 1.1: Known APN power power functions x^d on $\mathbb{F}_{2^n}$

Functions	Exponents d	Conditions	Proven in
Gold	2^i+1	$\gcd(i,n)=1$	[Gol68, Nyb94]
Kasami	$2^{2i}-2^i+1$	$\gcd(i,n)=1$	[JW93, Kas71]
Welch	2^t+3	$n=2t+1$	[Dob99b]
Niho	$2^t+2^{\frac{t}{2}}-1, t$ even $2^t+2^{\frac{3t+1}{2}}-1, t$ odd	$n=2t+1$	[Dob99a]
Inverse	$2^{2t}-1$	$n=2t+1$	[BD94, Nyb94]
Dobbertin	$2^{4i}+2^{3i}+2^{2i}+2^i-1$	$n=5i$	[Dob00]

APN functions. Budaghyan, Carlet and Pott [BCP06] constructed APN functions EA inequivalent (but CCZ equivalent) to any power function. Then Edel, Kyureghyan and Pott [EKP06] constructed a quadratic function from $\mathbb{F}_2^{10}$ to itself that they showed to be CCZ inequivalent to any power function, and shortly afterwards an infinite class of functions with this property was found [BCFL06]. Since then even more APN functions that are CCZ inequivalent to any power function have been found in dimensions as low as 6 [Dil06].

Despite this recent progress, many elementary questions about APN functions remain unanswered. For example, it is unknown if there exist bijective APN functions in even dimensions. A partial negative answer given in [dH03] is not conclusive, as it is not very restrictive compared to the size of the concerned function spaces.

APN functions have been found to be quite elusive in other ways as well. For example, APN functions are difficult to construct from smaller APN functions [dH03] or from functions satisfying weaker constraints [Mil98].

1.2 Overview

In this thesis, we do not investigate families of APN functions which vary over the dimension, but exhaustively search for and classify all APN functions in specific dimensions, thus adding to a solid foundation of facts for future APN related research. It is clear that this can only succeed for very small dimensions n, as the search space grows super-exponentially. A complete classification of dimension up to 3 is easy to do with just pen and paper. We add the classification of dimension 4 and 5. Although the techniques presented here apply to any dimension, finding complete solutions for higher dimensions will require the development of additional or different techniques.

We solve the classification problem using backtrack programming with isomorph rejection. In backtrack programming the search space is divided into increasingly finer subsets which contain all functions agreeing on increasingly larger subsets of their domain, and we reject subsets that do not contain a canonical representative of the desired equivalence class. This is possible because the APN and canonicity properties can be tested efficiently even on functions that are only locally determinate.

We demonstrate that there are two EA equivalence classes for $n = 4$, one of which is not EA equivalent to any power function. For $n = 5$, there are seven EA equivalence classes, two of which are not EA equivalent to any power function. The functions not EA equivalent to any power function are members of the infinite families of APN functions presented in Theorem 1 and 3 of [BCP06]. We show that no other classes of APN functions exist in dimension 4 and 5.

Furthermore, we demonstrate that all APN functions for $n = 4$ are in the same CCZ equivalent class. For $n = 5$, all APN functions are equivalent to one of three CCZ equivalence classes, each containing power functions. There exist APN functions not CCZ equivalent to any power function for dimensions as low as 6, see [Dil06]. We show that this is an exact lower bound.

All computations were performed on a Pentium 4 processor with 2.8 GHz. The results for $n \leq 4$ are immediate, and the case $n = 5$ takes about three weeks.

The thesis is structured as follows:

Backtrack Algorithms: In Chapter 2, we give a formal description of backtrack algorithms for certain constraint satisfaction problems and describe the main optimisation techniques.

2-dimensional affine subspaces in $\mathbb{F}_2^n$: In Chapter 3, we take a closer look at 2-dimensional afine subspaces contained in arbitrary subsets of $\mathbb{F}_2^n$. It turns out that the APN property restricts the local behaviour of a function on all such subspaces, and we can use the generic results about these subspaces in the evaluation of the performance of a backtrack search for APN functions.

Classification: In Chapter 4, we give a complete classification of all APN functions up to dimension 5 with respect to three equivalence relations preserving the APN property: Affine equivalence (permutations only), EA equivalence and CCZ equivalence.

1.2.1 Main contributions

We consider the main contributions of this thesis to be:

- A complete classification of all APN functions in $\mathbb{F}_2^n$ with $n \leq 5$ with regards to (extended) affine and CCZ equivalence (Chapter 4).
- A constructive method to enumerate the 2-dimensional affine subspaces in an arbitrary subset of $\mathbb{F}_2^n$ and an exact upper bound on their number (Section 3.4).
- A formalisation of backtrack algorithms which unifies different existing optimisation techniques (Section 2.3.3).
- An efficient filter predicate for the APN property, to be used in backtrack algorithms (Section 4.1.2) and a model to estimate its performance (Section 4.1.3).
- An efficient filter predicate for canonical elements of affine equivalence classes (Section 4.2.2).
- An efficient algorithm to test for CCZ equivalence (Section 4.3.1).

1.2.2 Notation

In the text, we identify vectors $a \in \mathbb{F}_2^n$ with binary numbers $(a_{n-1} \dots a_0)_2 = \sum_i a_i 2^i \in [0; 2^n - 1] \subset \mathbb{N}_0$. We use $\oplus, \ominus$ as symbols for the addition in $\mathbb{F}_2^n$ and $+, -$ as symbols for the addition and subtraction in $\mathbb{N}_0 \subset \mathbb{Z}$.

Note 1.1 *It is well known that a function from $\mathbb{F}_2^n$ to itself can be seen as a polynomial on the finite field $\mathbb{F}_{2^n}$; the algebraic degree is an EA (but not CCZ) invariant.*

1.3 Future Work

The results in this thesis are self-contained, but raise a number of new questions and lead to opportunities for further research, for example:

- APN permutations exist for odd dimensions but it is an open question if they also exist for even dimensions. This suggests that the structure of APN functions in odd dimensions is somewhat different from even dimensions, which limits the comparibility of the solutions for $n = 4$ and $n = 5$. Thus, it would be very interesting to have a complete classification for the case $n = 6$. It has to be investigated which additional or different techniques are necessary to find the solution. Alternatively, one can try to find partial (existence) results if a complete classification is intractable.

- The optimisation techniques described in this thesis are very generic, and can be applied readily to other, similar constraint satisfaction problems.

  A function is called *crooked* if for all $c \in \mathbb{F}_2^n$, $c \neq 0$, the set $\{f(x)+f(x+c) \mid x \in \mathbb{F}_2^n\}$ is an affine hyperplane in $\mathbb{F}_2^n$. It is easy to see that crooked is an EA invariant and implies APN (the converse is false).

  It is an open question whether all crooked functions are quadratic. If the case $n = 6$ is too hard to solve for APN functions, maybe it can be solved for related, more constraint problems like this.

- The notion of CCZ equivalence is interesting in its own right, and it would be good to know more about CCZ equivalence classes in the set of all vectorial boolean functions, for example their number and sizes.

Chapter 2

Backtrack Algorithms

In this chapter we will first develop the formal framework for backtrack programming and then describe how backtrack algorithms can be used to solve generic constraint satisfaction problems. We are interested in enumerating all functions $\mathcal{F}(A, B)$ with certain properties, where A and B are finite, ordered sets. Later on we will always set $A = B = \mathbb{F}_2^n$, but the more general case can be treated without any extra effort and prevents possible confusion by using a co-domain distinct from the domain.

2.1 Partial functions

In the discussion of backtrack algorithms it is neccessary to be able to reason about functions between finite, ordered sets whose images are only determinate on a subset of the domain.

We describe two ways to interpret such partial functions: First, a *procedural* view, in which a partial function is an array indexed by the domain which contains entries from the codomain extended by a marker $\diamond$ for indeterminate positions. In this picture, a partial function can be made more determinate by substituting the value $\diamond$ at an indeterminate position by any value from the original codomain.

Second, we have a *functional* view, in which a partial function represents the set of all functions that agree on the determinate positions of the domain (cf. Faradžev's fragments in [Far78]). In this case, a partial function can be made more determinate by taking the subset of all functions that take the same fixed value at the position made determinate.

Both interpretations of partial functions are tightly linked, and their relationship allows us to connect the procedural language of elementary computer programming seamlessly with the functional language of mathematics.[1] We will switch views frequently, choosing whichever is more efficient in any given situation.

[1] Recall that (mathematical) functions return values but are side-effect free, while (algorithmic) procedures do not return values and are only invoked for their side-effects. Low-level machine instructions are procedures that have the side-effect of causing a change in the machine configuration, see [Tur36].

2.1.1 Function templates

We begin with the procedural view on partial functions. For any set A let $\tilde{A} := A \uplus \{\diamond\}$ be the disjoint union of A and $\diamond$. The value $\diamond$ is called the *indeterminate value*.

Definition 2.1 (Function templates)
Let A, B be finite, ordered sets. By a function template from the domain A to the codomain B *we mean a function $\tilde{f} : \tilde{A} \to \tilde{B}$ with $\tilde{f}(\diamond) = \diamond$. The set of all such templates will be denoted by $\tilde{\mathcal{F}}(A, B)$.*

We define the determinate positions $D_{\tilde{f}}$ *as the set $D_{\tilde{f}} := \tilde{f}^{-1}(B) \subseteq A$. Its cardinality is called the* degree of determination $\deg \tilde{f} := \#D_{\tilde{f}}$. *The template $\tilde{f}$ is* injective *if its restriction $\tilde{f} \mid D_{\tilde{f}}$ is injective.*

We further define the indeterminate positions $I_{\tilde{f}} := A \setminus D_{\tilde{f}}$ *as the complement of the set of determinate positions in A. Its cardinality is called the* co-degree $\operatorname{codeg} \tilde{f} := \#A - \deg \tilde{f}$.

If the degree of $\tilde{f}$ takes the maximal value $\#A$, the template has no indeterminate positions and is said to be determinate. *On the contrary, the template $\tilde{\diamond} : \tilde{A} \to \tilde{B}, \tilde{\diamond} \equiv \diamond$, is the* fully indeterminate *template with the minimal degree* 0.

If $\tilde{f}$ is a function template from A to B and $\tilde{g}$ is a function template from B to C, then the concatenation $\tilde{g}\tilde{f}$ is a function template from A to C (for us, that is the only reason to include $\diamond$ in the domain). We note that all function templates from A to A form a monoid.

Given an indeterminate function template, new templates can be constructed by substituting an indeterminate value in the template by a determinate value. This process is called refinement:

Definition 2.2 (Refinement of templates)
Let $\tilde{f}, \tilde{g} : \tilde{A} \to \tilde{B}$ be templates. The template $\tilde{g}$ is a one-step refinement *of $\tilde{f}$ if $\tilde{f}$ and $\tilde{g}$ agree on all positions except one indeterminate position of $\tilde{f}$, i. e. if there exist $a \in A$ and $b \in B$ such that $\tilde{f}(a) = \diamond$ and for all $i \in A$:*

$$\tilde{g}(i) = \begin{cases} \tilde{f}(i) & : \quad i \neq a \\ b & : \quad i = a \end{cases}$$

In this case, we will write $\tilde{g} = \diamond_{a \mapsto b} \tilde{f}$, where $\diamond$ is called the refinement operator.

*The template $\tilde{g}$ is said to be a k-*step refinement *of $\tilde{f}$, $k \in \mathbb{N}_0$, if there exists a sequence of k one-step refinements such that $\tilde{g} = \diamond_{a_k \mapsto b_k} \cdots \diamond_{a_1 \mapsto b_1} \tilde{f}$. The k-tuple $((a_1, b_1), \ldots, (a_k, b_k))$ is the* refinement sequence *for $\tilde{g}$ from $\tilde{f}$.*

We define $\diamond_* \tilde{f}$ as the set of all k-step refinements of $\tilde{f}$ where $k \in \mathbb{N}_0$ arbitrary.

The refinement operator $\diamond$ can be seen as a restriction of the substitution operator $S : \tilde{\mathcal{F}}(A, B) \times A \times B \to \tilde{\mathcal{F}}(A, B)$, $(\tilde{f}, a, b) \mapsto \tilde{g}$, which substitutes the value of $\tilde{f}$ at the position a by b unconditionally. The substitution operator corresponds to an elementary state transition in a computer system that represents a memory store operation. This illustrates the importance of the refinement operator in the procedural interpretation of partial functions.

Clearly, any refinement sequence for $\tilde{g}$ from $\tilde{f}$ is uniquely determined up to the order of its elements, because the refinement operators commutate. This motivates the following definition:

Definition 2.3 (Left-refinement)
Let $\tilde{g}$ be a one-step refinement of $\tilde{f}$ such that $\tilde{g} = \diamond_{a\mapsto b}\tilde{f}$. Then $\tilde{g}$ is called a one-step left-refinement *of $\tilde{f}$ if $\tilde{f}(i) \neq \diamond$ for all $i < a$. In this case, we will write $\tilde{g} = \diamond^{\ell}_{a\mapsto b}\tilde{f}$.*

*The template $\tilde{g}$ is said to be a k-*step left-refinement *of $\tilde{f}$, $k \in \mathbb{N}_0$, if there exists a sequence of k one-step left-refinements such that $\tilde{g} = \diamond^{\ell}_{a_k\mapsto b_k} \cdots \diamond^{\ell}_{a_1\mapsto b_1}\tilde{f}$.*

We define $\diamond^{\ell}_{*}\tilde{f}$ as the set of all k-step left-refinements of $\tilde{f}$ where $k \in \mathbb{N}_0$ arbitrary.

We say that $\tilde{g}$ is a left-refined *template if $\tilde{g} \in \diamond^{\ell}_{*}\tilde{\diamond}$.*

In any step of a left-refinement the indeterminate value with the lowest position is made determinate. If a template $\tilde{g}$ is a left-refinement of a template $\tilde{f}$, then the left-refinement sequence is uniquely determined.

2.1.2 Compatible functions

We now proceed with the functional view on partial functions and develop the relationship between functions and function templates.

Definition 2.4 (Compatible functions)
Let $\tilde{f} : \tilde{A} \to \tilde{B}$ be a template and $f : A \to B$ be a function. Then f and $\tilde{f}$ are said to be compatible *if they agree on all determinate positions of $\tilde{f}$, i. e. if for all $a \in A$ with $\tilde{f}(a) \neq \diamond$ we have that $f(a) = \tilde{f}(a)$.*

The set of all functions compatible with $\tilde{f}$ will be denoted by $\diamond_{\bullet}\tilde{f}$.

Let $\tilde{f}$ be a determinate template and f be a function compatible with $\tilde{f}$. Then it holds that $f \equiv \tilde{f}|A$ and thus $\diamond_{\bullet}\tilde{f}$ consists of a single element. In this case, we will simply identify $\tilde{f}$ with f and write $\tilde{f} = f$.

The following lemma illustrates why $\diamond$ is called the refinement operator.

Lemma 2.5 *Let $\tilde{f} : \tilde{A} \to \tilde{B}$ be an indeterminate template and $a \in A$ be an indeterminate position of $\tilde{f}$. Then:*

1. $\diamond_{\bullet}\diamond_{a\mapsto b}\tilde{f} = \{f \in \diamond_{\bullet}\tilde{f} \mid f(a) = b\}$ *for all* $b \in B$

2. *The template $\tilde{f}$ has $\#B$ different one-step left-refinements which partition the set of compatible functions:*
$$\diamond_{\bullet}\tilde{f} = \biguplus_{b\in B} \diamond_{\bullet}\diamond_{a\mapsto b}\tilde{f} \tag{2.1}$$

3. *The number of functions compatible with $\tilde{f}$ is $\#\diamond_{\bullet}\tilde{f} = \#B^d$ where $d := \operatorname{codeg}\tilde{f}$. It follows that the partitions in (2.1) have equal size.*

2.1.3 Examples

We conclude this section with some elementary examples. Further applications of function templates can be found in Section 2.3 and in all backtrack algorithms throughout.

Example 2.6 *Let $f : A \to B$ be a function. The templates that are compatible with f can be mapped to the selector functions $\mathcal{F}(A, \{\text{FALSE}, \text{TRUE}\})$ in the following way: $\tilde{f} \mapsto s \in \mathcal{F}(A, \{\text{FALSE}, \text{TRUE}\})$ if and only if for all $a \in A$:*

$$s(a) \begin{cases} \text{TRUE} & : \quad \tilde{f}(a) = \diamond \\ \text{FALSE} & : \quad \textit{else} \end{cases}$$

Thus there are $2^{\#A}$ templates compatible with f.

Example 2.7 *A k-step refinement increases the degree of determination by k. Therefore every determinate template in $\tilde{\mathcal{F}}(A, B)$ is a $\#A$-step refinement of the fully indeterminate template $\tilde{\diamond}$.*

In particular, every function $s : \mathbb{F}_2^n \to \mathbb{F}_2^n$ is a 2^n-refinement of $\tilde{\diamond}$. Its unique left-refinement sequence is $((0, s(0)), \ldots, (2^n - 1, s(2^n - 1)))$, and any permutation of this is another refinement sequence for s.

2.2 Index function

The following definition allows us to represent a finite, ordered set by natural numbers. It is used for notational convenience.

Definition 2.8 (Index function)
Let A be a finite, ordered set. Then the index function $\text{idx}_A : A \mapsto [0, \#A - 1] \subset \mathbb{N}_0$ *is defined by* $\text{idx}(a) := \#\{i \in A \mid i < a\}$. *We write* idx *for* idx_A *if the domain is determined by the context.*

We write $A[i] := \text{idx}_A^{-1}(i)$ for the i-th element in A.

The index function yields the position of its argument in the given order of the set, starting from 0 for the smallest element.[2]

Example 2.9 *Let A, B be finite, ordered sets. For $f \in \mathcal{F}(A, B)$ let $f_i := \text{idx}_B f(A[i])$ for $0 \le i < \#A - 1$. The set $\mathcal{F}(A, B)$ has $\#B^{\#A}$ elements and can be identified with the set $[0, \#B^{\#A} - 1] \subset \mathbb{N}_0$ by means of the bijection:*

$$f \mapsto (f_{\#A-1} \ldots f_0)_{\#B} \tag{2.2}$$

This shows that enumerating the functions in $\mathcal{F}(A, B)$ is equivalent to listing the natural numbers from 0 to $\#B^{\#A} - 1$ to the base $\#B$.

Example 2.10 *Let A, B be finite, ordered sets. Then $\tilde{g} \in \tilde{\mathcal{F}}(A, B)$ is left-refined if and only if $D_{\tilde{g}} = \text{idx}^{-1}([0; \deg \tilde{g} - 1]) = \{A[0], \ldots, A[\deg \tilde{g} - 1]\}$.*

[2] Real Programmers count from zero.

2.3 Backtrack

Several problems solved in this thesis are constraint satisfaction problems of a combinatorial nature, such as finding all vectorial boolean functions that satisfy the APN property. We assume that for a complete classification of all such functions an exhaustive search is the appropriate method, for example because no algebraic solution in closed form is known for the constraints that must be satisfied.

The number of vectorial boolean functions is $\#\mathcal{F}(\mathbb{F}_2^n, \mathbb{F}_2^n) = (2^n)^{2^n}$ of which $2^n!$ are permutations. Super-exponential growth of this kind precludes linear search (a. k. a. *brute force*) algorithms if $n > 3$. It is therefore necessary to partition the set of all functions into suitable disjoint subsets and make statements about all members of a subset at once.

A first idea would be to enumerate equivalence classes according to some interesting stability property and analyse only a single representantive from each class. However, this approach can fail if it is computationally hard to enumerate exactly one representantive from each class, or if there are still too many equivalence classes. For example, no efficient method to list all CCZ equivalence classes in $\mathcal{F}(\mathbb{F}_2^n, \mathbb{F}_2^n)$ is currently known, and the number of affine equivalence classes is huge for $n > 4$, see Section 4.2.3.

2.3.1 Basic backtrack algorithm

To avoid linear search, it is necessary to partition the search space into coarser sets than equivalence classes given naturally by interesting mathematical properties. If we succeed in excluding all members of such a coarser subset with an efficient test, we might be able to overcome the limitations of linear search. On the other hand, if the partitions are too coarse we might not be able to make specific interesting statements about their members. This suggest an adaptive approach where we start with coarse partitions which are then step-wise refined. This is the core idea of backtrack programming which has been independently developed and applied by many people. Early descriptions can be found in [Leh57] and [Wal60], and the first formal treatment is due to [GB65].

Definition 2.11 (Backtrack problem)
A backtrack problem P *is a tuple* (A, B, ρ, ϕ) *where* A *and* B *are finite, ordered sets,* $\rho : \mathcal{F}(A, B) \to \{\text{TRUE}, \text{FALSE}\}$ *is the* result predicate *and* $\phi : \diamondsuit_*^\ell\tilde{\diamond} \to \{\text{TRUE}, \text{FALSE}\}$ *is the* filter predicate *satisfying the following condition for all* $f \in \mathcal{F}(A, B)$*:*

$$\rho(f) = \text{TRUE} \iff (\phi(\tilde{f}) = \text{TRUE} \text{ for all } \tilde{f} \in \diamondsuit_*^\ell\tilde{\diamond} \text{ with } f \in \diamondsuit_\bullet\tilde{f}) \tag{2.3}$$

The solution $L(P) \subseteq \mathcal{F}(A, B)$ *of the backtrack problem* P *is the set* $L(P) := \rho^{-1}(\text{TRUE})$.

The domain A, usually an index set, and the codomain B specify the search space $\mathcal{F}(A, B) = \diamondsuit_\bullet\tilde{\diamond}$ of the problem. The result predicate indicates the functions that are considered to be a solution to the problem, and the filter predicate $\phi(\tilde{f})$ provides an upper bound (with FALSE $<$ TRUE) of the values $\rho(f)$ for all functions f compatible with $\tilde{f}$.

The backtrack problem has a natural interpretation as a tree structure. The associated tree consists of the set of all left-refined templates $\diamondsuit_*^\ell\tilde{\diamond}$ as nodes, and all one-step

left-refinements $\tilde{g} = \diamond^{\ell}_{A[i]\mapsto g_i}\tilde{f}$ as edges from $\tilde{f}$ to $\tilde{g}$ labeled with g_i where i is simply the depth of $\tilde{f}$ in the tree. The root of the tree is $\tilde{\diamond}$, and the leaves are the fully determinate functions $\diamond_\bullet\tilde{\diamond}$. In fact, the tree is a complete, ordered tree of order $\#B$ with height $\#A$ and thus the number of nodes is:

$$T_{\#B}(\#A) := \sum_{i=0}^{\#A} \#B^i = \begin{cases} (\#B^{\#A+1}-1)/(\#B-1) & : \quad \#B \neq 1 \\ \#A+1 & : \quad \#B = 1 \end{cases} \tag{2.4}$$

The result predicate ρ labels all leaves of the tree with a boolean value. The filter ϕ labels each node of the tree with a boolean value with the property that a path from the root to a leaf is labeled with TRUE at every node if and only if the leaf is labeled TRUE by ρ.

Note that in the literature (e. g. [Knu75], but not [Far78]) usually the stricter requirement is made for ϕ that for all $\tilde{g} \in \diamond^{\ell}_{*}$ it holds that:

$$\phi(\tilde{g}) = \text{TRUE} \iff (\phi(\tilde{f}) = \text{TRUE for all } \tilde{f} \in \diamond^{\ell}_{*}\tilde{\diamond} \text{ with } \tilde{g} \in \diamond^{\ell}_{*}\tilde{f}) \tag{2.5}$$

Then $\phi \mid \mathcal{F}(A,B) \equiv \rho$ and the result predicate is implicit in ϕ. However, the weaker form above is sufficient for Algorithm 1 and more convenient, because it allows to optimise the filter predicate to check only for new failure conditions that result from the last one-step left-refinement that lead to $\tilde{f}$. If desired, the stricter filter predicate can be recovered from ϕ by defining for $\tilde{f} = \diamond^{\ell}_{A[k]\mapsto f_k} \cdots \diamond^{\ell}_{A[0]\mapsto f_0}\tilde{\diamond}$:

$$\phi^S(\tilde{f}) := \bigwedge_{i=0}^{k} \phi\left(\diamond^{\ell}_{A[i]\mapsto f_i} \cdots \diamond^{\ell}_{A[0]\mapsto f_0}\tilde{\diamond}\right) \tag{2.6}$$

With this stricter filter predicate, Condition 2.3 in Definition 2.11 is equivalent to:

$$\rho \equiv \phi^S \mid \mathcal{F}(A,B) \tag{2.7}$$

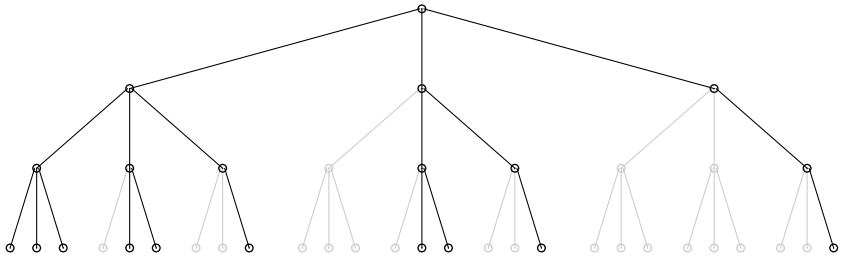
The following definition illustrates what the stricter requirement (2.5) means for the associated tree:

Definition 2.12 (Active search tree)
Let P be a backtrack problem. Then the active search tree *T is the subtree of the associated tree spanned by the nodes that are labeled* TRUE *by ϕ and which have only ancestors labeled* TRUE *by ϕ. That means, using (2.6):*

$$T := \{\tilde{f} \mid \phi^S(\tilde{f}) = \text{TRUE}\} \tag{2.8}$$

The backtrack problem is solved by a pre-order walk through the associated tree, thus enumerating the left-refinements $\tilde{f}$ of $\tilde{\diamond}$ and evaluating the filter predicate $\phi(\tilde{f})$ at each step. If the filter yields FALSE, the result predicate yields FALSE for all functions f compatible with $\tilde{f}$. In this case the template is not compatible with any solution and thus no further refinement is necessary. At this point, a different refinement of an earlier considered template has to be chosen (this step is called *backtrack* and gives the algorithm its name). However, if we reach a leaf node f because ϕ yields TRUE for all

Figure 2.1: An example search tree



The optimal, ordered search tree for a backtrack problem P which lists all multi-sets with k out of three elements, where k is the depth of the tree. The active search tree is shown in black, nodes labeled with FALSE are shown in gray. All other labels have been omitted.

nodes from the root to the leaf, we know that $\rho(f) = \text{TRUE}$ and we found a solution. It is clear that this search visits exactly all nodes of the active search tree and their immediate children.

In the worst case the filter ϕ is TRUE on all indeterminate templates, and all $T_{\#B}(\#A) - \#B^{\#A} = (\#B^{\#A} - 1)/(\#B - 1)$ internal nodes have to be visited in addition to the leaves (for these examples, let $\#B \neq 1$). This is worse than linear search, so in this case backtrack is a failure. However, if $\phi(\tilde{f}) = \text{FALSE}$ for a template $\tilde{f}$ with co-degree d, then $T_{\#B}(d) - 1$ nodes are pruned (a complete subtree of height d excluding its root). For example, if $d = \#A - 1$ then the excluded subtree has $T_{\#B}(\#A-1) = (\#B^{\#A}-1)/(\#B-1)$ nodes, thereby completely amortizing the cost of backtrack. In the best case there exists a path from every node $\tilde{f}$ labeled $\phi(\tilde{f}) = \text{TRUE}$ to a leaf node $f \in \diamond^{\ell}_{*}\tilde{f}$ with $\rho(f) = \text{TRUE}$. Then no dead-ends (nodes that do not lead to a solution) are followed and the active search tree has minimal size.

Algorithm 1 Solve the backtrack problem $P := (A, B, \rho, \phi)$

```
procedure BACKTRACK(f̃)
    if φ(f̃) = TRUE then
        if codeg f̃ = 0 then
            OUTPUT(f̃)                                  ▷ Found solution f̃ = f
        else
            for all b ∈ B do
                BACKTRACK (◇^ℓ_{A[deg f̃]↦b} f̃)
            end for
        end if
    end if
end procedure

BACKTRACK (◇̃)                                          ▷ Invocation
```

Algorithm 1 solves the backtrack problem $P := (A, B, \rho, \phi)$. There are several variations of this algorithm in the literature which will not be used in this thesis. Some

descriptions restrict $f(A[i])$ to different subsets of the codomain $B_i \subseteq B$, $0 \leq i < \#A$. This can easily be handled by replacing the set B in line 6 with $B_{\deg \tilde{f}}$. Some allow to select other refinements than left-refinements at any step. In Lemma 4.8 we will show that left-refinements are an appropriate choice for the specific problems we deal with in this thesis. Some variations limit the codomains such that "isomorph" solutions can not occur more than once. We will make that – just like preclusion – an obligation of the filter predicate in Section 4.2.

2.3.2 Problem representation and representation problems

There is substantial freedom in the representation of the search problem by the sets A, B and in the choice of ϕ for internal nodes. We will now discuss the choices we make for the applications in this thesis and give some necessary conditions that the problems must fulfil for the chosen representation to be appropriate. Later on we will demonstrate that these conditions are fulfilled, and of course we find further justification after the fact in producing the actual results.

There are several interesting non-trivial ways to represent vectorial boolean functions, for example in algebraic normal form or as Walsh-transform, but it is not clear how such representations could support efficient filter predicates for the constraints we are trying to satisfy. In particular, the APN property is defined by the additive structure, while the algebraic normal form and the Walsh-transform are based on multiplication. Therefore it seems appropriate to choose the function look-up table as the most natural representation.

Care has to be taken if one wants to satisfy several different constraints at the same time, for example when one wants to find all bijective APN functions, or only one APN function in each equivalence class according to some equivalence relation. In this case, the same representation has to support efficient filters for any of the desired constraints to achieve maximum impact on pruning.

We will use backtrack problems $(\mathbb{F}_2^n, \mathbb{F}_2^n, \rho, \phi)$, and directly identify a template $\tilde{s} \in \tilde{\mathcal{F}}(\mathbb{F}_2^n, \mathbb{F}_2^n)$ with a vectorial boolean function template. This choice can only be justified by its consequences which will be demonstrated further below. The following note provides a basis for evaluation.

Note 2.13 (Evaluation of problem representation)
When selecting filters ϕ for a backtrack problem P as above, the following two requirements should be fulfilled as far as possible:

1. *The filter should have a high probability to fail function templates $\tilde{f}$ with a low degree of determination. This ensures that backtrack is applicable to the problem in the first place.*

2. *For any template $\tilde{f}$ and any compatible function f the one-step left-refinement with $f \in \diamond^{\ell}_{a \mapsto f(a)} \tilde{f}$ should have a high probability of satisfying the constraints less (or equally) than any other one-step refinement with $f \in \diamond_{a' \mapsto f(a')} \tilde{f}$. This ensures that choosing left-refinements in the recursive descend is not a limitation.*

The requirements in Note 2.13 are only heuristic. The filter will often have a higher probability to fail templates with a high degree of determination because then more information is available about the compatible functions. Also, refinement sequences other than left-refinements may lead at times to a better pruning of the search tree. However, in the absence of better global knowledge about the behaviour of the filter predicate with regards to the refinement sequence chosen, the heuristics above have proven to be useful in practice.

2.3.3 Backtrack with stateful filters

A typical application of backtrack programming that we need later on is generating all permutations in S_n for $n \in \mathbb{N}_0$.

The filter ϕ_σ is checking the constraint that no value from 0 to $n-1$ occurs twice in the image of the permutation template. However, it is unnecessarily expensive to perform this check given only the currently considered template. Instead, we could use the information obtained by earlier refinements to avoid assigning an already assigned value to another position as well. This enhancement of backtrack is called *preclusion* [Leh57] because assigning a value $f(a)$ to the position a precludes that the same value $f(a)$ is assigned to any other position. More generally, preclusion can be seen as a filter predicate that is optimised by carrying along additional state beside $\tilde{f}$ throughout the recursion. Modifications to this state must then be rolled back when backtrack occurs. Such additional state can also be used to avoid repetitive calculation in other filters which traditionally would not be considered preclusions.

Definition 2.14 (Backtrack problem with stateful filter)
A backtrack problem P with a stateful filter ϕ *is a tuple* $(A, B, \rho, \phi, \Sigma, S_{\tilde{\diamond}})$ *where* A, B *are finite, ordered sets,* $\rho : \mathcal{F}(A,B) \to \{\text{TRUE}, \text{FALSE}\}$ *is the* result predicate, Σ *is a finite* set of states *with* $\text{FALSE} \in \Sigma$, *and* $S_{\tilde{\diamond}} \in \Sigma$ *is the* initial state. *Finally,* ϕ *is the* stateful boolean filter predicate $\phi : \Sigma \times B \to \Sigma$ *such that* $\hat{P} := (A, B, \rho, \hat{\phi})$ *with* $\hat{\phi}$ *as defined below is a backtrack problem.*

The filter ϕ *induces a function* $S_\phi : \diamondsuit^\ell_* \tilde{\diamond} \to \Sigma$ *as follows, writing* $\phi_b(S)$ *for* $\phi(S, b)$*:*

$$S_\phi(\tilde{f}) := \phi_{f_{k-1}} \cdots \phi_{f_0} S_{\tilde{\diamond}} \quad \text{for} \quad \tilde{f} = \diamondsuit^\ell_{A[k-1]\mapsto f_{k-1}} \cdots \diamondsuit^\ell_{A[0]\mapsto f_0} \tilde{\diamond} \tag{2.9}$$

The function S_ϕ *in turn induces a function* $\hat{\phi} : \diamondsuit^\ell_* \tilde{\diamond} \to \{\text{TRUE}, \text{FALSE}\}$ *with*

$$\hat{\phi}(\tilde{f}) := \begin{cases} \text{FALSE} & : \quad S_\phi(\tilde{f}) = \text{FALSE} \\ \text{TRUE} & : \quad \text{else} \end{cases} \tag{2.10}$$

The solution $L(P) \subseteq \mathcal{F}(A,B)$ *of the backtrack problem* P *with a stateful filter is* $L(P) := \rho^{-1}(\text{TRUE})$.

Although its definition is complex, a stateful filter is conceptually very easy: Each node $\tilde{f}$ in the associated tree of the backtrack problem is labeled by ϕ with a state $S_\phi(\tilde{f})$. The state of a node and the label of the edge to one of its children are used to calculate the state of that child. Note that in this formalisation all knowledge about $\tilde{f}$ that ϕ can access is subsumed by $S_\phi(\tilde{f})$ in cause for a lighter notation. This is not a loss

of generality, as it is always possible to expand the state space by $\Sigma' := \Sigma \times \diamondsuit_*^\ell \tilde{\diamond}$ and removing all unreached states. In an actual implementation this overhead can easily be avoided (see Section A.2).

The construction of $\hat{P}$ in the definition contains a simulation of the backtrack problem $\hat{P}$ using P. Both have the same solution set, but their computational complexity is very different. In the simulation, $\phi_b(S_\phi(\tilde{f}))$ is evaluated once for every descendant node $\tilde{g} \in \diamondsuit_*^\ell \tilde{f}$ visited by the backtrack search. Algorithm 2 solves the backtrack problem with a stateful filter $P := (A, B, \rho, \phi, \Sigma, S_{\tilde{\diamond}})$ much more efficiently, evaluating $\phi_b(S_\phi(\tilde{f}))$ only once for every visited node.

Algorithm 2 Solve the backtrack problem $P := (A, B, \rho, \phi, \Sigma, S_{\tilde{\diamond}})$. Note that ϕ_b is never invoked with the argument $S = \text{FALSE}$.

```
procedure BACKTRACK(f̃, S)
    if S ≠ FALSE then
        if codeg f̃ = 0 then
            OUTPUT(f̃)                                   ▷ Found solution f̃
        else
            for all b ∈ B do
                BACKTRACK (◇^ℓ_{A[deg f̃]↦b} f̃, φ_b(S))
            end for
        end if
    end if
end procedure

BACKTRACK (◇̃, S_◇̃)                                      ▷ Invocation
```

To carry on with the permutation example, the stateful filter ϕ_σ can keep track of the (determinate) image of $\tilde{\sigma}$ as a subset of B. The set of states is $\Sigma := \wp(B) \cup \{\text{FALSE}\}$, the initial state is $S_{\tilde{\diamond}} := \emptyset$, and then we let:

$$\phi_b(S) := \begin{cases} \text{FALSE} & : \quad b \in S \\ S \cup \{b\} & : \quad \text{else} \end{cases}$$

In this example every one-step refinement will purge exactly one descendant sub-tree from every inner node of the search tree without regards to which value is substituted for $\diamond$ at which position. Thus, the probability that the filter fails $\tilde{\sigma}$ is $\deg \tilde{\sigma} / \# B$. Furthermore, the order of refinements does not matter and a left-refinement is as good as any other choice; there is no prejudice towards the choice of position or value in selecting the next refinement that satisfies the constraints the least. This means that both requirements in Note 2.13 are fulfilled.

2.4 Isomorph rejection in backtrack algorithms

The constraints satisfied by the solutions of a backtrack problem are often stable under some transformations. For example, the fact that a configuration of n queens on an $n \times n$ chess board solves the classical n queens puzzle is stable under the action of the symmetry group D_8 of the (uncoloured) chess board. The orbits under the group action partition the search space and thus induce an equivalence relation. The modified constraint satisfaction problem then becomes the problem of finding exactly one representative from each equivalence class satisfying the solution. In principle, all solutions can then be recovered by reconstructing all members of the equivalence class given by a specific representative. By pruning those subtrees from the search tree which do not contain any representatives, the search space and the solution set can be vastly reduced in size. Filter predicates with this property will be called *canonicity filters* (see Definition 2.17 below).

2.4.1 Canonicity filter

Isomorph rejection in backtrack programming has been studied extensively in conjunction with combinatorial problems in graph theory. A simple technique that requires only constant space was developed independently several times and first published by Faradžev [Far78] and Read [Rea78]. This method is based on defining a canonical representative that is extremal among all members of the same equivalence class under some order. If a bound can be calculated for the index of all functions compatible to a template, it may be possible to fail the template and prune all its refinements in a backtrack search.

Definition 2.15 (Lexicographical order)
Let A and B be finite, ordered sets. With the notation from Example 2.9, p. 8, the orders on A, B and $\mathbb{N}_0$ induce an order $<$ on $\mathcal{F}(A,B)$ in the following way:

$$f < g \iff (f_{\#A-1} \dots f_0)_{\#B} < (g_{\#A-1} \dots g_0)_{\#B} \tag{2.11}$$

We extend this to a partial order $<$ on $\tilde{\mathcal{F}}(A,B)$ by letting $\tilde{f} < \tilde{g}$ if and only if $f < g$ for all $f \in \diamond_\bullet \tilde{f}$ and $g \in \diamond_\bullet \tilde{g}$.

Note that the partial order in the definition coincedes with the total lexicographic order on all functions in $\mathcal{F}(A,B)$ defined above. We use this order to define the canonical representatives as the smallest member of each equivalence class.

Definition 2.16 (Canonical representatives)
Let A, B be finite ordered sets. Let $\simeq$ be an equivalence relation on $\mathcal{F}(A,B)$. Then the canonical set of representatives for $\simeq$ *is the set:*

$$R_\simeq := \{f \in \mathcal{F}(A,B) \mid f \leq f' \text{ for all } f \simeq f'\} \tag{2.12}$$

Definition 2.17 (Canonicity filter)
Let A, B be ordered sets and $\simeq$ an equivalence relation on $\mathcal{F}(A,B)$. Define the result predicate $\rho_\simeq$ by:

$$\rho_\simeq(f) = \text{TRUE} \iff f \in R_\simeq \tag{2.13}$$

Then $\phi_\simeq : \diamond_*^\ell \tilde{\diamond} \to \{\text{TRUE}, \text{FALSE}\}$ *is a* canonicity filter *for* $\simeq$ *if* $(A, B, \rho_\simeq, \phi_\simeq)$ *is a backtrack problem.*

If $P = (A, B, \rho, \phi)$ is a backtrack problem, then $P' = (A, B, \rho', \phi')$ with $\rho'(f) = \rho(f) \wedge \rho_\simeq$ and $\phi'(\tilde{f}) = \phi(\tilde{f}) \wedge \phi_\simeq(\tilde{f})$ is a backtrack problem with the solution $L(P') = L(P) \cap R_\simeq$.

2.4.2 Weak canonicity filter

A canonicity filter fails a template only if the template has no canonical representatives among its compatible functions. We capture this observation in a definition.

Definition 2.18 (Weak canonicity filter)
Let A, B *be ordered sets and* $\simeq$ *an equivalence relation on* $\mathcal{F}(A, B)$*. Then* $\phi_\simeq^w : \diamond_*^\ell \tilde{\diamond} \to \{\text{TRUE}, \text{FALSE}\}$ *is a* weak canonicity filter *for* $\simeq$ *if:*

$$\diamond_\bullet \tilde{f} \cap R_\simeq \neq \emptyset \Rightarrow \phi_\simeq^w(\tilde{f}) = \text{TRUE} \tag{2.14}$$

Note that a weak canonicity filter for $\simeq$ is *not* a canonicity filter for $\simeq$. Equation (2.14) is a necessary but not sufficient condition for a canonicity filter: It ensures that the backtrack search will not suppress valid solutions, but it does not prevent the production of false positives which are not $\simeq$ canonical. That means if $P_\simeq = (A, B, \rho_\simeq, \phi_\simeq)$ and $P_\simeq^w = (A, B, \rho_\simeq^w, \phi_\simeq^w)$ are backtrack problems, we have:

$$L(P_\simeq) \subseteq L(P_\simeq^w) \tag{2.15}$$

We call the elements of $L(P_\simeq^w)$ *candidates*. To recover the solution $L(P_\simeq)$ from the set of all candidates we have to determine in a post-processing step which candidates are $\simeq$ canonical and which are not. This puts weak canonicity filters at a disadvantage. However, the benefit is that they can be implemented more efficiently than proper canonicity filters.

Determining the $\simeq$ canonical candidates efficiently may require elaborate reasoning and manual effort. We will now describe three general techniques that can be used to perform this post-processing. All of these techniques will be applied to specific problems in Section 4.2.4 and Section 4.3.2.

Canonicity: A canonicity filter $\phi_\simeq$ may exhibit a worst case behaviour that makes it unusable in a backtrack problem. But if it has good average case behaviour it may still be useful to reject many candidates. This works particularly well if the canonicity filter is weakened first, for example by forcing it to return TRUE if it does not terminate within a certain time limit or by adding a (possibly human-operated) oracle that provides heuristic guesses. If this weakened filter is stronger than $\phi_\simeq^w$ at least for some candidates, it can be used to reject those.

Equivalence: Even if there is no efficient canonicity filter $\phi_{\simeq}$, there may be an efficient equivalence test $s \simeq t$. In this case, and if the number of candidates is small, the following naive approach at isomorph rejection will perform well: We iterate over the candidates in lexicographical order, beginning with the smallest. For each candidate, we test its equivalences to all known canonical representatives. If the candidate is not equivalent to any of those, it is itself a canonical representative and we add it to the list of known ones to test against. After all candidates have been processed, this list contains all canonical representatives among the candidates. Algorithm 3 implements this method.

Algorithm 3 Naive method to determine a set of canonical representatives $L := L(R_{\simeq})$ from a candidate set $L^w := L(R^w_{\simeq})$.

```
function Can(L^w)
    var L ∈ ℘(F(F_2^n, F_2^n))
    L ← ∅
    for all s ∈ L^w do                          ▷ In lexicographical order
        if s ≄ t for all t ∈ L then
            L ← L ∪ {s}
        end if
    end for

    return L
end function
```

If this technique is used, one further optimisation suggests itself immediately. Because the equivalence test $s \simeq t$ is executed many times for the same $t \in L(R_{\simeq})$, it may be useful to write equivalence tests $s \simeq^t$ which are specific to t. We will discuss this important optimisation in more detail later on in Section 4.2.4 and Section 4.3.2.

Invariants: Let $\chi : \mathcal{F}(A, B) \to X$ be a mapping from all determinate functions to an indicator set X. We say that χ is $\simeq$ *invariant* if it is stable under $\simeq$. In this case $\chi(s) \neq \chi(t)$ implies $s \not\simeq t$ and χ induces a partition on $\mathcal{F}(A, B)$ by its preimages $\chi^{-1}(x)$ for $x \in X$. It is not hard to see that then for Algorithm 3:

$$L(R_{\simeq}) = \mathrm{Can}(L(R^w_{\simeq})) = \biguplus_x \mathrm{Can}(L(R^w_{\simeq}) \cap \chi^{-1}(x)) \tag{2.16}$$

This reduction is beneficial because it lowers the number of equivalence tests $s \simeq t$ that are executed in Algorithm 3 if more than one non-empty χ equivalence class is hit by $L(R^w_{\simeq})$.

Typically we are interested in invariants that can be computed much more efficiently than any canonicity filter or equivalence test, at the cost of being less specific. For example, we already mentioned in Note 1.1 that the algebraic degree of a vectorial boolean function is an extended affine invariant.

2.5 Estimating the efficiency of backtrack algorithms

Note: The content of this subsection is due to an article by D. E. Knuth with the same title [Knu75]. We bring in the results in our notation here, and apply the method to our problems in later sections.

No systematic method to predict the performance of a backtrack algorithm is known. However, there is a method to estimate the computational complexity by sampling random paths in the search tree. The same method can be used to estimate other functions defined on the tree, for example the number of nodes at depth k.

Let P be a backtrack problem, and T be the active search tree as defined in Definition 2.12. Let $c : T \to \mathbb{R}$ be any function defined on T, and define the total of c by:

$$c_T := \sum_{\tilde{f} \in T} c(\tilde{f}) \tag{2.17}$$

The following theorem provides an estimate of c_T based on a single random walk in the tree without backtracking:

Theorem 2.19 *Let $\tilde{f} = \diamond^{\ell}_{A[k-1]\mapsto f_{k-1}} \cdots \diamond^{\ell}_{A[0]\mapsto f_0} \tilde{\diamond} \in \diamond^{\ell}_{*}$ be a node in T, and d_i, $0 \leq i < k$, the number of one-step left-refinements of $\diamond^{\ell}_{A[i-1]\mapsto f_{i-1}} \cdots \diamond^{\ell}_{A[0]\mapsto f_0} \tilde{\diamond}$ in T (that is the number of child nodes for the nodes along the path from the root to $\tilde{f}$). Define:*

$$C_T(\tilde{f}) := \sum_{i=0}^{k} d_0 d_1 \dots d_{k-1} c(\diamond^{\ell}_{A[k-1]\mapsto f_{k-1}} \cdots \diamond^{\ell}_{A[1]\mapsto f_1} \diamond^{\ell}_{A[0]\mapsto f_0} \tilde{\diamond}) \tag{2.18}$$

Pick a random leaf node for $\tilde{f}$. Then the expected value of $C_T(\tilde{f})$ is c_T.

Although the estimates have a high variance in general, they turn out to be rather accurate in practice. The estimate can be improved by taking the average of many samples $C_T(\tilde{f})$ which will approximate c_T due to the law of large numbers.

Example 2.20 *Let $c(\tilde{f}) = 1$ if $\tilde{f}$ is a node at depth k, and 0 otherwise. Then the expected value of $C_T = d_0 d_1 \dots d_{k-1}$ is the number of nodes in T at depth k.*

Chapter 3

Affine subspaces in $\mathbb{F}_2^n$ of dimension 2

The APN property is closely related to the 2-dimensional affine subspaces of $\mathbb{F}_2^n$, as we will see in Theorem 4.3. Knowledge about construction and enumeration of 2-dimensional affine subspaces will allow us to construct and evaluate a stateful filter predicate for a backtrack search for APN functions.

Let $\mathcal{A}(M)$ be the set of all 2-dimensional affine subspaces in $M \subseteq \mathbb{F}_2^n$ (*not* the affine subspace generated by M). The following proposition characterises such subspaces:

Proposition 3.1 (2-dimensional affine subspaces)
The 2-dimensional affine subspaces $\mathcal{A}(M)$ in $M \subseteq \mathbb{F}_2^n$ are the sets $\{t, u, v, w\} \subseteq M$ of four pairwise different vectors with $t \oplus u \oplus v \oplus w = 0$.

Proof: Two vectors $u, v \in \mathbb{F}_2^n \setminus \{0\}$ are linearly independent if and only if $u \neq v$.

"$\subseteq$": A 2-dimensional affine subspace is a set $A = t \oplus L(u, v) = t \oplus \{0, u, v, u \oplus v\}$ where $t, u, v \in \mathbb{F}_2^n$ and u, v are linearly independent. Clearly A consists of four pairwise different vectors whose sum is 0.

"$\supseteq$": Let $A = \{t, u, v, w\} \in \mathbb{F}_2^n$ be a set of four pairwise different vectors with $t \oplus u \oplus v \oplus w = 0$. Then $u \ominus t$ and $v \ominus t$ are linearly independent and $A = t \oplus L(u \ominus t, v \ominus t)$ is a 2-dimensional affine subspace. □

3.1 The case $\mathcal{A}(\mathbb{F}_2^n)$

The cardinality of $\mathcal{A}(\mathbb{F}_2^n)$ can be calculated using gaussian binomials[1] or directly, as in the following proposition.

Proposition 3.2 (Cardinality of $\mathcal{A}(\mathbb{F}_2^n)$)
The number of different 2-dimensional affine subspaces of $\mathbb{F}_2^n$ is:

$$\#\mathcal{A}(\mathbb{F}_2^n) = \frac{1}{4} \cdot \binom{2^n}{3} \tag{3.1}$$

[1] See also sequence A016290 in [Slo06].

Proof: There are $\binom{2^n}{3}$ different sets $\{t, u, v\}$ of three pairwise different vectors in $\mathbb{F}_2^n$. For each of these $w := t + u + v$ is different from t, u and v. Thus $\{t, u, v, w\}$ is a 2-dimensional affine subspace. However, exactly three other sets of three pairwise different vectors result in the same subspace, namely $\{t, u, w\}$, $\{t, v, w\}$ and $\{u, v, w\}$. Thus we have to divide by 4. □

Example 3.3 *There are* $\binom{8}{3}/4 = 14$ *affine subspaces of dimension 2 in* $\mathbb{F}_2^3$*:*

$0,1,2,3$	$0,3,4,7$	$1,3,5,7$
$0,1,4,5$	$0,3,5,6$	$2,3,4,5$
$0,1,6,7$	$1,2,4,7$	$2,3,6,7$
$0,2,4,6$	$1,2,5,6$	$4,5,6,7$
$0,2,5,7$	$1,3,4,6$	

3.2 Decomposition Lemma

To calculate the number $\#\mathcal{A}(M)$ for arbitrary subsets $M \subseteq \mathbb{F}_2^n$, the following Decomposition Lemma can be used. It is preceeded by a characterisation of the hyperplanes in $\mathbb{F}_2^n$.

Definition 3.4 (Hyperplane in $\mathbb{F}_2^n$)
Let $\langle ; \rangle : \mathbb{F}_2^n \times \mathbb{F}_2^n \to \mathbb{F}_2$ *denote the canonical inner product of* $\mathbb{F}_2^n$*. An* affine hyperplane *of* $\mathbb{F}_2^n$ *is an* $n-1$ *dimensional affine subspace* $H_\lambda^a = \{b \mid \langle a; b\rangle = \lambda\}$ *with* $a \in \mathbb{F}_2^n \setminus \{0\}$ *and* $\lambda \in \mathbb{F}_2$*.*

Clearly, two hyperplanes H_0^a and H_1^a partition the set $\mathbb{F}_2^n$ for any $a \in \mathbb{F}_2^n \setminus \{0\}$, that means $\mathbb{F}_2^n = H_0^a \uplus H_1^a$. This partition is the basis for the Decomposition Lemma:

Lemma 3.5 (Decomposition of $A(M)$)
Let $M \subseteq \mathbb{F}_2^n$ *be an arbitrary set and* $a \in \mathbb{F}_2^n$ *with* $a \neq 0$*. Then the hyperplanes* $H_0 := H_0^a$ *and* $H_1 := H_1^a$ *partition* $\mathbb{F}_2^n$*, inducing a partition of* $\mathcal{A}(M)$ *into three disjoint subsets. Let* $\mathcal{A}_+ := \{A \in \mathcal{A}(M) \mid \#(A \cap H_0) = 2\}$*. Then:*

$$\mathcal{A}(M) = \mathcal{A}(M \cap H_0) \uplus \mathcal{A}(M \cap H_1) \uplus \mathcal{A}_+$$

Proof: The direction "$\supseteq$" is trivial. It is also clear that the subsets are disjoint. To see "$\subseteq$", let $A = \{t, u, v, w\} \in \mathcal{A}(M)$. Then $t \oplus u \oplus v \oplus w = 0 \in H_0$. The set H_0 (resp. H_1) is a linear (resp. affine) subspace, so $H_0 \oplus H_0 = H_0$, and it holds that $H_0 \oplus H_1 = H_1$ and $H_1 \oplus H_1 = H_0$ (because $\mathbb{F}_2^n$ has characteristic 2). This means that the number of elements in $A \cap H_1$ must be even and thus 0, 2 or 4. It follows that A is in $\mathcal{A}(M \cap H_0)$, $\mathcal{A}_+$ or $\mathcal{A}(M \cap H_1)$ respectively. □

Any 2-dimensional affine subspace in $\mathbb{F}_2^n$ with $n \geq 2$ can be written as an intersection of $n-2$ affine hyperplanes. Thus the Decomposition Lemma can be used successively to isolate the 2-dimensional affine subspaces in M by appropriate choices of $a \in \mathbb{F}_2^n \setminus \{0\}$.

3.3 The case $\mathcal{A}([0;k-1])$

If some information about the set M is available, the construction of $\mathcal{A}(M)$ can be efficient and explicit, as for example in the proof of the next proposition.

Definition 3.6 *Let $k \in \mathbb{N}_0$, then define $A(k)$ to be the number of 2-dimensional affine subspaces in $[0;k-1]$, that means $A(0) = \#\mathcal{A}(\emptyset) = 0$ and $A(k) := \#\mathcal{A}([0;k-1])$ for $k > 0$.*

Further define the difference $\Delta(k)$ to be the number of 2-dimensional affine subspaces in $[0;k-1]$ that contain the point $k-1$, that means $\Delta(0) := 0$ and $\Delta(k) := A(k)-A(k-1)$ for $k > 0$.

Proposition 3.7 *Let $k \in \mathbb{N}_0$ such that $2 \le k \le 2^n$ and $i \in \mathbb{N}_0$ such that $2^i \le k-1 < 2^{i+1}$. Then for $A(k)$ and $\Delta(k)$ the following recurrence relations hold:*

$$\begin{aligned} A(1) &= 0 \\ A(k) &= A(2^i) + A(k-2^i) + \binom{k-2^i}{2} \cdot 2^{i-1} \\ \Delta(1) &= 0 \\ \Delta(k) &= \Delta(k-2^i) + (k-2^i-1) \cdot 2^{i-1} \end{aligned}$$

Proof: We first show the relation for Δ by induction over k. Clearly $\Delta(2) = 0 = \Delta(1) + 0$. Let now the relation be true for $j < k$. By the Decomposition Lemma 3.5 for $a = 2^i$ we have:

$$\mathcal{A}([0;k-1]) = \mathcal{A}([0;2^i-1]) \uplus \mathcal{A}([2^i;k-1]) \uplus \mathcal{A}_+ \tag{3.2}$$

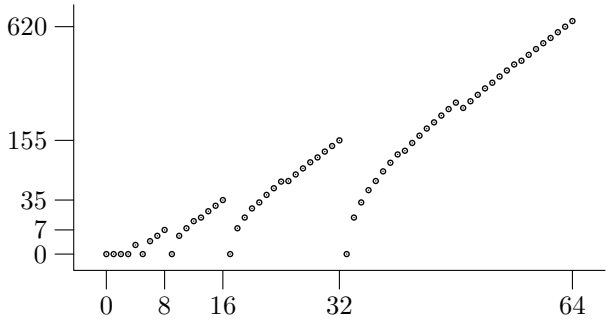
Note that $\Delta(k)$ is the number of 2-dimensional affine subspaces in $\mathcal{A}([0;k-1])$ which contain the point $t := k-1$. The first term on the right hand side of (3.2) contributes none of those and the second term contributes $\Delta(k-2^i)$ by means of the bijection $j \mapsto j \oplus 2^i$.

This leaves $\mathcal{A}_+$: Choose any $u \in [2^i;k-2]$, $v \in [0;2^i-1]$, then $\{t,u,v,t\oplus u\oplus v\} \in \mathcal{A}_+$. This gives $(k-2^i-1)\cdot 2^i$, but choosing $v' = t\oplus u\oplus v$ yields the same solution, so we have to divide by 2. On the other hand, if $\{t,u,v,w\} \in \mathcal{A}_+$, let without loss of generality $v,w \in H_0^{2^i}$ and thus $u \in [2^i;k-2]$, because $u \in H_1^{2^i} \setminus \{t\}$.

We now derive the relation for $A(k)$ directly:

$$\begin{aligned} A(k) &= \sum_{j=0}^{k} \Delta(j) = \sum_{j=0}^{2^i} \Delta(j) + \sum_{j=2^i+1}^{k} \Delta(j) \\ &= A(2^i) + \sum_{j=2^i+1}^{k} \left(\Delta(j-2^i) + (j-2^i-1)\cdot 2^{i-1}\right) \end{aligned}$$

Figure 3.1: The function $\Delta(k)$, square root scale.



$$= A(2^i) + \sum_{j=1}^{k-2^i} \Delta(j) + 2^{i-1} \cdot \sum_{j=0}^{k-2^i-1} j$$
$$= A(2^i) + A(k-2^i) + \binom{k-2^i}{2} \cdot 2^{i-1}$$

□

The integer sequences $A(i)$ and $\Delta(i)$ are, starting with $i = 1$:

$$A(i) : \underline{0}, \underline{0}, 0, \underline{1}, 1, 3, 7, \underline{14}, 14, 18, 26, 39, 55, 77, 105, \underline{140}, \ldots \tag{3.3}$$
$$\Delta(i) : 0, 0, 0, 1, 0, 2, 4, 7, 0, 4, 8, 13, 16, 22, 28, 35, \ldots \tag{3.4}$$

The values $A(2^k)$ are underlined and correspond to the number of two-dimensional affine subspaces in $\mathbb{F}_2^k$. The following equations are easy to verify and show that $A(2^k)$ is cubic in 2^k, $\Delta(2^k)$ is quadratic[2] in 2^k and $\Delta(2^k + j)$ is linear in 2^k for a fixed $0 < j < 2^k$.

$$A(2^k) = \frac{1}{24}\left((2^k - 1)^3 - 2^k + 1\right) \tag{3.5}$$
$$\Delta(2^k) = \frac{1}{6}\left(2^k - \frac{3}{2}\right)^2 - \frac{1}{24} \tag{3.6}$$
$$\Delta(2^k + j) = \Delta(j) + \frac{1}{2}(j-1) \cdot 2^k \tag{3.7}$$

The different growth of $\Delta(2^k)$ and $\Delta(2^k + j)$ has an analogy in the sum $\sum_{i=j}^{k+j} i$ which grows linearly with j but quadratic with k, even if the analytical treatment of the recurrence relation for Δ is more complicated. A quick glance at Figure 3.1 supports the conjecture that Δ is well approximated by (3.6).

3.4 The case $\mathcal{A}(M)$, M arbitrary

The number of affine subspaces of an arbitrary set $M \subseteq \mathbb{F}_2^n$ can be smaller than $A(\#M)$, for example it is zero if M is a basis. The following proposition shows that it can never

[2] See also sequence A006095 in [Slo06].

be greater and thus $A(\#M)$ constitutes an exact upper bound on $\#\mathcal{A}(M)$. To get this result we construct an injective mapping from $\mathcal{A}(M)$ to $\mathcal{A}([0;k-1])$ with $k := \#M$. Unfortunately, this construction is rather delicate. For example, a simple induction over k fails, as applying the Decomposition Lemma to arbitrary sets with k elements yields upper bounds that exceed $A(k)$. Another failure is to successively map elements in M to elements in $[0;k-1]$: There is one 2-dimensional affine subspace in $\{4,5,6,7\}$, but none in any other set which has exactly three points in common with this one.

We combine induction over k for sets which fulfil the additional property of being saturated (see Definition 3.8 below) and small local modifications to unsaturated sets to saturate them.

3.4.1 The case $\mathcal{A}(M)$, M saturated

Definition 3.8 (Gaps and saturated sets)
Let $M \subseteq \mathbb{F}_2^n$ be a set. The gaps of M *is the set $G(M) := \mathbb{N}_0 \setminus M$, and the* minimal gap *is $g := g(M) := \min G(M) \leq 2^n$.*

The set M is saturated *if $M = \emptyset$, or $M = \{0\}$, or otherwise if $\min G(M) \geq 2^i$ with $i := \lfloor \log_2 \max M \rfloor \in \mathbb{N}_0$ being the most significant bit of $\max M$.*

Example 3.9 *The set $M_1 := \{0,1,2,3,5,7\}$ is saturated, while the set $M_2 := \{0,1,3,4,6,7\}$ is not. If one writes the numbers from 0 to $2^i - 1$ below the numbers from 2^i to $2^{i+1} - 1$ and marks those that belong to the set M_j, the following diagrams result. Note that the set is saturated if and only if the lower row is completely marked.*

$M_1 \mathrel{\hat{=}} {}^{4}_{0}$▦ saturated $M_2 \mathrel{\hat{=}} {}^{4}_{0}$▦ not saturated

A different way to look at saturated sets may be helpful to motivate their name: A non-empty set M with $\max M > 0$ and $i := \lfloor \log_2 \max M \rfloor \in \mathbb{N}_0$ is saturated if and only if $M \cap H_0^{2^i} = H_0^{2^i}$. Because of this property saturated sets provide an ideal target for the Decomposition Lemma.

Lemma 3.10 (2-dimension affine subspaces of saturated sets)
Let M be a saturated set, $M \neq \emptyset$, $M \neq \{0\}$ and $i := \lfloor \log_2 \max M \rfloor \in \mathbb{N}_0$. Then the following equation holds with $k = \#M$:

$$\#\mathcal{A}(M) = A(2^i) + \#\mathcal{A}(M \setminus [0;2^i-1]) + \binom{k-2^i}{2} \cdot 2^{i-1}$$

Proof: First, apply the Decomposition Lemma with $a = 2^i$. Using that $M \cap H_0^{2^i} = H_0^{2^i}$ because M is saturated, we can apply Proposition 3.7 to $\#\mathcal{A}(M \cap H_0^{2^i})$ which gives $A(2^i)$. Clearly $M \cap H_1^{2^i} = M \setminus [0;2^i-1]$.

This leaves $\mathcal{A}_+$ (cf. proof of Proposition 3.7): Let $t := \max M$. Choose any $u \in M \cap [2^i;t-1]$, $v \in [0;2^i-1]$, then $\{t,u,v,t \oplus u \oplus v\} \in \mathcal{A}_+$. This gives $(k-2^i-1)\cdot 2^i$, but choosing $v' = t \oplus u \oplus v$ yields the same solution, so we have to divide by 2. On the other hand, if $\{t,u,v,w\} \in \mathcal{A}_+$, let without loss of generality $v,w \in H_0^{2^i}$ and thus $u \in M \cap [2^i;t-2]$, because $u \in H_1^{2^i} \setminus \{t\}$. □

3.4.2 Saturation of arbitrary sets

If M is a saturated set, in general $(M \setminus H_0^{2^i}) \oplus 2^i$ is not again a saturated set, and of course not every set is saturated to begin with. The following lemma overcomes this problem by showing that every set can be mapped to a saturated set without decreasing the number of 2-dimensional affine subspaces it contains.

Lemma 3.11 *Let $M \subseteq \mathbb{F}_2^n$ be an arbitrary subset. Then there exists a saturated set $M' \subseteq \mathbb{F}_2^n$ with $\#M = \#M'$ and $\#\mathcal{A}(M) \leq \#\mathcal{A}(M')$.*

Proof: The claim is trivial if M is saturated, then $M' = M$ is a solution. So assume that M is not saturated, and thus $M \neq \emptyset$, $M \neq \{0\}$. Let $g := \min G(M)$ and let i be the largest integer such that $2^i \leq g$ if $g > 0$ and let $i := -1$ if $g = 0$. Then $m := \max M \geq 2^{i+1}$ because M is not saturated, and there exists an integer j with $j > i$ and $2^j \leq m < 2^{j+1}$.

We will now map m to g without affecting the elements below the minimal gap, and without decreasing the number of 2-dimensional affine subspaces. Iterating the process will lead to a sequence of sets which terminates in a saturated set with at least as many 2-dimensional affine subspaces as M. For clarity we will present the mapping that does this trick in two steps.

Shuffle: As a first step, consider the affine function $\alpha \in \mathcal{F}(\mathbb{F}_2^n, \mathbb{F}_2^n)$ which maps 2^j to $m \oplus g$ and leaves all other powers of 2 constant. Because $m \oplus g$ lies in $H_1^{2^j}$, the function α maps a basis to a basis and is thus a bijection on $\mathbb{F}_2^n$ and leaves the number of 2-dimensional affine subspaces in an arbitrary subset invariant. Furthermore, α is constant on $[0; 2^{i+1} - 1] \subset [0; 2^j - 1]$ and $2^j \leq \max \alpha(M) < 2^{j+1}$. The desired outcome of applying α on M is $\alpha(m) = g \oplus 2^j$, that means α *shuffles* the elements of $H_1^{2^j}$ around so that at least one of them is at the location of the minimal gap g modulo 2^j. We shall call the set $S := \{s \in G(\alpha(M)) \cap H_0^{2^i} \mid s \oplus 2^j \in M\}$ the set of *sinkable* positions of $\alpha(M)$.

Sink: In the second step, we will map all sinkable elements of $\alpha(M)$ to their respective gaps. Let σ be the permutation on $\mathbb{F}_2^n$ which swaps each element $s \in S$ with $s \oplus 2^j$ and does not affect any other element. It is clear that $\sigma\alpha(M) = S \uplus M \setminus (S \oplus 2^j)$ and thus $[0; g] \subseteq \sigma\alpha(M)$, so we have $\min G(\sigma\alpha(M)) > \min G(M)$. That means σ *sinks* each sinkable element of $\alpha(M)$ into its associated gap, while leaving all other elements invariant. In the process, the minimal gap is filled as well.

Monotony of $\#\mathcal{A}$: We shall see now that $\#\mathcal{A}(M) = \#\mathcal{A}(\alpha(M)) \leq \#\mathcal{A}(\sigma\alpha(M))$. For this, we have to construct an injective mapping $s_{\mathcal{A}} : \mathcal{A}(\alpha(M)) \to \mathcal{A}(\sigma\alpha(M))$. Unfortunately, this is rather tricky. We will require that $s_{\mathcal{A}}$ leaves the values of A invariant modulo 2^j. The we only need to show that $s_{\mathcal{A}}$ is injective on each subset $\mathcal{A}_{t,u,v,w} = \{\{t', u', v', w'\} \in \mathcal{A}(\alpha(M)) \mid t \equiv t', u \equiv u', v \equiv v', w \equiv w' \pmod{2^j}\}$ with $t, u, v, w \in [0; 2^j - 1]$, not necessarily pairwise different. To differentiate between the possible cases, we characterize a position $t < 2^j$ by its *signature* $\operatorname{sig} : [0; 2^j - 1] \to \{\square, \blacksquare, \square, \blacksquare\}$ with:

$$\operatorname{sig} t := \begin{cases} \square & : \quad t \notin \alpha(M), t \oplus 2^j \notin \alpha(M) \\ \blacksquare & : \quad t \in \alpha(M), t \oplus 2^j \notin \alpha(M) \\ \square & : \quad t \notin \alpha(M), t \oplus 2^j \in \alpha(M) \\ \blacksquare & : \quad t \in \alpha(M), t \oplus 2^j \in \alpha(M) \end{cases}$$

In other words (and for all positions mod 2^j), the symbol ▯ denotes a position with a gap above a gap, ▮ an element in the saturated part of $\alpha(M)$ that does not have a corresponding candidate for sinking, ▮ a position at which a sinkable element resides and ▮ a position with a candidate for sinking that is blocked by a corresponding element in the saturated part.

We will now define $s_{\mathcal{A}}$ on the sets $\mathcal{A}_{t,u,v,w}$ based on the multi-set $\operatorname{sig}\{t,u,v,w\}$.

- If $S \cap \{t,u,v,w\}$ has even cardinality (i. e. 0, 2 or 4) then let $s_{\mathcal{A}} \mid \mathcal{A}_{t,u,v,w}$ be the concatenation of all transpositions $(x, x \oplus 2^j)$ where $x \in S \cap \{t,u,v,w\}$.

- Otherwise, if ▮ $\in \operatorname{sig}\{t,u,v,w\}$ then take $y := \max\{x \in \{t,u,v,w\} \mid \operatorname{sig}(x) = $ ▮$\}$ and let $s_{\mathcal{A}} \mid \mathcal{A}_{t,u,v,w}$ be the concatenation of the transposition $(y, y \oplus 2^j)$ with all transpositions $(x, x \oplus 2^j)$ where $x \in S \cap \{t,u,v,w\}$.

- In all other cases the set $\mathcal{A}_{t,u,v,w}$ is actually empty, because a 2-dimensional affine subspace in it would necessarily have an odd number of points in $H_1^{2^j}$, a contradiction.

To illustrate these rules, assume that ▮ $\in \operatorname{sig}\{t,u,v,w\}$ (otherwise $s_A \mid \mathcal{A}_{t,u,v,w}$ is simply the identity). In that case $t \oplus 2^j \notin A$ for every $A \in \mathcal{A}_{t,u,v,w}$ and $\{t,u,v,w\}$ are pairwise different. Up to renaming the variables this case splits into 10 distinct sub-cases (see Figure 2.1, p. 11) which are listed in Table 3.1.

Table 3.1: The mapping $s_{\mathcal{A}}$ from the proof of Lemma 3.11.

$\operatorname{sig}(t,u,v,w)$	$s_{\mathcal{A}} \mid \mathcal{A}_{t,u,v,w}$
(▯, ▯, ▯, ▯), (▮, ▯, ▯, ▯)	n/a, the set $\mathcal{A}_{t,u,v,w}$ is empty
(▯, ▯, ▯, ▮), (▮, ▯, ▯, ▮)	$(t, t \oplus 2^j)(u, u \oplus 2^j)(v, v \oplus 2^j)(w, w \oplus 2^j)$
(▯, ▯, ▮, ▯), (▮, ▮, ▯, ▯)	$(t, t \oplus 2^j)(u, u \oplus 2^j)$
(▯, ▯, ▮, ▮), (▮, ▮, ▮, ▯)	$(t, t \oplus 2^j)(u, u \oplus 2^j)$
(▮, ▮, ▮, ▯)	$(t, t \oplus 2^j)(x, x \oplus 2^j)$ where $x = \max\{u, v\}$
(▮, ▮, ▮, ▮)	$(t, t \oplus 2^j)(x, x \oplus 2^j)$ where $x = \max\{u, v, w\}$

All these restrictions of $s_{\mathcal{A}}$ are permutations generated by an even number of transpositions $(x, x \oplus 2^j)$, $x \in \{t,u,v,w\}$ with $\operatorname{sig} x \in \{$▯, ▮$\}$. Because they are permutations, they are injective. Because a transpositions swaps two elements $x \equiv x \oplus 2^j \pmod{2^j}$, these restrictions leave any subspace affine invariant modulo 2^j. Thus injectivity of $s_{\mathcal{A}}$ on the sets $\mathcal{A}_{t,u,v,w}$ is sufficient for $s_{\mathcal{A}}$ to be injective. Furthermore, there is an even number of transpositions and thus the 2-dimensional affine subspaces are mapped to 2-dimensional affine subspaces in $\mathcal{A}(\mathbb{F}_2^n)$. Finally, because $\operatorname{sig} x \in \{$▯, ▮$\}$ for all transpositions $(x, x \oplus 2^j)$, these images are also in $\mathcal{A}(\sigma\alpha(M))$. In summary we have that $s_{\mathcal{A}} : \mathcal{A}(\alpha(M)) \to \mathcal{A}(\sigma\alpha(M))$ is well-defined and injective on its whole domain.

Iterate: The algorithm above constructs a new set $\sigma\alpha(M)$ with the same number of elements as M, and with at least the same number of 2-dimensional affine subspaces. In addition, the minimal gap of $\sigma\alpha(M)$ is strictly larger than that of M, while the

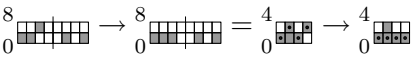
maximum $\max \sigma\alpha(M)$ is bound by 2^{j+1}. By iterating this process as often as possible we get a sequence:

$$M =: M_0 \overset{\sigma_1\alpha_1}{\to} M_1 \overset{\sigma_2\alpha_2}{\to} \cdots \overset{\sigma_r\alpha_r}{\to} M_r$$

This sequence has a strictly monotically increasing minimal gap g_l, while $j_l := \lfloor \log_2 \max M_l \rfloor$ is monotonically decreasing. Eventually, the sequence terminates when $g_r \geq 2^{j_r}$ and thus M_r is saturated.

This sequence has the additional property that $\#\mathcal{A}(M_l) \leq \#\mathcal{A}(M_{l+1})$ for all $l \in [0; r-1]$ from which the original claim finally follows. □

Example 3.12 *The subset $\{0, 1, 5, 7, 10\} \subset \mathbb{F}_2^n$ with $n = 4$ does not contain any 2-dimensional affine subspace, while the saturated set $\{0, 1, 2, 3, 6\}$ has exactly one. The following diagram shows the iterations in the sinking step of the proof.*



3.4.3 Upper bound on $\mathcal{A}(M)$, M arbitrary

Finally we can derive the main result of this section: The exact upper bound for the number of 2-dimensional affine subspaces in arbitrary subsets of $\mathbb{F}_2^n$.

Theorem 3.13 (Number of 2-dimensional affine subspaces)
Let $M \subseteq \mathbb{F}_2^n$ be an arbitrary subset. Then:

$$\#\mathcal{A}(M) \leq A(\#M)$$

Proof: Proof by induction over $k := \#M$. It is clear that $\#\mathcal{A}(\emptyset) = 0 = A(0)$ and $\#\mathcal{A}(\{0\}) = 0 = A(1)$. Let $k \geq 2$ and the claim be true for sets with fewer elements than k. By Lemma 3.11 find a saturated set M' such that $\#\mathcal{A}(M) \leq \#\mathcal{A}(M')$. We apply Lemma 3.10 and get:

$$\#\mathcal{A}(M) \leq \#\mathcal{A}(M') = A(2^i) + \#\mathcal{A}(M' \setminus [0; 2^i - 1]) + \binom{k - 2^i}{2} \cdot 2^{i-1}$$

But $M' \setminus [0; 2^i - 1]$ is a set with $k - 2^i + 1 < k$ elements, so the induction assumption applies, and we get:

$$\#\mathcal{A}(M) \leq A(2^i) + A(k - 2^i) + \binom{k - 2^i}{2} \cdot 2^{i-1} = A(k)$$

□

Note that this proof contains a way to construct the injective mapping from $\mathcal{A}(M)$ to $\mathcal{A}([0; k-1])$ by applying the shuffle-sink algorithm from the proof of Lemma 3.11 to the sets $(M' \cap H_1^{2^j}) \oplus 2^j$ recursively, and then mapping the result back to $H_1^{2^j}$ by adding 2^j, expanding the functions outside of their domain by the identity. We do not give the pseudocode for this algorithm, as it is not further needed here.

Also note that saturated sets are only a tool introduced to split the burden of the proof at the point where two nested iterations interact. They are not used further in this thesis.

Chapter 4

Classification of APN functions

In this chapter, we will develop the classification of APN functions according to the affine and CCZ equivalence relations.

4.1 APN functions

Definition 4.1 (Almost perfect nonlinear)
A function $s \in \mathcal{F}(\mathbb{F}_2^n, \mathbb{F}_2^n)$ is almost perfect nonlinear *(APN) if the equation*

$$s(x) \oplus s(x \oplus c) = a \tag{4.1}$$

for $x \in \mathbb{F}_2^n$ has 0 or 2 solutions for all $a, c \in \mathbb{F}_2^n$ and $c \neq 0$.

We define the result predicate $\rho_{\text{APN}} : \mathcal{F}(\mathbb{F}_2^n, \mathbb{F}_2^n) \to \{\text{TRUE}, \text{FALSE}\}$ by $\rho_{\text{APN}}(s) = \text{TRUE} \iff s$ is APN, to be used as the result function in backtrack problems related to finding APN functions.

Note that if x is a solution of (4.1), then $x \oplus c$ is another solution because $s(x \oplus c) \oplus s(x \oplus c \oplus c) = s(x) \oplus s(x \oplus c)$ and $x \neq x \oplus c$ due to $c \neq 0$. Thus, the number of different solutions is always even and the requirement in the definition is optimal.

Consider an affine function $\alpha \in \mathcal{F}(\mathbb{F}_2^n, \mathbb{F}_2^n)$. Then the equation

$$\alpha(x \oplus c) = \alpha(x) \oplus \alpha(c) \ominus \alpha(0) \iff \alpha(x) \oplus \alpha(x \oplus c) = \alpha(0) \oplus \alpha(c) \tag{4.2}$$

holds for all $x, c \in \mathbb{F}_2^n$. This shows that APN functions can not be approximated well by affine functions and explains the origin of their name.

Example 4.2 *Let $n = 1$ and $s \in \mathcal{F}(\mathbb{F}_2^n, \mathbb{F}_2^n)$. Then s is trivially APN, because $\mathbb{F}_2^n$ has only 2 members. There are $(2^1)^{2^1} = 4$ such functions.*

4.1.1 APN functions and affine subspaces

The following theorem shows how the APN property is related to the set of 2-dimensional affine subspaces in $\mathbb{F}_2^n$. The first part of the theorem is central to the development and analysis of a filter function suitable to be used in APN-related backtrack problems.

Theorem 4.3 *Let $s \in \mathcal{F}(\mathbb{F}_2^n, \mathbb{F}_2^n)$. Then the following conditions are equivalent:*

1. *The function s is APN.*
2. *For all pairwise different $t, u, v, w \in \mathcal{F}(\mathbb{F}_2^n, \mathbb{F}_2^n)$ it holds that:*

$$t \oplus u \oplus v \oplus w = 0 \Rightarrow s(t) \oplus s(u) \oplus s(v) \oplus s(w) \neq 0 \tag{4.3}$$

   (If s is a permutation, this is equivalent to the condition that no image of a 2-dimensional affine subspace of $\mathbb{F}_2^n$ is a 2-dimensional affine subspace.)

3. *For the image of any 2-dimensional affine subspace $A = \{t, u, v, w\} \in \mathcal{A}(\mathbb{F}_2^n)$ under s the following is true:*
   - *$s(A)$ is not a 0-dimensional affine subspace of $\mathbb{F}_2^n$, and*
   - *$s(A)$ is not a 1-dimensional affine subspace of $\mathbb{F}_2^n$ where $\#(s^{-1}(s(a)) \cap A) = 2$ for any $a \in A$, and*
   - *$s(A)$ is not a 2-dimensional affine subspace of $\mathbb{F}_2^n$.*

   (This is equivalent to saying that the multi-set $\{s(t), s(u), s(v), s(w)\}$ is not an affine subspace with even multiplicities.)

Proof: First we show that 1 is equivalent to 2.

"1 $\Rightarrow$ 2": Assume that $s \in \mathcal{F}(\mathbb{F}_2^n, \mathbb{F}_2^n)$ is APN and $t, u, v, w \in \mathbb{F}_2^n$ pairwise different such that $t \oplus u \oplus v \oplus w = 0$. Let $c := t \oplus u \neq 0$, then we have $s(t) \oplus s(t \oplus c) \neq s(v) \oplus s(v \oplus c)$ because s is APN and $v \neq t$, $v \neq u = t \oplus c$. Thus, $s(t) \oplus s(t \oplus c) \oplus s(v) \oplus s(v \oplus c) \neq 0$ but $t + c = u$ and $v + c = v + t + u = w$.

"2 $\Rightarrow$ 1": Assume that we have $x, y, c, a \in \mathbb{F}_2^n$, $c \neq 0$, such that $s(x) \oplus s(x \oplus c) = a$ and $s(y) \oplus s(y \oplus c) = a$. Then we have $s(x) \oplus s(y) \oplus s(x \oplus c) \oplus s(y \oplus c) = 0$ and thus $x, y, x \oplus c, y \oplus c$ can not be pairwise different. But $c \neq 0$ so we find either $y = x$ or $y = x \oplus c$.

Now it remains to be shown that 2 is equivalent to 3.

"2 $\Rightarrow$ 3": Assume that $s \in \mathcal{F}(\mathbb{F}_2^n, \mathbb{F}_2^n)$ is APN, and $A \in \mathcal{A}(\mathbb{F}_2^n)$ is a two-dimensional affine subspace. Then $s(t) \oplus s(u) \oplus s(v) \oplus s(w) \neq 0$ and clearly $s(A)$ is not a single point, and thus not a 0-dimensional affine subspace. Also, $s(A)$ is not a 2-dimensional affine subspace because of Proposition 3.1. Let $s(A) = \{x, y\}$, $x \neq y$, be a 1-dimensional affine subspace. Then without loss of generality $s(t) = s(u) = x$ and $s(v) = y$. It follows that $x \oplus x \oplus y \oplus s(w) \neq 0$, so $s(w) \neq y$ which leaves $s(w) = x$. This shows that $\#(s^{-1}(s(a)) \cap A) \in \{1, 3\}$ for all $a \in A$.

"3 $\Rightarrow$ 2": Assume that we have $A = \{t, u, v, w\} \in \mathcal{A}(\mathbb{F}_2^n)$ pairwise different with $t \oplus u \oplus v \oplus w = 0$ and s satisfies the conditions under 3. Consider the multi-set {s(t), s(u), s(v), s(w)}, which we can assume without loss of generality to be ordered by increasing multiplicity. The cases are:

- Multiplicities $(1, 1, 1, 1)$: We have that $s(t), s(u), s(v), s(w)$ are all pairwise different, but do not form a 2-dimensional affine subspace and thus $s(t) \oplus s(u) \oplus s(v) \oplus s(w) \neq 0$.

- Multiplicities $(1,1,2)$: We have that $s(t) \neq s(u)$ and $s(v) = s(w)$ and it follows that $s(t) \oplus s(u) \oplus s(v) \oplus s(w) = s(t) \oplus s(u) \neq 0$ trivially.

- Multiplicities $(1,3)$: We have that $s(t) \neq s(u) = s(v) = s(w)$ and it follows that $s(t) \oplus s(u) \oplus s(v) \oplus s(w) = s(t) \oplus s(u) \neq 0$ trivially.

- Multiplicities $(2,2)$: Then the image would be a 1-dimensional affine subspace with $\#(s^{-1}(s(a)) \cap \{t,u,v,w\}) = 2$ for all $a \in A$, a contradiction.

- Multiplicities (4): Then the image would be a 0-dimensional affine subspace, a contradiction.

□

Example 4.4 *Let $n = 2$ and $s \in \mathcal{F}(\mathbb{F}_2^n, \mathbb{F}_2^n)$. Then s is APN if and only if $s(0) \oplus s(1) \oplus s(2) \oplus s(3) \neq 0$. This means that of the $(2^n)^{2^n} = 4^4 = 256$ functions there are $4 \cdot 4 \cdot 4 \cdot 3 = 192$ APN functions (choose $s(0)$, $s(1)$, and $s(2)$ out of 2^2, then $s(3) \neq s(0) \oplus s(1) \oplus s(2)$ out of $2^2 - 1$). There are $4 \cdot 4 \cdot 4 \cdot 1 = 64$ functions which are not APN, including all permutations (because $0 \oplus 1 \oplus 2 \oplus 3 = 0$).*

4.1.2 APN filter predicate

We will now develop a filter predicate for the constraint that s is APN. We can fail a left-refined template $\tilde{s}$ and backtrack if we can determine that $\diamond_\bullet \tilde{s}$ does not contain any function that is APN. Condition 2 in Theorem 4.3 gives us a suitable local property of APN functions to check. What is still missing is an efficient method to check this condition for all 2-dimensional affine subspaces among the determinate positions of $\tilde{s}$.

Consider the maximum $\max A$ of a 2-dimensional affine subspace A. If $\tilde{s}$ is a left-refined template we can check Condition 2 in Theorem 4.3 only for subspaces $A \subseteq D_{\tilde{s}}$ among the determinate positions of $\tilde{s}$, that means for those subspaces with $\max A < \deg \tilde{s}$. This motivates the following partition of $\mathcal{A}(\mathbb{F}_2^n)$:

$$\mathcal{A}_i := \{A \in \mathcal{A}(\mathbb{F}_2^n) | \max A = i - 1\}, \text{ for } 0 \leq i \leq 2^n \tag{4.4}$$

$$\mathcal{A}(\mathbb{F}_2^n) = \biguplus_{i=0}^{2^n} \mathcal{A}_i \tag{4.5}$$

Lemma 4.5 *Let ϕ_{APN} be a stateful filter such that the induced filter $\hat{\phi}_{\text{APN}}(\tilde{s})$ is* TRUE *if and only if $\tilde{s}$ satisfies Condition 2 in Theorem 4.3 on all subspaces in $A_{\deg \tilde{s}}$. Then $\hat{\phi}_{\text{APN}}$ satisfies (2.7), p. 10, that means:*

$$\rho_{\text{APN}} \equiv \hat{\phi}_{\text{APN}}^S \mid \mathcal{F}(\mathbb{F}_2^n, \mathbb{F}_2^n) \tag{4.6}$$

Proof: For $s \in \mathcal{F}(\mathbb{F}_2^n, \mathbb{F}_2^n)$ with $s = \diamond^{\ell}_{2^n-1\mapsto s_{2^n-1}} \cdots \diamond^{\ell}_{0\mapsto s_0} \tilde{\diamond}$ we have:

$$\begin{aligned}
\hat{\phi}^S_{\mathrm{APN}}(s) &= \bigwedge_{i=0}^{2^n} \hat{\phi}_{\mathrm{APN}}(\diamond^{\ell}_{i-1\mapsto s_{i-1}} \cdots \diamond^{\ell}_{0\mapsto s_0} \tilde{\diamond}) \\
&= \bigwedge_{i=0}^{2^n} (s \text{ satisfies Condition 2 in Theorem 4.3 for } A \in \mathcal{A}_i) \\
&= (s \text{ satisfies Condition 2 in Theorem 4.3 for } A \in \mathcal{A}(\mathbb{F}_2^n)) \\
&= \rho_{\mathrm{APN}}(s)
\end{aligned}$$

□

We now turn to the implementation of the filter predicate ϕ_{APN}. One possibility would be to iterate through the subspaces $\{t, u, v, w\} \in \mathcal{A}_{t+1}$, $w < v < u < t = \deg \tilde{s} - 1$, and check that $\tilde{s}(t) \oplus \tilde{s}(u) \oplus \tilde{s}(v) \oplus \tilde{s}(w) \neq 0$ for each of them. The subspace sets $\mathcal{A}_t$ can be pregenerated; to our knowledge there is no efficient[1] way to generate them anyway. However, the total time and space complexity of this algorithm (for a single walk from the root to a leaf in the search tree) is $O(A(2^n))$, which is cubic in 2^n as was shown in (3.5), p. 22.

A much better method is to determine, after each one-step refinement, the new equations $\tilde{s}(t) \oplus \tilde{s}(t \oplus c) = a$ for which solutions have now been found, where c is chosen such that $t \oplus c < t = \deg \tilde{s} - 1$. These equations can be stored in a table indexed by $(c, a) \in \mathbb{F}_2^n \times \mathbb{F}_2^n$ with values in $\{\textsc{true}, \diamond\}$. A boolean value is sufficient, because we do not need to remember what the solution is, only that one exists. When a table entry already contains TRUE a conflict arises and a backtrack occurs. Otherwise, if $\tilde{s}$ is determinate and no such conflict occured, the function is APN. The following theorem asserts the correctness of Algorithm 4.

Theorem 4.6 *Algorithm 4 implements a filter ϕ_{APN} for the backtrack problem $P_{\mathrm{APN}} := (\mathbb{F}_2^n, \mathbb{F}_2^n, \rho_{\mathrm{APN}}, \phi_{\mathrm{APN}}, \Sigma, S_{\tilde{\diamond}})$.*

Proof: Let $\tilde{s}_i = \diamond^{\ell}_{i-1\mapsto s_{i-1}} \cdots \diamond^{\ell}_{0\mapsto s_0}$ be the templates compatible with any function $s \in \mathcal{F}(\mathbb{F}_2^n, \mathbb{F}_2^n)$ and let T be the active search tree (see Definition 2.12, p. 10). Because of Lemma 4.5 it suffices to show that for all $\tilde{s}_k$ with $\tilde{s}_{k-1} \in T$ it holds that:

$$\begin{aligned}
&\phi_{\mathrm{APN}}(s_{k-1}, S_{\tilde{s}_{k-1}}) = \textsc{false} \\
\iff &\tilde{s}_k(t) \oplus \tilde{s}_k(u) \oplus \tilde{s}_k(v) \oplus \tilde{s}_k(w) = 0 \text{ for some } \{t, u, v, w\} \in \mathcal{A}_k
\end{aligned}$$

Note that Algorithm 4 returns FALSE if and only if a conflict arises when writing the table entries.

"⇒": Assume that a conflict arises. This means that at the refinement of $\tilde{s}_t$ the equation $\tilde{s}_{t+1}(t) \oplus \tilde{s}_{t+1}(t \oplus c) = a$ with $t \oplus c < t$ was added to the table without creating a conflict. Then at the refinement of $\tilde{s}_{k-1}$ with $k - 1 =: t' > t > t \oplus c$ another equation $\tilde{s}_k(t') \oplus \tilde{s}_k(t' \oplus c) = a$ is found with $t' \oplus c < t'$. Note that necessarily $t' > t$ because all

[1] Efficient here means that there are constant time algorithms to determine the initial element and the successor function to any natural order of the set A_i.

Algorithm 4 The stateful filter ϕ_{APN} for APN functions with $\Sigma := (\mathcal{F}(\mathbb{F}_2^n \setminus \{0\} \times \mathbb{F}_2^n, \{\diamond, \textsc{true}\}) \times \mathcal{F}(\mathbb{F}_2^n, \tilde{\mathbb{F}}_2^n)) \cup \{\textsc{false}\}$, $S_{\tilde{\diamond}} := (\tilde{\diamond}, \tilde{\diamond})$.

function $\phi_{\mathrm{APN}}(b,\ (\tilde{f}, \tilde{s}))$
  var $k \in \mathbb{N}_0$
  var $c \in \mathbb{F}_2^n \setminus \{0\}$
  var $a \in \mathbb{F}_2^n$

  $k \leftarrow \deg \tilde{s}$ ▷ Depth of recursion
  $\tilde{s} \leftarrow \diamond^{\ell}_{\deg \tilde{s} \mapsto b} \tilde{s}$

  for all $x \in [0; k-1]$ **do**
    $c \leftarrow k \oplus x$
    $a \leftarrow \tilde{s}(x) \oplus \tilde{s}(k)$ ▷ $k = x \oplus c$

    if $\tilde{f}(c, a) = \textsc{true}$ **then**
      return FALSE ▷ Table conflict
    else
      $\tilde{f} \leftarrow \diamond_{(c,a) \mapsto \mathrm{TRUE}} \tilde{f}$ ▷ Table update
    end if
  end for

  return $(\tilde{f}, \tilde{s})$
end function

Figure 4.1: The configuration of the APN filter algorithm just after detecting a conflict for $\tilde{s} = \diamond^{\ell}_{3\mapsto 3}\diamond^{\ell}_{2\mapsto 2}\diamond^{\ell}_{1\mapsto 1}\diamond^{\ell}_{0\mapsto 0}\tilde{\diamond}$. Table entries with TRUE are marked by the depth k at which they were written and by the equation that produced them. Rows and columns with all FALSE are omitted.

	c=1	c=2	c=3
a=1	2: $s(0) \oplus s(0 \oplus 1) = 1$		
a=2		3: $s(0) \oplus s(0 \oplus 2) = 2$	
a=3			3: $s(1) \oplus s(1 \oplus 3) = 3$ 4: $s(0) \oplus s(0 \oplus 3) = 3$

solutions entered in a single invocation of the filter are in different table entries due to different values of c. From $t' > t > t \oplus c$ it follows that $t', t' \oplus c, t$ and $t \oplus c$ are pairwise different and the APN property is violated for $\{t', t' \oplus c, t, t \oplus c\} \in A_{k+1}$.

"$\Leftarrow$": Proof is by induction over k. Clearly the claim holds for any one-step refinement of $\tilde{\diamond}$, this settles the case $k = 1$. Assume now that the claim is true for all $k' < k$ and that $\tilde{s}(t) \oplus \tilde{s}(u) \oplus \tilde{s}(v) \oplus \tilde{s}(w) = 0$ for some $\{t, u, v, w\} \in \mathcal{A}_k$. Without loss of generality assume that $t < u < v < w = k - 1$. Let $c := t \oplus u$, then $u \oplus c < u < w$, $w \oplus c < w$ and $s(u) \oplus s(u \oplus c) = s(w) \oplus s(w \oplus c) =: a$.

Because $\tilde{s}_{k-1} \in T$ we know by the induction hypothesis that no conflict occured for any ancestor $\tilde{s}_i$ with $i < k$. Thus we can reconstruct fully what happens to the table entry (c, a): For all states $S_{\tilde{s}_i} = (\tilde{f}_i, \tilde{s}_i)$ we have $\tilde{f}_i(c, a) = \diamond$ for $i \leq u$ and $\tilde{f}_i(c, a) =$ TRUE for $u < i \leq w$ because the table entry $f_i(c, a)$ is written when the loop in Algorithm 4 is executed for $\phi_{\mathrm{APN}}(s_u, S_{\tilde{s}_u})$ with $x = t$. Finally, if the loop is executed for $\phi_{\mathrm{APN}}(s_{k-1}, S_{\tilde{s}_{k-1}})$ with $x = v$ a conflict will occur for the table entry (c, a). The only reason for the loop not to execute with $x = v$ is that a conflict occurs for $x < v$. In either case the return value is FALSE. □

Figure 4.1 shows an example for the internal state of Algorithm 4 at the critical step right after a conflict occured.

4.1.3 Evaluation of ϕ_{APN}

The number of memory bits needed to store the table entries is $(2^n - 1) \cdot 2^n$ and thus only quadratic[2] in 2^n. At most half of the table entries will be populated at any time during the execution of the algorithm. To check for a conflict, only $\deg \tilde{s} - 1$, $\tilde{s} \neq \tilde{\diamond}$, table entries have to be checked at each step, requiring this very number of fetch and store operations. Hence time complexity (for a single walk from the root to a leaf in the search tree) is quadratic in 2^n as well.[3] The downside is that some calculations which are only needed to fail a node at a deeper depth are done early. However, the results of these calculations are shared by all descendant nodes and thus the costs are more than

[2] For $n \leq 8$ such a table can fit completely into the level 1 data cache of a modern desktop CPU.

[3] Note that because in this case every table entry is written at most once, keeping a stack of modified table entries and undoing the modifications at rollback time (see Section A.2) increases processing space and time only by a small linear factor.

amortized.

The success in this construction of the filter is rooted in the way the data to calculate potential conflicts is stored: An unused table entry (i. e. a single memory fetch operation) gives a lot of information about which conflicts do *not* occur at that step. This is one way in which a stateful filter function may vastly outperform a stateless one.

Example 4.7 *Algorithms 2 and 4 can be combined to a program that lists all APN functions for $n = 3$ in a few of seconds. There are 668128 in total, among them 10752 permutations.*

If used in addition to the permutation filter ϕ_σ from Section 2.3.3, one can verify in a couple of hours that there are no bijective APN functions for $n = 4$. However, listing the existing, non-bijective APN functions for $n = 4$ takes too long. We will see in Section 4.2 why that is the case and also develop techniques to deal with the cases $n = 4$ and $n = 5$.

We will now show that the requirements in Note 2.13, p. 12, are fulfilled, and then give quantitative results about the constraints imposed by the APN property.

Proposition 4.8 *A left-refined template $\tilde{s}$ creates the maximum number of opportunities for conflicts $C_{\tilde{s}}$ compared to all other possible templates $\tilde{t}$ with the same degree.*

Proof: An opportunity for a conflict is a 2-dimensional affine subspace in the set $D_{\tilde{t}}$ of determinate positions of $\tilde{t}$. The number of elements in $D_{\tilde{t}}$ is $\#D_{\tilde{t}} = \deg \tilde{t}$. Thus, the number of opportunities for a conflict created by $\tilde{t}$ is $\#\mathcal{A}(D_{\tilde{t}})$ and according to Theorem 3.13 we have:

$$C_{\tilde{t}} = \#\mathcal{A}(D_{\tilde{t}}) \leq A(\deg \tilde{t}) = A(\deg \tilde{s}) = \#\mathcal{A}(D_{\tilde{s}}) = C_{\tilde{s}}$$

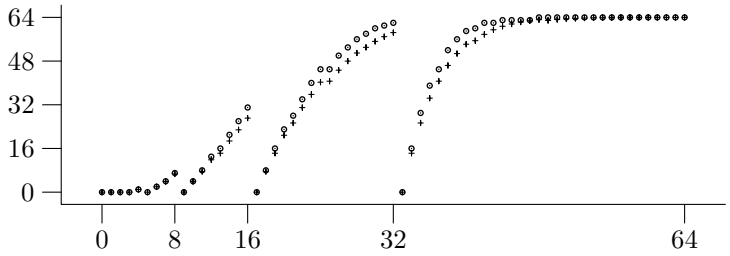
The proof uses that $D_{\tilde{s}} = [0; \deg \tilde{s} - 1]$. □

Note 4.9 *The sequence $\Delta(k)$ gives the number of new opportunities for conflicts created at depth k of the recursion and thus is an indication for how well the filter satisfies Condition 1 in Note 2.13, p. 12.*

It is clear that this is only a heuristic approach to the problem in which order the APN function template should be refined, because we are only looking at the number of opportunities for new conflicts, not at the number of actual conflicts. Indeed, if the approximation (3.6) is correct, then $\Delta(k) \approx \frac{1}{6}k^2 \gg 2^n$ for sufficiently large k, and the possible conflicts vastly overdetermine the possible values at most positions. The opposite scenario can also occur. If, for example, we have a ridiculously high dimension like $n = 2^k$ and $\tilde{s}(i) = 2^i$ for $0 \leq i < 2^k - 1$, then all $\Delta(2^k)$ possible conflicts are proper because there are no linearly dependent vectors among $\tilde{s}([0; 2^k - 1])$.

Example 4.10 *The 2-dimensional affine subspaces $\{0, 3, 9, 10\}$ and $\{4, 6, 8, 10\}$ lead to the two constraints $s(10) \neq s(0) \oplus s(3) \oplus s(9)$ and $s(10) \neq s(4) \oplus s(6) \oplus s(8)$, but there is no constraint that forbids $s(0) \oplus s(3) \oplus s(9) = s(4) \oplus s(6) \oplus s(8)$. In fact, one can easily find APN functions with this property, for example for $n = 4$:*

Figure 4.2: Actual number of conflicts for $n = 6$ over depth k. The estimated model values $C(k)$ are given by crosses and the measured estimates by dots.



x	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$s(x)$	0	0	0	1	0	2	4	7	0	5	6	8	3	14	11	13

To determine the actual number of conflicts occuring at each one-step refinement, we bravely assume that all possible conflicts $\Delta(k)$ are independent probability events and are not influenced by the determinate values at positions smaller than k. Then the following proposition holds:

Proposition 4.11 *The estimated number of actual conflicts $C(k)$ at the k-th one-step refinement is*

$$C(k) = 2^n \cdot \left(1 - \left(\frac{2^n - 1}{2^n}\right)^{\Delta(k)}\right) \tag{4.7}$$

Proof: The probability that an individual conflict does not occur is $(2^n - 1)/2^n$, and all conflicts are assumed to be independent, thus the probability for no conflict at the k-th one-step refinement is $(\frac{2^n-1}{2^n})^{\Delta(k)}$. There are 2^n such one-step refinements. □

Corollary 4.12 *The expected number of nodes at level k in the active search tree T is:*

$$\prod_{i=0}^{2^n} (2^n - C(k)) \tag{4.8}$$

This estimate is astonishingly accurate, in particular for large n and small k, for example $k < 2^{n-1}$. The estimate for the total number of nodes breaks down for $k \approx 2^n$, because the errors are propagating multiplicatively.

Note 4.13 *We can now appreciate the difficulties in constructing APN functions. The number of potential new conflicts $\Delta(k)$ grows with the square of k, and thus the number of actual conflicts $C(k)$ approaches 2^n rapidly, see Figure 4.2. There are "breathing holes" at every power of 2 (and small "dips" at short sums of powers of 2) where the number of conflicts goes down, but overall an APN function is faced with about $2^n - \alpha \log_2 2^n = 2^n - \alpha n$ positions for which the expected number of non-conflicting values is close to 0, where $\alpha \in \mathbb{R}_{>0}$ is a small constant.*

One should note that even if the actual number of conflicts at any particular step were not maximal for a left-refinement, the left-refinement might still be the appropriate choice in some situations. After all, the refinement chosen must also support other interesting filter predicates that we want to apply at the same time, for example affine equivalence as in Section 4.2.

4.2 Affine equivalence

Even though the estimated number of nodes in the active search tree T given by (4.8) is not very reliable, it indicates that trying to list all APN functions for $n > 4$ is a hopeless task. Not only is the search space enormous, but the number of APN functions itself is rather large. Therefore, it is necessary to partition the vectorial boolean functions into equivalence classes according to some equivalence relation which stabilizes the APN property. We can then apply the methods described in Section 2.4. Suitable equivalence relations are well known [NK93]:

Definition 4.14 (Affine and extended affine equivalence)
Two functions $s, t \in \mathcal{F}(\mathbb{F}_2^n, \mathbb{F}_2^n)$ are said to be affine equivalent*, written $s \simeq_{\mathrm{A}} t$, if there exist affine permutations $\alpha, \beta \in \mathcal{F}(\mathbb{F}_2^n, \mathbb{F}_2^n)$, such that:*

$$t = \beta s \alpha \tag{4.9}$$

Two functions $s, t \in \mathcal{F}(\mathbb{F}_2^n, \mathbb{F}_2^n)$ are said to be extended affine (EA) equivalent*, written $s \simeq_{\mathrm{EA}} t$, if there exist affine permutations $\alpha, \beta \in \mathcal{F}(\mathbb{F}_2^n, \mathbb{F}_2^n)$ and an affine function $\gamma \in \mathcal{F}(\mathbb{F}_2^n, \mathbb{F}_2^n)$, such that:*

$$t = \beta s \alpha \oplus \gamma \tag{4.10}$$

Proposition 4.15 *For $s, t \in \mathcal{F}(\mathbb{F}_2^n, \mathbb{F}_2^n)$ the following propositions hold:*

1. *If $s \simeq_{\mathrm{A}} t$ then s is APN (and bijective) if and only if t is APN (and bijective).*
2. *If $s \simeq_{\mathrm{EA}} t$ then s is APN if and only if t is APN.*
3. *If s is bijective then s is APN if and only if s^{-1} is APN.*

Affine and EA equivalence are special cases of CCZ equivalence (see Section 4.3). However, from an algorithmic point of view it is preferable to consider them separately. The known tests for affine equivalence are more efficient than those for the more general CCZ equivalence. For the description of these tests we need to extend the concept of partial functions to affine functions.

4.2.1 Affine functions and refinements

Let $\alpha \in \mathcal{F}(\mathbb{F}_2^n, \mathbb{F}_2^n)$ be an affine function. Then α is uniquely determined by its values on any affine basis on $\mathbb{F}_2^n$. Let $b_0, \ldots, b_n$ be such an affine basis. Then Algorithm 5 can be used to reconstruct the affine function α by a sequence of *affine refinements* $\diamond^{\mathcal{A}}_{b_n \mapsto \alpha(b_n)} \cdots \diamond^{\mathcal{A}}_{b_0 \mapsto \alpha(b_0)} \tilde{\diamond}$, where each refinement expands a template compatible with α

to an enclosing affine subspace of a higher dimension. The algorithm makes use of the relation:

$$\alpha(t \oplus u \oplus v) = \alpha(t \oplus u) \oplus \alpha(v) \ominus \alpha(0) = \alpha(t) \oplus \alpha(u) \oplus \alpha(v)$$

Algorithm 5 Refinement on affine subspaces.

function $\diamond^{\mathcal{A}}_{k \mapsto a_k}(\tilde{\alpha})$
  var $\tilde{\alpha}' \in \mathcal{F}(\mathbb{F}_2^n, \tilde{\mathbb{F}}_2^n)$
  var $D \in \wp(\mathbb{F}_2^n)$
  $\tilde{\alpha}' \leftarrow \diamond_{k \mapsto \alpha_k} \tilde{\alpha}$
  $D \leftarrow D_{\tilde{\alpha}}$
  if $D \neq \emptyset$ **then**
    var $\alpha_0 \in \mathbb{F}_2^n$
    $\alpha_0 \leftarrow \min D$ ▷ Pick a displacement vector
    for all $i \in D \setminus \{\alpha_0\}$ **do**
      $\tilde{\alpha}' \leftarrow \diamond_{\alpha_0 \oplus i \oplus k \mapsto \tilde{\alpha}(\alpha_0) \oplus \tilde{\alpha}(i) \oplus \tilde{\alpha}(\alpha_k)} \tilde{\alpha}'$
    end for
  end if
  return $\tilde{\alpha}'$
end function

An *affine function template* is the result of k affine refinements of $\tilde{\diamond}$ where $k \in \mathbb{N}_0$, $k \leq n+1$. Any affine (injective) function template can be refined to an affine (bijective) function by extending the determinate positions (and values) to an affine basis. Algorithm 6 implements this method.

Algorithm 6 Expand an affine function template $\tilde{\alpha}$ to an affine function. If the template is injective the result will be bijective.

function ExpandAffine($\tilde{\alpha}$)
  var $I \in \wp(\mathbb{F}_2^n)$
  $I \leftarrow I_{\tilde{\alpha}}$
  while $I \neq \emptyset$ **do**
    var $k \in \mathbb{F}_2^n$
    var $a_k \in \mathbb{F}_2^n$
    $k \leftarrow \min I$
    $a_k \leftarrow \min(\mathbb{F}_2^n \setminus \tilde{\alpha}(\mathbb{F}_2^n))$
    $\tilde{\alpha} \leftarrow \diamond^{\mathcal{A}}_{k \mapsto a_k} \tilde{\alpha}$
    $I \leftarrow I_{\tilde{\alpha}}$
  end while
  return $\tilde{\alpha}$
end function

4.2.2 Affine canonicity filter

We will now develop a canonicity filter $\phi_{\simeq_A}$ for affine equivalence of functions in $\mathcal{F}(\mathbb{F}_2^n, \mathbb{F}_2^n)$. An efficient algorithm to determine affine equivalence and canonicity of functions in $\mathcal{F}(\mathbb{F}_2^n, \mathbb{F}_2^n)$ is given in [BCBP03]. Algorithm 7 gives a generalisation of this algorithm which works on functions as well as indeterminate templates:

$$\phi_{\simeq_A}(\tilde{s}) = \text{FALSE} \iff \beta\tilde{s}\alpha < \tilde{s} \text{ for some affine permutations} \alpha, \beta \tag{4.11}$$

If $\phi_{\simeq_A}(\tilde{s}) = \text{FALSE}$ then clearly $R_{\simeq_A} \cap \diamond_\bullet \tilde{s} = \emptyset$ and we can fail the function template $\tilde{s}$. A perfect filter would check this inequality for each function in $\diamond_\bullet \tilde{s}$ separately. By making α and β dependent only on $\tilde{s}$ we weaken the filter considerably, thereby decreasing specificity but increasing efficiency. However, if $\tilde{s} =: s$ is determinate, there is only one compatible function and $\phi_{\simeq_A}(s) = \text{TRUE}$ if and only if $s \in R_{\simeq_A}$. This means that Algorithm 7 implements a proper canonicity filter for affine equivalence.

The filter algorithm itself uses backtracking informally, with the current recursion depth and function templates $\tilde{\alpha}, \tilde{\beta}$ for α, β as state. The algorithm is based on the two core ideas presented in [BCBP03]: The *needlework effect*, which means that the guesses for $\tilde{\alpha}$ and $\tilde{\beta}$ are immediately evaluated for their consequences on domain-value constraints, and the *exponential amplification of guesses*, which means that for $k > 0$ a k-step affine refinement has 2^{k-1} determinate positions. The algorithm ensures that refinements for $\tilde{\alpha}$ and $\tilde{\beta}$ are always affine and injective. When the algorithm terminates with FALSE, $\tilde{\alpha}$ and $\tilde{\beta}$ can be extended to affine permutations α and β that satisfy (4.11) using Algorithm 6.

Note 4.16 *Algorithm 7 can also be used to determine the canonical representative of the affine equivalence class of an arbitrary function s by calculating $s' = \beta s \alpha$ and iterating the process until a canonical representative $\beta_m \cdots \beta_0 s \alpha_0 \cdots \alpha_m$ has been found.*

4.2.3 Evaluation of the affine canonicity filter

Affine equivalence is the preferred equivalence relation for APN permutations because it preserves bijectivity. Using the filter $\phi_{\simeq_A}$ together with the permutation filter ϕ_σ results in a backtrack problem whose solution is the canonical set of representatives for the affine equivalence classes of all permutations. Their number was calculated first in [Lor64] and corrected in [dH03], and is 4 for $n = 3$, 302 for $n = 4$ but $2,569,966,041,123,938,084$ for $n = 5$, which is already too large to be constructed explicitly. Thus, the filter $\phi_{\simeq_A}$ is primarily useful in conjunction with other filters that further restrict the search space, in particular the APN filter.

The affine canonicity filter is relatively expensive, and thus it should only be run after the APN (and permutation) filter. It is also of advantage to use it in a weaker form. We modify the filter to return TRUE unconditionally if the degree of $\tilde{s}$ is greater than a certain cut-off depth and less than 2^n (so that its validity as a proper canonicity filter is preserved). The best cut-off depth was experimentally determined to be 11 for $n = 4$ and 15 for $n = 5$. Beyond that depth, excluding affine equivalent functions is slower than a brute force search using only the APN (and permutation) filter.

Algorithm 7 The canonicity filter $\phi_{\simeq_{\mathrm{A}}}$ for affine equivalence.

function $\phi_{\simeq_{\mathrm{A}}}(\tilde{s})$
  return $\phi^{\alpha}_{\simeq_{\mathrm{A}}}(\tilde{s}, (0, \tilde{\diamond}, \tilde{\diamond}))$
end function

function $\phi^{\alpha}_{\simeq_{\mathrm{A}}}(\tilde{s},\ (d, \tilde{\alpha}, \tilde{\beta}))$ ▷ Make a guess for $\tilde{\alpha}$
  if $\tilde{\alpha}(d) = \diamond$ **then**
    for all $\alpha_d \in [0; \deg \tilde{s} - 1]$ **do** ▷ Ensure comparability of $\tilde{\beta}\tilde{s}\tilde{\alpha}$ and $\tilde{s}$
      if $\alpha_d \notin \alpha(\mathbb{F}_2^n)$ **then** ▷ Ensure bijectivity of $\tilde{\alpha}$
        if $\neg\phi^{\beta}_{\simeq_{\mathrm{A}}}(\tilde{s}, (d, \diamond^{\mathcal{A}}_{d\mapsto\alpha_d}\tilde{\alpha}, \tilde{\beta}))$ **then**
          return FALSE
        end if
      end if
    end for
  else
    if $\tilde{s}(\tilde{\alpha}(d)) = \diamond$ **then**
      return TRUE ▷ $\tilde{\beta}\tilde{s}\tilde{\alpha}$ not comparable with $\tilde{s}$
    end if
  end if
end function

function $\phi^{\beta}_{\simeq_{\mathrm{A}}}(\tilde{s},\ (d, \tilde{\alpha}, \tilde{\beta}))$ ▷ Make a guess for $\tilde{\beta}$
  var $d' \in \mathbb{F}_2^n$
  $d' \leftarrow \tilde{s}(\tilde{\alpha}(d))$
  if $\tilde{\beta}(d') = \diamond$ **then**
    var $\tilde{\beta}' \in \mathcal{F}(\mathbb{F}_2^n, \tilde{\mathbb{F}}_2^n)$
    $\tilde{\beta}' \leftarrow \diamond^{\mathcal{A}}_{d'\mapsto\beta_{d'}}\tilde{\beta}$
    for all $\beta_{d'} \in [0; \deg \tilde{s}]$ **do** ▷ Ensure comparability of $\tilde{\beta}\tilde{s}\tilde{\alpha} \leq \tilde{s}$
      if $\beta_{d'} < \tilde{s}(d)$ **then**
        return FALSE ▷ $\tilde{\beta}\tilde{s}\tilde{\alpha} < \tilde{s}$
      else if $d = \deg \tilde{s} - 1$ **then**
        return TRUE ▷ $\tilde{\beta}\tilde{s}\tilde{\alpha}$ not comparable with $\tilde{s}$
      else if $\neg\phi^{\alpha}_{\simeq_{\mathrm{A}}}(\tilde{s}, d+1, \tilde{\alpha}, \tilde{\beta}')$ **then**
        return FALSE
      end if
    end for
    return TRUE
  else
    if $\tilde{\beta}(d') < \tilde{s}(d)$ **then**
      return FALSE ▷ $\tilde{\beta}\tilde{s}\tilde{\alpha} < \tilde{s}$
    else if $\tilde{\beta}(d') > \tilde{s}(d)$ **then**
      return TRUE ▷ $\tilde{\beta}\tilde{s}\tilde{\alpha} > \tilde{s}$
    else if $d = \deg \tilde{s} - 1$ **then**
      return TRUE ▷ $\tilde{s}\tilde{\alpha}$ not comparable with $\tilde{\beta}\tilde{s}$
    else
      return $\phi^{\alpha}_{\simeq_{\mathrm{A}}}(\tilde{s}, d+1, \tilde{\alpha}, \tilde{\beta})$
    end if
  end if
end function

Table 4.1: Canonical APN permutation in $\mathcal{F}(\mathbb{F}_2^3, \mathbb{F}_2^3)$ up to affine equivalence. The algebraic degree is also shown.

0	1	2	3	4	5	6	7	°
0	1	2	4	3	6	7	5	2

Table 4.2: Canonical APN permutations in $\mathcal{F}(\mathbb{F}_2^5, \mathbb{F}_2^5)$ up to to affine equivalence. The algebraic degree and equivalences to power functions are also shown.

#	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	°	Aff.
1	0	1	2	4	3	6	8	16	5	10	15	27	19	29	31	20	7	18	25	21	12	14	24	28	26	11	23	13	30	9	17	22	4	x^{15}
2	0	1	2	4	3	8	13	16	5	11	21	31	23	15	19	30	6	28	29	9	24	27	14	18	10	17	12	26	7	25	20	22	3	x^{11}
3	0	1	2	4	3	8	13	16	5	17	28	27	30	14	24	10	6	19	11	20	31	29	12	21	18	26	15	25	7	22	23	9	3	x^7
4	0	1	2	4	3	8	16	28	5	10	25	17	18	23	31	29	6	20	13	24	19	11	9	22	27	7	14	21	26	12	30	15	2	x^3
5	0	1	2	4	3	8	16	28	5	10	26	18	17	20	31	29	6	21	24	12	22	15	25	7	14	19	13	23	9	30	27	11	2	x^5

Both the APN filter and the affine equivalence canonicity filter have higher specificity with increasing depth. However, they exclude different branches of the tree, and that makes their combination very effective.

The affine equivalence is mainly interesting for permutations. For non-bijective functions the extended affine equivalence is a more appropriate equivalence relation. We will see how to extend Algorithm 7 for EA equivalence in the next subsection. For now, we collect the results for the permutations in $\mathcal{F}(\mathbb{F}_2^n, \mathbb{F}_2^n)$ with $n = 3, 4, 5$, which were calculated using $\phi = \phi_\sigma \wedge \phi_{\text{APN}} \wedge \phi_{\simeq_{\text{A}}}$ (Section 2.3.3, Section 4.1.2 and Algorithm 7).

Theorem 4.17 *All bijective APN functions in $\mathcal{F}(\mathbb{F}_2^3, \mathbb{F}_2^3)$ are affine equivalent, see Table 4.1.*

There are no bijective APN functions in $\mathcal{F}(\mathbb{F}_2^4, \mathbb{F}_2^4)$. This was found by exhaustive search in less than 100 ms.

There are 5 bijective APN functions in $\mathcal{F}(\mathbb{F}_2^5, \mathbb{F}_2^5)$ up to affine equivalence. These functions are all equivalent to APN power functions, see Table 4.2. This was found by exhaustive search in under 24 hours.

4.2.4 EA canonicity filter

We will now extend the treatment of affine equivalence to EA equivalence. First, we show that the canonical representatives of EA equivalence classes vanish on the affine standard basis $B_{\mathcal{A}} := \{0, 2^0, 2^1, \ldots, 2^{n-1}\}$.

Lemma 4.18 *Let $s \in R_{\simeq_{\text{EA}}}$ be a canonical representative for EA equivalence. Then $s(0) = s(2^i) = 0$ for all $0 \leq i < n$.*

Proof: By contradiction. Let $s \in R_{\simeq_{\text{EA}}}$ and let k be the smallest number in $\{0, 2^0, 2^1, \ldots, 2^{n-1}\}$ such that $s(k) \neq 0$. Define:

$$\gamma := \diamondsuit^{\mathcal{A}}_{2^{n-1} \mapsto s(2^{n-1})} \cdots \diamondsuit^{\mathcal{A}}_{2^k \mapsto s(2^k)} \diamondsuit^{\mathcal{A}}_{2^{k-1} \mapsto 0} \cdots \diamondsuit^{\mathcal{A}}_{2^0 \mapsto 0} \diamondsuit^{\mathcal{A}}_{0 \mapsto 0} \tilde{\diamondsuit} \tag{4.12}$$

Then $s \oplus \gamma$ agrees with s on $[0, 2^k - 1]$ and $(s \oplus \gamma)(2^k) = 0 < s(2^k)$. Thus $s \oplus \gamma < s$ which contradicts the assumption that s is the smallest element in its EA equivalence class. □

Let ϕ_0 be a filter predicate that fails all templates that do not have compatible functions vanishing on $B_{\mathcal{A}}$, and let ϕ be any (weak) EA canonicity filter. Then by Lemma 4.18 it follows that $\phi_0 \wedge \phi$ is also a (weak) EA canonicity filter which in general will perform better than ϕ. We now show that $\phi_0 \wedge \phi_{\simeq_{\mathrm{A}}}$ is a weak canonicity filter for EA equivalence:

Proposition 4.19 *Let the template $\tilde{s}$ vanish on its determinate positions of $B_{\mathcal{A}}$. If $\phi_{\simeq_A}(\tilde{s}) =$ FALSE then there exist affine bijections α, β and an affine function γ such that $\tilde{t} := \beta\tilde{s}\alpha + \gamma$ vanishes on its determinate positions of $B_{\mathcal{A}}$ and $\tilde{t} < \tilde{s}$.*

Proof: By the assumption $\phi_{\simeq_{\mathrm{A}}}(\tilde{s}) =$ FALSE we have affine functions α' and β', such that $\tilde{s}' := \beta'\tilde{s}\alpha' < \tilde{s}$. Let $\tilde{t} := \beta\tilde{s}'\alpha \oplus \gamma \leq \tilde{s}' < \tilde{s}$ be the EA canonical representative of $\tilde{s}'$. Then the claim follows immediately by Lemma 4.18. □

It is not very difficult to turn Algorithm 7 into a proper EA canonicity filter by allowing $\beta(d')$ to take arbitrary values and moving the relevant constraint checks into the guess for $\gamma(d)$, which has to be inserted at the locations just before $\phi^{\beta}_{\simeq_{\mathrm{A}}}$ invokes $\phi^{\alpha}_{\simeq_{\mathrm{A}}}$. It is also quite easy to turn $\phi_{\simeq_{\mathrm{A}}}$ or $\phi_{\simeq_{\mathrm{EA}}}$ into an equivalence test rather than a canonicity filter by changing the constraints from

$$(\beta s\alpha \oplus \gamma)(i) < s(i) \text{ to } (\beta s\alpha \oplus \gamma)(i) = t(i) \tag{4.13}$$

where t is a target function. We omit the rather lengthy and mostly redundant pseudocode for all these slight variations.

It turns out that the resulting three stage recursion is not efficient enough to be useful as a filter in a backtrack problem. The EA canonicity filter is vastly outperformed by the weaker affine canonicity filter. Thus, we use $\phi_0 \wedge \phi_{\simeq_{\mathrm{A}}}$ as a weak canonicity filter and apply the techniques described in Section 2.4.2 to recover the EA canonical representatives from the candidate sets.

4.2.5 Evaluation of the EA canonicity filter

We find 16 ($n = 4$) resp. 11768 ($n = 5$) candidates for EA canonical representatives. We use the inefficient EA canonicity filter with a time limit of several seconds per candidate to quickly eliminate 14 resp. 11760 non-canonical candidates. The two remaining candidates for $n = 4$ have a different algebraic degree, therefore they are EA inequivalent (Note 1.1) and canonical. The eight remaining candidates for $n = 5$ are separated by their algebraic degree and the remaining EA equivalences and inequivalences are determined by the EA equivalence test described by the modification (4.13).

An important optimisation of the EA equivalence test is exploiting self-equivalences. The affine permutations α for which we have $s = \beta s\alpha$ form a group that acts on all affine permutations by concatenation from the left. The orbits of this group action induce an equivalence relation, leading to canonical representatives and canonicity filters for α which can be used to optimise the EA equivalence test. We merely hint at this possibility here and treat this subject more thoroughly in the context of CCZ equivalence in Section 4.3.1.

Table 4.3: Canonical functions in $\mathcal{F}(\mathbb{F}_2^3, \mathbb{F}_2^3)$ up to EA equivalence. The algebraic degree is also shown.

0 1 2 3 4 5 6 7	°
0 0 0 1 0 2 4 7	2

Table 4.4: Canonical APN functions in $\mathcal{F}(\mathbb{F}_2^4, \mathbb{F}_2^4)$ up to EA equivalence. The algebraic degree, equivalences to power functions and CCZ equivalences are also shown.

#	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	°	EA	CCZ
1	0 0 0 1 0 2 4 7 0 4 6 3 8 14 10 13	2	x^3	can.
2	0 0 0 1 0 2 4 7 0 4 6 3 8 14 11 12	3	cf. [BCP06]	1

Theorem 4.20 *All APN functions in $\mathbb{F}_2^3\mathbb{F}_2^3$ are EA equivalent, see Table 4.3.*

There are 2 APN functions in $\mathcal{F}(\mathbb{F}_2^4, \mathbb{F}_2^4)$ up to EA equivalence, see Table 4.4. One of those is EA equivalent to a power function.

There are 7 APN functions in $\mathcal{F}(\mathbb{F}_2^5, \mathbb{F}_2^5)$ up to EA equivalence, see Table 4.5. Five of those are EA equivalent to a power function.

Note that the three APN functions in Table 4.4 and 4.5 that are EA inequivalent to any power function actually belong to infinite families of APN functions presented in Theorem 1 and 2 of [BCP06]. Our contribution here is that there are no further equivalence classes with APN functions in these dimensions.

4.3 CCZ equivalence

Affine equivalence, EA equivalence and inverse (for permutations) relations are special cases of a more general affine equivalence, which was introduced by Carlet, Charpin and Zinovievin in [CCZ98] and is thus nicknamed CCZ equivalence. For $s \in \mathcal{F}(\mathbb{F}_2^n, \mathbb{F}_2^n)$ we define the *graph* $\mathcal{G}(s) \subseteq \mathbb{F}_2^n \times \mathbb{F}_2^n$ as the set:

$$\mathcal{G}(s) := \{(x, s(x)) \mid x \in \mathbb{F}_2^n\} \tag{4.14}$$

Definition 4.21 (CCZ equivalence)
Two functions $s, t \in \mathcal{F}(\mathbb{F}_2^n, \mathbb{F}_2^n)$ are said to be CCZ equivalent*, written $s \simeq_{\text{CCZ}} t$, if there exists an affine permutation $\lambda : \mathbb{F}_2^{2n} \to \mathbb{F}_2^{2n}$ such that:*

$$\lambda(\mathcal{G}(s)) = \mathcal{G}(t) \tag{4.15}$$

Note that in this case $\lambda \equiv (\lambda_1, \lambda_2)$ for two affine functions $\lambda_{1,2} : \mathbb{F}_2^{2n} \to \mathbb{F}_2^n$ where $\lambda_1(x, s(x))$ is a bijection.

Proposition 4.22 *Let $s, t \in \mathcal{F}(\mathbb{F}_2^n, \mathbb{F}_2^n)$ be CCZ equivalent. Then s is APN if and only if t is APN.*

Table 4.5: Canonical APN functions in $\mathcal{F}(\mathbb{F}_2^5, \mathbb{F}_2^5)$ up to EA equivalence. The algebraic degree, equivalences to power functions and CCZ equivalences are also shown.

#	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	°	EA	CCZ
1	0	0	0	1	0	2	4	7	0	4	8	13	16	22	28	27	0	8	16	25	5	15	17	26	22	26	14	3	3	13	31	16	2	x^5	can.
2	0	0	0	1	0	2	4	7	0	4	8	13	16	22	28	27	0	8	16	25	5	15	17	26	27	23	3	14	14	0	18	29	2	x^3	can.
3	0	0	0	1	0	2	4	7	0	4	8	13	16	22	29	26	0	8	16	25	5	15	19	24	7	11	27	22	26	20	1	14	3	[BCP06]	1
4	0	0	0	1	0	2	4	7	0	4	8	13	16	22	29	26	0	8	16	25	5	15	19	24	10	6	22	27	23	25	12	3	3	[BCP06]	2
5	0	0	0	1	0	2	4	8	0	3	6	12	7	16	25	23	0	7	3	22	28	19	9	0	19	8	15	28	21	9	29	2	4	x^{15}	can.
6	0	0	0	1	0	2	4	8	0	3	6	16	8	21	26	29	0	5	12	27	20	6	31	16	7	31	8	22	9	26	17	11	3	x^{11}	2
7	0	0	0	1	0	2	4	8	0	3	6	16	8	21	26	29	0	6	15	24	18	3	17	30	2	29	14	20	25	13	9	23	3	x^7	1

CCZ equivalence does not lend itself easily to the definition of an efficient filter predicate in a similar manner to affine equivalence because a relation like $\lambda(\mathcal{G}(\tilde{s})) < \mathcal{G}(\tilde{s})$ means that the graph of every function $s \in \diamondsuit_\bullet \tilde{s}$ must be mapped to a graph by a fixed affine function λ, and these conditions have to be verified. But verifying all these conditions is very expensive.

However, because EA equivalence implies CCZ equivalence, an EA canonicity filter constitutes a weak canonicity filter for CCZ equivalence and we can apply the techniques from Section 2.4.2. As indicated there, this requires an efficient CCZ equivalence test which we will develop in the next section.

4.3.1 CCZ equivalence test

Algorithm 8 implements an efficient test for CCZ equivalence of functions $s, t \in \mathcal{F}(\mathbb{F}_2^n, \mathbb{F}_2^n)$ using a backtrack problem $P := (\mathbb{F}_2^n, \mathbb{F}_2^n, \rho, \phi, \mathbb{F}_2^n \times \tilde{\mathcal{F}}(\mathbb{F}_2^{2n}, \mathbb{F}_2^{2n}) \times \tilde{\mathcal{F}}(\mathbb{F}_2^n, \mathbb{F}_2^n) \uplus \{\text{FALSE}\}, (0, \tilde{\diamond}, \tilde{\diamond}))$ where $\rho(\sigma) = \text{TRUE}$ if and only if exists $\lambda \in \mathbb{F}_2^{2n}$ with $\mathcal{G}(t) = \lambda(\mathcal{G}(s))$ such that $\pi := \lambda_1(x, s(x))^{-1} \equiv \sigma$. The state $(d, \tilde{\lambda}, \tilde{\pi})$ is the search depth d, $\tilde{\pi}$, and the affine, injective template $\tilde{\lambda}$.

The nodes in the search tree correspond to left-refinements for π, such that at depth d it holds that

$$\tilde{\lambda}(\mathcal{G}(s)) \setminus \{\diamond\} = \mathcal{G}(\tilde{t} \mid D_{\tilde{t}}) \tag{4.16}$$

for a template $\tilde{t} \ni t$ which is determinate on all positions less than d. Due to affine refinements of $\tilde{\lambda}$, the template $\tilde{t}$ may be determinate on additional positions. This means that the algorithm recursively finds the preimage $(a, s(a)) \in \mathcal{G}(s)$ of a point $(d, t(d)) \in \mathcal{G}(t)$ with $\pi(d) = a$, while ensuring affinity and injectivity of $\tilde{\lambda}$.

Note 4.23 *We choose to refine π rather than $\pi^{-1} = \lambda_1(x, s(x))$ because then $\tilde{t}$ in (4.16) is determinate on all positions less than d. This makes it possible to modify Algorithm 8 to determine CCZ canonicity of arbitrary functions s. The modified algorithm finds any function $t < s$ with $t \simeq_{\text{CCZ}} s$ if such a function exists, otherwise it fails.*

In this way we can quickly determine that candidates no. 3 and 4 in Table 4.5 are not canonical. However, these functions can also be eliminated, albeit less quickly, using explicit equivalence tests and thus the classification presented here does not depend on this optimisation. Furthermore, the canonicity test is not efficient enough to make a

statement about the remaining candidates, therefore we omit the details of the necessary modifications to Algorithm 8.

Algorithm 8 The filter ϕ_π for a CCZ equivalence test $s \simeq_{\mathrm{CCZ}} t$.

function $\phi_\pi((d, \tilde{\lambda}, \tilde{\pi}), a)$
  if $\tilde{\pi}(d) \neq \diamond$ **then**
    return $(d+1, \tilde{\lambda}, \tilde{\pi})$ ▷ Verified previously.
  end if
  if $\tilde{\lambda}(a, s(a)) \neq \diamond$ **then**
    return FALSE ▷ Position a already used.
  end if
  if $(d, t(d)) \in \tilde{\lambda}(\mathbb{F}_2^{2n})$ **then**
    return FALSE ▷ Ensure bijectivy of $\tilde{\lambda}$.
  end if
  if $\phi_{\simeq_\pi}((d, \tilde{\pi}), a) =$ FALSE **then**
    return FALSE ▷ Self-equivalences, see text.
  end if
  $\tilde{\pi} \leftarrow \diamondsuit_{d \mapsto a} \tilde{\pi}$
  var $\tilde{\lambda}' \in \mathbb{F}_2^{2n}$
  $\tilde{\lambda}' \leftarrow \diamondsuit^{\mathcal{A}}_{(a,s(a)) \mapsto (d,t(d))} \tilde{\lambda}$
  for all $(x, s(x)) \in \tilde{\lambda}'^{-1}(\mathcal{G}(t)) \setminus \tilde{\lambda}^{-1}(\mathcal{G}(t))$ **do** ▷ Follow implications.
    var $(y, t_y) \in \tilde{\mathbb{F}}_2^n \times \tilde{\mathbb{F}}_2^n$
    $(y, t_y) \leftarrow \tilde{\lambda}'(x, s(x))$
    if $\tilde{\pi}(y) \neq \diamond$ **then**
      return FALSE ▷ Ensure bijectivity of $\tilde{\lambda}'_1(x, s(x))$.
    end if
    if $t_y \neq t(y)$ **then**
      return FALSE ▷ Ensure $\mathcal{G}(t) \supseteq \tilde{\lambda}'(\mathcal{G}(s)) \setminus \{\diamond\}$.
    end if
    if $\phi_{\simeq_\pi}((y, \tilde{\pi}), t_y)$ **then**
      return FALSE ▷ Self-equivalences, see text.
    end if
    $\tilde{\pi} \leftarrow \diamondsuit_{y \mapsto x} \tilde{\pi}$
  end for

  return $(d+1, \tilde{\lambda}', \tilde{\pi})$
end function

We optimise further using self-equivalences: Let Λ_t be the subgroup of affine permutations λ_t which stabilize $\mathcal{G}(t)$.

With $\mathcal{G}(t) = \lambda(\mathcal{G}(s))$ we have $\mathcal{G}(t) = \lambda_t\lambda(\mathcal{G}(s))$ for all $\lambda_t^{-1} \in \Lambda_t$. If also $\mathcal{G}(t) = \lambda'(\mathcal{G}(s))$ we have $\lambda' = (\lambda'\lambda^{-1})\lambda$ with $\lambda'\lambda^{-1} \in \Lambda_t$ and the orbit of λ under the action of Λ_t is the set of all affine functions λ' which map $\mathcal{G}(s)$ to $\mathcal{G}(t)$.

The group Λ_t induces a permutation subgroup Π_t which acts on π. The orbit of π under Π_t is the set of all permutations π' for which a λ' exists with $\pi' = \lambda'_1(x, s(x))^{-1}$

and $\mathcal{G}(t) = \lambda'(\mathcal{G}(s))$. This allows us to search only for the canonical representative in the orbit of π by a stateful canonicity filter $\phi_{\simeq_\pi}$ for $\tilde{\pi}$ that is usefully defined on arbitrary (not just left-) refinements. Note that Π_t can be found incrementally using Algorithm 8 and canonicity filters derived from subgroups of Π_t.

Example 4.24 *Consider the function t that is no. 1 in Table 4.5. The group Λ_t induces a permutation subgroup Π_t with elements π_t generated by:*

(0)(1)(2 17 25 22 28)(3 16 24 23 29)(4 21 12 26 6)(5 20 13 27 7)(8 15 10 30 19)(9 14 11 31 18)

(0)(1 2 10 13 4)(3 8 7 9 5)(6 11 15 14 12)(16 27 20 26 22)(17 25 30 23 18)(19)(21 24 28 29 31)

(0 1)(2 3)(4 5)(6 7)(8 9)(10 11)(12 13)(14 15)(16 17)(18 19)(20 21)(22 23)(24 25)(26 27)(28 29)(30 31)

This leads to the following canonicity test for $\tilde{\pi}$:

function $\phi_{\simeq_\pi}((d, \tilde{\pi}), a)$
  if ($d = 0$ **and** $a \neq 0$) **or** ($d = 1$ **and** $a \neq 1$) **then**
    return FALSE
  end if
  if $\tilde{\pi}(2) \neq \diamond$ **and** $d = 17, 22, 25$ or 28 **then and** $a < \tilde{\pi}(2)$ **then**
    return FALSE
  end if
  return TRUE
end function

4.3.2 Weak CCZ canonicity filter

As mentioned before, an EA canonicity filter constitutes a weak canonicity filter for CCZ equivalence. We will now apply the techniques from Section 2.4.2 to this situation. The set of candidates is the set of all EA canonical APN functions, and our task is to purge from this set those functions which are not CCZ canonical. We use CCZ invariants and the CCZ equivalence test from Section 4.3.1; see Note 4.23 why we forfeit the CCZ canonicity test.

CCZ invariants: The Walsh spectrum of $s \in \mathcal{F}(\mathbb{F}_2^n, \mathbb{F}_2^n)$ is the multi-set $W_s := \{|w_s(a,b)| \mid a, b \in \mathbb{F}_2^n, b \neq 0\}$ where $w_s(a,b) = \sum_{x \in \mathbb{F}_2^n} (-1)^{bs(x)+ax}$, and is known to be a CCZ invariant. For $n = 4$, all APN functions have the same Walsh spectrum. But for $n = 5$, the APN functions with degree 2 and 3 have the Walsh spectrum $(0; 527)$, $(2^8; 496)$, $(2^{32}; 1)$ given as (value; multiplicity), while those with degree 4 have the Walsh spectrum $(0; 217)$, $(2^4; 465)$, $(2^8; 310)$, $(2^{12}; 31)$, $(2^{32}; 1)$. Therefore, the CCZ equivalence classes split into two sets which we can treat separately.

CCZ equivalence: We only deal with $n = 5$ here, the case $n = 4$ is analogous. We already know due to the Walsh spectrum invariant that no. 1 and 5 in Table 4.5 are CCZ canonical. No. 5 is alone in its class of EA canonical representatives with degree 4, so we turn our attention to the remaining candidates no. 2, 3, 4, 6 and 7. The optimised equivalence test $\simeq_{\text{CCZ}}^{\text{no. 1}}$ from Example 4.24 can be used to find out that no. 2 is CCZ inequivalent and no. 3 is CCZ equivalent to no. 1. That no. 3, 6 and 7 are not canonical follows similarly, following Algorithm 3 faithfully. We find:

Theorem 4.25 *All APN functions in $\mathcal{F}(\mathbb{F}_2^4, \mathbb{F}_2^4)$ are in the same CCZ equivalence class.*

There are only three APN functions in $\mathcal{F}(\mathbb{F}_2^5, \mathbb{F}_2^5)$ up to CCZ equivalence. They are no. 1, 2, and 5 respectively in Table 4.5. Any APN function in $\mathcal{F}(\mathbb{F}_2^5, \mathbb{F}_2^5)$ is CCZ equivalent to a power function.

Appendix A

Implementation notes

The pseudocode in this thesis is provided for mathematical clarity, but it is far remote from the actual implementation of the algorithms on a real machine architecture. As Golomb and Baumert point out in [GB65]:

> In fact "most" combinatorial problems grow to such an extent that there is at most one additional case beyond hand computation that can be handled by our present high speed digital computers. For many of these problems it takes an extremely sophisticated application of the principles of backtrack even to do this one additional case. Thus the success or failure of backtrack often depends on the skill and ingenuity of the programmer in his ability to adapt the basic methods to the problem at hand and in his ability to reformulate the problem so as to exploit the characteristics of his own computing device. That is, backtrack programming (as many other types of programming) is somewhat of an art.

Therefore, we believe that to give a description of an algorithm without saying how it can be efficiently implemented on an existing architecture is like telling a joke but omitting the punchline. In this section we will give a different presentation of the basic Algorithm 1 which is closer to the actual implementation used in producing the results of this thesis. We will also give some general principles which lead to this implementation.

All algorithms were implemented in the C programming language [KR88], compiled using the GNU C Compiler [GNU06] and executed on Pentium 4 2.8 GHz processor running the Ubuntu [Ubu06] operating system based on GNU/Linux.

The following tricks of the trade were used to arrive at the implementation:

1. Efficient data structures
2. Globally shared state and rollback functions
3. Static compilation for a fixed problem size
4. Reordered execution flow to delay expensive operations
5. Exploitation of machine characteristics

We did not try to pessimize the algorithms to compare the performance of these optimisations against possible alternatives. Some algorithms were prototyped in an interpreted programming language without any optimisation before cast into C. These prototypes were improved upon by several orders of magnitude. Considering that all computations for this thesis required about four weeks in total, even differences of a few percent are significant.

A.1 Efficient data structures

Mathematical objects are built for clarity, not algorithmic performance. To improve the performance of algorithms, it is often advisable to extend the mathematical object by additional properties – usually precalculated values of interesting functions – which are then updated whenever the object itself is modified. The literature on efficient data types for commonly encountered objects is vast and extensive (see for example the bibliography in [AHU83]), as this is one of the most commonly used methods of optimisation.

Example A.1 *The backtrack algorithm makes frequent use of the degree* $\deg \tilde{f}$ *of a function template* $\tilde{f}$*. Therefore, performance can be improved by adding the degree of* $\tilde{f}$ *to the implementation of the data type for* $\tilde{f}$*. When* $\tilde{f}$ *is refined to* $\tilde{f}'$ *by a one-step refinement, the degree of* $\tilde{f}'$ *can be computed efficiently from the degree of* $\tilde{f}$ *by adding 1.*

Also used in this thesis is the representation of sets as bit vectors [Leh57] and functions as value vectors.

It is important, but also often easy, to ensure consistency among all members of the data structures if this optimisation is used.

A.2 Global state and rollback functions

Recursive algorithms frequently use local state to hold copies of important data objects private to a particular invocation of the procedure. The incurred direct cost is creating a temporary copy of the object. The indirect cost is reduced sharing which affects memory caching negatively. One solution is to share as much state as possible in global state, and rollback any local changes before returning from the local function.

Example A.2 *At each step of the recursion the template function* $\tilde{f}$ *is refined by a one-step left-refinement. However, that refinement does not change the existing determined positions of* $\tilde{f}$*. As only the determined positions of* $\tilde{f}$ *are ever used in the execution of the algorithm, the same object instance can be used for all instances of the* Backtrack *and even the* ϕ *functions.*

Algorithm 9 shows the optimised version of the backtrack algorithm with this modification. Also, the state of the filter is split into a global part Σ_G *and a local part* Σ_L*.*

A.3 Static compilation for a fixed problem size

Mathematical problems are often formulated in a general way with a dependency on a flexible parameter set. For example, the APN Definition 4.1 is valid for all $s \in \mathcal{F}(\mathbb{F}_2^n, \mathbb{F}_2^n)$ with $n \in \mathbb{N}$. This allows reasoning over a class of interesting objects. But in the actual implementation of algorithms, parametrization comes at an indirect cost of increased code size and less optimised data structures.

A good compromise is to use programming language support to select different implementations fine-tuned for a given parameter that is fixed statically at compile time. This allows for more opportunities for optimisation by the programmer as well as by the compiler itself.

Example A.3 *Most sets that occur in the algorithms presented here have at most $\mathbb{F}_2^n$ elements. If n is fixed at compile time, macros can be used to select the most efficient bit vector representation for such sets of fixed sizes that is available on the hardware architecture. In particular, the maximum size of a set is statically determined and thus does not need to be handled at run-time.*

A.4 Reordered execution flow

In Algorithm 1, the filter function ϕ is invoked immediately after creating a new instance of the Backtrack function. However, if ϕ fails, the main body of the Backtrack function is not executed. In this case, it is preferable to execute the filter ϕ first and only create a new instance of Backtrack if the result is true.

In general, it is a good guideline to execute an algorithm in an order which maximizes the impact of decisions that can be made early. This is why in a backtrack problem the next parameter should be selected from the most constrained parameter set.

Example A.4 *Algorithm 9 shows the optimised backtrack algorithm for stateful filters.*

It is also worth noting that some execution flow restructuring can be done by the compiler. However, this requires special attention by the developer to avoid pitfalls due to the language specification.

A.5 Exploitation of machine characteristics

All of the above items have one goal: to optimize the execution performance of an algorithm on a specific architecture. There are numerous other techniques which can be deployed which are not listed above. The following list is not meant to be exhaustive.

- Branch prediction hints which allow optimisation of the execution of conditional statements in the instruction pipeline of the CPU
- Special CPU instructions which are not exposed by the programming language (and thus can not be used in a portable manner), like finding the most/least significant bit of a number

Algorithm 9 Solve the backtrack problem $P := (A, B, \rho, \phi, \Sigma_G \times \Sigma_L, S_{\tilde{\diamond}})$, optimised version.

```
var depth ∈ ℕ_0                                          ▷ Global declarations
var f̃ array [0; #A − 1] of B
var S_G ∈ Σ_G

procedure BacktrackOpt
    depth ← depth + 1
    for all b ∈ B do
        var S_L ∈ Σ_L
        f̃[depth − 1] ← b
        S_L ← φ()                                        ▷ Modifies S_G
        if S_G ∉ Σ_FALSE then
            if depth = #A then
                Output(f̃)                                ▷ Found solution f̃
            else
                BacktrackOpt ()
            end if
        end if
        φ_R(S_L)                                         ▷ Rollback, modifies S_G
    end for
    depth ← depth - 1;
end procedure

depth ← 0                                                ▷ Initialisation
S_G ← S_◇̃
if S_G ∉ Σ_FALSE then
    BacktrackOpt ()                                      ▷ Invocation
end if
```

- Special compiler options to control optimisation settings and the target architecture for code generation
- Knowledge about the cache layout of the target architecture to optimize cache locality [CHL99]

Last but not least, one should also know when *not* to use certain "optimisations." Programmers are notoriously bad at judging the run-time performance of their own programs, and the final judge is the experiment. Profiling tools can help to analyze the call tree and memory access patterns, which establishes a factual basis from which optimisation decisions can be made.

A.6 Engineering and mathematics

The above considerations are in fact more concerned about engineering than mathematics, although there are many interesting mathematical problems hiding in the description and analysis of these techniques. However, as Golomb and Baumert point out so eloquently, there is no single technique that leads to success for these type of problems. In the end, the engineers may be reassured that to achieve the "one additional case beyond hand computation" their skills in the art of programming are required, while the mathematicians may be reassured that the most significant steps forward in algorithm design are derived from new mathematical insights. We hope that this thesis exemplifies both to a modest extent.

Bibliography

[AHU83] Aho, Hopcroft, and Ulmann. *Data Structures and Algorithms.* Addison-Wesley, 1983.

[BCBP03] Alex Biryukov, Christophe De Cannière, An Braeken, and Bart Preneel. A toolbox for cryptanalysis: Linear and affine equivalence algorithms. In *EUROCRYPT*, pages 33–50, 2003.

[BCFL06] Lilya Budaghyan, Claude Carlet, Patrick Felke, and Gregor Leander. An infinite class of quadratic APN functions which are not equivalent to power mappings. In *IEEE International Symposium on Information Theory*, pages 2637–2641, 2006.

[BCP06] Lilya Budaghyan, Claude Carlet, and Alexander Pott. New classes of almost bent and almost perfect nonlinear polynomials. In *IEEE Transactions on Information Theory*, volume 52, pages 1141–1152, 2006.

[BD94] T. Beth and C. Ding. On almost perfect nonlinear permutations. In *EUROCRYPT '93*, pages 65–76, 1994.

[BS91] Eli Biham and Adi Shamir. Differential cryptanalysis of DES-like cryptosystems. *J. Cryptology*, 4:3–72, 1991.

[Car06] Claude Carlet. Vectorial boolean functions for cryptography. To appear as chapter of the volume *Boolean Methods and Models*, published by Cambridge University Press, Eds Yves Crama and Peter Hammer, Nov. 2006.

[CCZ98] Claude Carlet, Pascale Charpin, and Victor Zinoviev. Codes, bent functions and permutations suitable for DES-like cryptosystems. *Designs, Codes and Cryptography*, 15:125–156, 1998.

[CHL99] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 1999.

[dH03] Xiang dong Hou. Affinity of permutations of $\mathbb{F}_2^n$. In *Proc. of the Workshop on Coding and Cryptography*, pages 273–280, 2003.

[Dil06] John F Dillon. APN polynomials and related codes. Banff International Research Station workshop on Polynomials over Finite Fields and Applications, Nov. 2006.

[Dob99a] Hans Dobbertin. Almost perfect nonlinear power functions over $GF(2^n)$: the Niho case. *Information and Computation*, 151:57–72, 1999.

[Dob99b] Hans Dobbertin. Almost perfect nonlinear power functions over $GF(2^n)$: the Welch case. *IEEE Trans. Inform. Theory*, 45:1271–1275, 1999.

[Dob00] Hans Dobbertin. Almost perfect nonlinear power functions over $GF(2^n)$: a new case for n divisible by 5. In *Proceedings of Finite Fields and Aplications FQ5*, pages 113–121, 2000.

[EKP06] Yves Edel, Gohar Kyureghyan, and Alexander Pott. A new APN function which is not equivalent to a power mapping. In *IEEE Transactions on Information Theory*, volume 52, pages 744–747, 2006.

[Far78] I. A. Faradžev. Constructive enumeration of combinatorial objects. In *Problèmes Combinatoires et Théorie des Graphes*, volume 260, pages 131–135. Coloques internationaux C.N.R.S., 1978.

[GB65] Solomon W. Golomb and Leonard D. Baumert. Backtrack programming. *J. ACM*, 12(4):516–524, 1965.

[GNU06] `http://gcc.gnu.org/`, 2006.

[Gol68] R. Gold. Maximal recursive sequences with 3-valued recursive cross-correlation functions. *IEEE Trans. Inform. Theory*, 14:154–156, 1968.

[JW93] H. Janwa and R. Wilson. Hyperplane sections of Fermat varieties in p^3 in char. 2 and some applications to cyclic codes. In *Proceedings of AAECC-10*, pages 180–194, 1993.

[Kas71] T. Kasami. The weight enumerators for several classes of subcodes of the second order binary Reed-Muller codes. *Inform. and Control*, 18:369–394, 1971.

[Knu75] Donald E. Knuth. Estimating the efficiency of backtrack programs. *Mathematics of Computation*, 29:121–136, 1975.

[KR88] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, second edition, 1988.

[Leh57] Derrick H. Lehmer. Combinatorial problems with digital computers. In *Proc. of the Fourth Canadian Math. Congress*, pages 160–173, 1957.

[Lor64] C. S. Lorens. Invertible boolean functions. *IEEE Trans. Electronic Computers*, 13(5):529–541, 1964.

[Mil98] William Millan. How to improve the nonlinearity of bijective s-boxes. In *ACISP '98: Proceedings of the Third Australasian Conference on Information Security and Privacy*, pages 181–192, London, UK, 1998. Springer-Verlag.

[NK93] Kaisa Nyberg and L. R. Knudsen. Provable security against differential cryptanalysis. In *Crypto '92*, 1993.

[Nyb94] Kaisa Nyberg. Differentially uniform mappings for cryptography. In *EUROCRYPT '93*, pages 55–64, 1994.

[Rea78] Ronald C. Read. Every one a winner. *Annals of Discrete Mathematics*, 2:107–120, 1978.

[Slo06] N. J. A. Sloane. The on-line encyclopedia of integer sequences. `http://www.research.att.com/~njas/sequences/`, 2006.

[Tur36] Alan Turing. On computable numbers, with an application to the entscheidungsproblem. In *Proceedings of the London Mathematical Society*, volume 42 of *2*, pages 230–265, 1936.

[Ubu06] `http://www.ubuntu.com/`, 2006.

[Wal60] R. J. Walker. An enumerative technique for a class of combinatorial problems. In *Proc. of the Symposium in Applied Mathematics*, volume 10, pages 91–95. Amer. Math. Soc., 1960.