Efficiently deciding and recovering CCZ and EA equivalence for arbitrary vectorial Boolean functions using the partition refinement framework

Nikolay Kaleyski[1] and Joakim Sunde[1*]

[1*]University of Bergen, Bergen, Norway.

*Corresponding author(s). E-mail(s): Joakim.Sunde@uib.no;

Abstract

We propose an algorithm based on the partition refinement framework for testing and recovering CCZ and EA equivalence between a pair of vectorial Boolean functions. In contrast to existing approaches, our method can be used for any pair of functions regardless of their algebraic degree, image size and other properties, and it outperforms all currently known algorithms in terms of time and memory. The algorithm can also compute the automorphism group of the functions efficiently. Our implementation is available at https://github.com/zskiley/CCZ-EA-equivalence.

Keywords: boolean functions, EA-equivalence, CCZ-equivalence, APN, partition refinment, S-boxes

1 Introduction

Vectorial Boolean functions, i.e. mappings between the vector spaces $\mathbb{F}_2^n$ and $\mathbb{F}_2^m$ over the finite field $\mathbb{F}_2$, play a central role across mathematics and computer science, including cryptography, coding theory, combinatorics, and various other disciplines. Also referred to as (n, m)-functions, they have been studied in multiple different contexts, and in particular as building blocks of symmetric block ciphers, where they are referred to as S-boxes, or substitution boxes. Numerous block cipher designs, such as Feistel networks and substitution-permutation networks, incorporate S-boxes as fundamental components in their design. The efficiency and security of the cipher then directly depend on the properties of the underlying S-boxes, which has motivated extensive

research into classes of vectorial Boolean functions that have cryptographically desirable properties, such as low differential uniformity, high nonlinearity and algebraic degree.

A significant challenge in the investigation of cryptographically optimal vectorial Boolean functions is the sheer number of mappings that have to be considered. Since an (n, m)-function has a domain consisting of 2^n elements and a codomain consisting of 2^m elements, there are $(2^m)^{2^n}$ such functions; even for relatively small values of the dimensions n and m, this makes it infeasible to traverse the space of all such functions in order to identify the ones with optimal properties.

In practice, these functions are classified only up to suitable equivalence relations, see for example [1–4]. The most commonly used notions are affine, extended-affine (EA), and Carlet–Charpin–Zinoviev (CCZ) equivalence. Concretely, for functions $F, G : \mathbb{F}_2^n \to \mathbb{F}_2^m$:

- F and G are *affine equivalent* if there exist affine permutations A_1 of $\mathbb{F}_2^n$ and A_2 of $\mathbb{F}_2^m$ such that
$$G = A_2 \circ F \circ A_1.$$
- F and G are *EA-equivalent* if there exist affine permutations A_1 of $\mathbb{F}_2^n$ and A_2 of $\mathbb{F}_2^m$, and an affine map $A_3 : \mathbb{F}_2^n \to \mathbb{F}_2^m$, such that
$$G = A_2 \circ F \circ A_1 + A_3.$$
- F and G are *CCZ-equivalent* if there exists an affine permutation A of $\mathbb{F}_2^{n+m}$ such that
$$A(\Gamma_F) = \Gamma_G,$$
where $\Gamma_F := \{(x, F(x)) : x \in \mathbb{F}_2^n\}$ denotes the graph of F.

Moreover, these notions form a hierarchy: affine equivalence is a special case of EA-equivalence, and EA-equivalence is a special case of CCZ-equivalence. In particular,

$$F \sim_{\text{aff}} G \implies F \sim_{\text{EA}} G \implies F \sim_{\text{CCZ}} G.$$

Determining whether two functions are equivalent with respect to any of these notions is a computationally challenging problem, and several algorithms have been proposed. The "back and forth" algorithm proposed in [5] can be used to test affine equivalence of permutations; this can also be extended to non-bijective functions at a higher computational cost. For functions of high algebraic degree, one can use the approach from [6]. For k-to-1 functions, which appear quite frequently among several important classes of cryptographically optimal functions (such as APN and PN functions), an efficient method for testing affine equivalence has been proposed in [7]. Algorithms have also been developed for testing equivalence in some special cases, such as when the function is a concatenation of smaller boxes [8, 9] , or when a certain structure on the equivalence relation itself can be assumed [10].

For determining EA and CCZ equivalence, the best currently known general method is reducing the problem to the problem of deciding code equivalence [11]. For specific classes of functions, several dedicated algorithms are known. The algorithm

in [12] can be used to test EA-equivalence between quadratic functions; however, for important classes of functions such as APN functions, the running times of this algorithm are comparable to a brute-force search. The approach in [13] derives information about the shape of the EA-equivalence between two functions based on certain invariants; however, it does not appear to be efficient enough to be used in practice for higher dimensions.

In practice, when trying to compute CCZ or EA equivalence one first tries to prove inequivalence by computing invariants for both functions, see e.g. [14]. Only after this does one run the more expensive equivalence algorithms on the functions. One of the main problems of equivalence testing is that the most interesting cryptographic functions (having low differential uniformity, high nonlinearity, etc) tend to be the hardest to test for equivalence. Testing EA- and CCZ-equivalence using code isomorphism becomes computationally expensive already for (n, n)-functions in dimension $n = 7$ (where some tests can take up to 2 hours), while for dimensions $n > 10$ the memory consumption of this method quickly grows which makes it difficult to use in practice.

Computing automorphism groups under CCZ- and EA-equivalence also has further applications. Such groups can be used as invariants for classification, and their orbit structure can be exploited to accelerate other computations. For example, [15] uses the CCZ automorphism group for orbit reduction in an algorithm that computes the minimum Hamming distance of a vectorial Boolean function to the set of affine functions.

1.1 Our Contribution

We propose a general partition-refinement algorithm for deciding and recovering CCZ- and EA-equivalence between vectorial Boolean functions. The same framework also computes the corresponding automorphism groups. For two functions $F, G : \mathbb{F}_2^n \to \mathbb{F}_2^m$, the CCZ-equivalence problem asks to determine whether there exists an affine permutation

$$A : \mathbb{F}_2^{n+m} \to \mathbb{F}_2^{n+m}$$

such that $A(\Gamma_F) = \Gamma_G$, where

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

Our algorithm constructs such a map incrementally. The computation is organized as a depth-first search (DFS). At each node of the search tree, we store a partial affine map A, together with two pairs of ordered partitions: point partitions of Γ_F and Γ_G, and hyperplane partitions of $\mathbb{F}_2^{n+m}$. These partitions are maintained so that any affine extension of the current partial map must send each cell on the F-side to the corresponding cell on the G-side. Consequently, the possible images of a newly chosen point are restricted to a single corresponding cell, often reducing the branching factor substantially.

The main refinement step alternates between the point and hyperplane partitions. A hyperplane partition induces signatures on points by counting, for each point, in how many hyperplanes from each cell it is contained. Conversely, a point partition induces

signatures on hyperplanes by counting their intersections with the point cells. Whenever these signatures split cells differently on the two sides, the current partial map cannot be extended to a CCZ-equivalence and the corresponding branch is discarded. Otherwise, the refined partitions give stronger constraints for the next branching step. In practice, we use weighted signatures, which retain the same validity property while allowing the required incidence information to be computed efficiently using the fast Fourier–Hadamard transform.

A second important ingredient is the use of automorphism groups during the search itself. If a subgroup of $\mathrm{Aut}(G)$ is known, then at any search node we restrict this group to the stabilizer of the current partial map. Points in the same orbit under this stabilizer give equivalent branching choices, so it is sufficient to branch only on one representative from each orbit. This orbit pruning can reduce the search tree by several orders of magnitude, especially for functions with large automorphism groups. Moreover, the automorphism group need not be known in advance: the same recursive search can be run with $F = G$ to compute $\mathrm{Aut}(F)$. During this automorphism search, every newly found automorphism is added to the current subgroup, and the orbit information used for pruning is updated dynamically throughout the remaining search.

The EA-equivalence algorithm is obtained by imposing some additional constraints on the CCZ mapping. Thus, the CCZ and EA algorithms share the same refinement, branching, and automorphism-pruning machinery, differing only in the final validity test and in the corresponding automorphism groups. This gives a single unified approach for testing CCZ-equivalence, testing EA-equivalence, computing CCZ automorphism groups, and computing EA automorphism groups.

1.2 Organization of the paper

The remainder of this paper is organized as follows. In Section 2, we cover the preliminaries and background needed for the rest of the discussion. Section 3 describes the main principles behind our algorithms. In Section 4, we discuss how to compute the so-called signatures that are used in pruning the search tree. Section 5 discusses how to use the automorphism groups to speed up the algorithm. In Section 6 we present the complete algorithms, and in Section 7, we present our empirical computational results, demonstrating the efficiency of our algorithms in practice. Finally, we present a complexity and correctness analysis of the algorithms in Section 8.

1.3 Previous work and state of the art

We briefly recall the main existing approaches for testing and reconstructing CCZ- and EA-equivalence between vectorial Boolean functions. In practice, one usually first computes equivalence invariants in order to rule out inequivalent pairs as early as possible. Such invariants cannot prove equivalence, but they are often effective for subdividing large collections of functions into smaller candidate sets; see, for instance, [14] for a survey on the topic.

For arbitrary functions, the general method is to reduce the problem to code equivalence. The CCZ-equivalence of two (n, m)-functions can be characterized as the permutation equivalence of certain linear codes associated with the functions [11, 16].

More precisely, for each function F one constructs a code $C_1(F)$ such that

$$F \sim_{\mathrm{CCZ}} G \quad \Longleftrightarrow \quad C_1(F) \sim_{\mathrm{perm}} C_1(G).$$

, where $\sim_{\mathrm{perm}}$ denotes permutation equivalence between linear codes. There is an analogous construction involving a different associated linear code $C_2(F)$ for EA-equivalence, giving

$$F \sim_{\mathrm{EA}} G \quad \Longleftrightarrow \quad C_2(F) \sim_{\mathrm{perm}} C_2(G).$$

The resulting code-equivalence instances can then be solved using general-purpose partition-backtracking algorithms, such as those following Leon's framework [17]. To the best of our knowledge, this is currently the only general approach for testing CCZ-equivalence, apart from exhaustive search, that can be applied to an arbitrary pair of functions. In our experiments (see Section 7), Magma provides the fastest available implementation of this strategy.

Several further specialized algorithms are known for EA-equivalence. The method of [13] uses additive invariants to restrict the possible affine permutations appearing in an EA-equivalence. It has relatively low memory requirements and can therefore be applied in dimensions where the code-equivalence approach may become memory-intensive. However, for important classes of functions (such as AB functions, and permutations), the induced partitions are trivial, in which case the method degenerates to an exhaustive search.

For quadratic functions, Canteaut, Couvreur, and Perrin proposed an algorithm based on the linearized Jacobian matrix [12]. The central observation is that EA-equivalent quadratic functions have Jacobian matrices related by the linear parts of the affine permutations defining the equivalence. This gives structural constraints that reduce the search space and can be used to test or recover EA-equivalence. The method is particularly relevant because for quadratic APN functions, CCZ and EA equivalence coincide [18]. This algorithm however, does not seem to be efficient for APN functions, although it does perform better for functions with non-uniform differential spectra. That paper also introduces the orthoderivative which can often be used to prove inequivalence of quadratic APN functions efficiently.

In addition, a lot of research has been done on algorithms for testing less general equivalence relations that can be seen as more computationally tractable than EA- and CCZ-equivalence, especially in cases where the functions being tested satisfy certain properties. These include algorithms for testing linear and affine equivalence between a pair of permutations [5], 3-to-1 functions [7], and functions of high algebraic degree [6]. Methods for testing specific subcases of EA-equivalence (referred to as "restricted EA-equivalence") have also been investigated in the literature [10].

Despite all this work, the code isomorphism reduction has so far remained the only reliable method for testing CCZ- and EA-equivalence in general. The algorithms introduced in this paper are designed to handle arbitrary functions while avoiding the code-equivalence reduction and the associated high memory costs. Our algorithms also allow for computing the automorphism group of a given function under EA-

and CCZ-equivalence, which can be used to further speed up computations, and is of independent interest.

2 Notation and preliminaries

2.1 Vectorial Boolean functions

We denote by $\mathbb{F}_2$ the finite field with two elements. For any positive integer n, $\mathbb{F}_2^n$ is the n-dimensional vector space over $\mathbb{F}_2$, and $\mathbb{F}_{2^n}$ is the extension field of degree n over $\mathbb{F}_2$. We note that $\mathbb{F}_2^n$ can naturally be identified with $\mathbb{F}_{2^n}$ by associating every element of the field $\mathbb{F}_{2^n}$ with its coordinate vector in $\mathbb{F}_2^n$. Thus, $\mathbb{F}_2^n$ and $\mathbb{F}_{2^n}$ can be used interchangeably.

An (n, m)-function, for positive integers n and m, also called a vectorial Boolean function (VBF), is a mapping from $\mathbb{F}_2^n$ to $\mathbb{F}_2^m$ or, equivalently, from $\mathbb{F}_{2^n}$ to $\mathbb{F}_{2^m}$.

We also note another natural generalization of (n, m)-functions, in which a finite field $\mathbb{F}_p$ of arbitrary prime characteristic p is considered in place of $\mathbb{F}_2$. In this setting, the objects under consideration would be mappings from $\mathbb{F}_p^n$ to $\mathbb{F}_p^m$ or, equivalently, from $\mathbb{F}_{p^n}$ to $\mathbb{F}_{p^m}$. These are sometimes referred to as (n, m, p)-functions, or as discrete vectorial functions in the literature. All the work and discussion presented hereafter can naturally be adapted to the case of discrete vectorial functions with $p > 2$. We restrict ourselves to the case of $p = 2$ in the presentation in order to simplify the discussion and implementation, and since this is arguably the most relevant case in practice.

An (n, m)-function, for given positive integers n and m, can be represented in numerous ways. We will represent them as truth tables which can be implemented as an array of length 2^n, with the inputs (interpreted as integers between 0 and $2^n - 1$) being used as indices into the array, and the elements of the array containing the corresponding outputs.

VBFs can also be represented in terms of polynomials. Any (n, m)-function F can be uniquely represented as a multivariate polynomial of the form

$$F(x_1, x_2, \ldots, x_n) = \sum_{I \in \mathcal{P}(\{1,2,\ldots,n\})} a_I \prod_{i \in I} x_i, \tag{1}$$

where $\mathcal{P}(1, 2, \ldots, n)$ denotes the power set of $\{1, 2, \ldots, n\}$, and the coefficients a_I belong to $\mathbb{F}_2^m$ for all $I \in \mathcal{P}(1, 2, \ldots, n)$. This is referred to as the algebraic normal form, or ANF.

Given an (n, m)-function F with ANF (1), its algebraic degree, denoted $\deg(F)$, is equal to the polynomial degree of the ANF; in other words,

$$\deg(F) = \max\{|I| : I \in \mathcal{P}(1, 2, \ldots, n), a_I \neq 0\}.$$

If $\deg(F) \leq 1$, we call F an affine function; if, in addition, F satisfies $F(0) = 0$, we say that F is a linear function. Similarly, functions F with $\deg(F) = 2$ and $\deg(F) = 3$ are called quadratic and cubic, respectively, and so forth.

In the particular case of $n = m$, (n, n)-functions can also be represented as polynomials in a single variable over the finite field $\mathbb{F}_{2^n}$, in the form

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

with $a_i \in \mathbb{F}_{2^n}$ for $i = 0, 1, \ldots, 2^n - 1$. This is referred to as the univariate representation of F, and is also unique.

For $a \in \mathbb{F}_2^n$, and a VBF F, define the discrete derivative in direction a by

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

We say that F is *APN* if, for every $a \in \mathbb{F}_2^n \setminus \{0\}$ and every $b \in \mathbb{F}_2^m$, the equation

$$D_a F(x) = b$$

has at most two solutions. More generally, the differential uniformity of F is defined as the maximum number of solutions to the above equation for any $a \in \mathbb{F}_2^n \setminus \{0\}$ and any $b \in \mathbb{F}_2^m$. APN functions are thus precisely the functions having differential uniformity equal to 2. Since a low differential uniformity indicates a good resistance to differential cryptanalysis, APN functions have attracted significant research interest, and constitute one of the main use cases for algorithms for testing equivalence.

2.2 Linear algebra

An *affine subspace* of $\mathbb{F}_2^d$ is a coset $u + U$, where $U \subseteq \mathbb{F}_2^d$ is a linear subspace and $u \in \mathbb{F}_2^d$. If $\dim U = k$, we call $u + U$ an *affine subspace of dimension* k. We write

$$\mathrm{AGr}(k, d) := \{u + U : U \leq \mathbb{F}_2^d,\ \dim U = k,\ u \in \mathbb{F}_2^d\}$$

for the set of all affine subspaces of dimension k.

For a nonempty subset $S \subseteq \mathbb{F}_2^d$, we denote by $\mathrm{Aff}(S)$ the *affine span* of S, i.e. the smallest affine subspace of $\mathbb{F}_2^d$ containing S.

A special type of affine subspaces are the so-called hyperplanes. A hyperplane of $\mathbb{F}_2^d$ is a set of the form $L_{\alpha,c} = \{x \in \mathbb{F}_2^d \mid \alpha \cdot x = c\}$, where $\alpha \in \mathbb{F}_2^d \setminus \{0\}$ and "$\cdot$" denotes the dot product on $\mathbb{F}_2^d$. The element α is then called the *defining element* of the hyperplane. Clearly, $L_{\alpha,c} \in \mathrm{AGr}(d-1, d)$ for any choice of α and c. Furthermore, if $c = 0$, $L_{\alpha,0}$ is a linear subspace, and if $c = 1$, $L_{\alpha,1}$ is an affine (but not linear) subspace of $\mathbb{F}_2^d$. We will denote the set of all hyperplanes of $\mathbb{F}_2^d$ by $\mathrm{Hyp}(\mathbb{F}_2^d) = \mathrm{AGr}(d-1, d)$.

Partial maps

A partial map A from $\mathbb{F}_2^d$ to $\mathbb{F}_2^d$ is a mapping $A : D \to \mathbb{F}_2^d$, where D is an affine subspace of $\mathbb{F}_2^d$ (we note that since the mappings being reconstructed are affine functions, the set of inputs where their outputs are known will always be an affine subspace of $\mathbb{F}_2^d$). We will call D the domain of the partial map A and denote it by $\mathrm{Domain}(A)$.

Let $A : D \subseteq \mathbb{F}_2^{n+m} \to \mathbb{F}_2^{n+m}$ be a partial affine map and F and G be two given (n,m)-functions. We call an affine permutation $\widehat{A} : \mathbb{F}_2^{n+m} \to \mathbb{F}_2^{n+m}$ an *extension* of A if $\widehat{A}(x) = A(x)$ for all $x \in D$ and $\widehat{A}(\Gamma_F) = \Gamma_G$.

2.3 Group theory

We use standard notation from group theory. Given a group $(\mathcal{G}, \circ)$ and a set of elements $S \subseteq \mathcal{G}$, we write $\langle S \rangle$ for the subgroup generated by S. For groups $H, \mathcal{G}$ we write $H \leq \mathcal{G}$ to denote that H is a subgroup of $\mathcal{G}$. For any set Ω, let $\mathrm{Sym}(\Omega)$ denote the symmetric group on Ω, i.e. the group of all bijections from Ω to itself.

If a group $\mathcal{G}$ acts on a finite set Ω, then for $p \in \Omega$ we denote by $\mathrm{Orb}_H(p) = \{g(p) : g \in H\}$ the orbit of p; the orbits form the so-called *orbit partition* which is a partition of Ω into equivalence classes, we denote this as Ω/H. For a point $p \in \Omega$ we write

$$\mathrm{Stab}_p(\mathcal{G}) = \{g \in \mathcal{G} : g(p) = p\}.$$

for the stabilizer of p, which is the set of elements that fix p.

If A is a partial map on $D \subseteq \mathbb{F}_2^{n+m}$, we write

$$\mathrm{Stab}_A(H) = \{g \in H : g(q) = q \text{ for all } q \in A(D)\}.$$

So $\mathrm{Stab}_A(H)$ is the point-wise stabilizer of all the points in the image of the domain of A.

We maintain such groups via a base and strong generating set (BSGS) and Schreier-Sims algorithm; see e.g. [19].

Induced action on the graph.

Let $A : \mathbb{F}_2^{n+m} \to \mathbb{F}_2^{n+m}$ be a CCZ-mapping for some (n,m)-functions F and G, i.e. an affine permutation with $A(\Gamma_F) = \Gamma_G$. Since A maps each point $(x, F(x))$ of Γ_F to a unique point $(x', G(x'))$ of Γ_G, it induces a bijection $\pi(A) : \Gamma_F \to \Gamma_G$ defined by

$$\pi(A)(x, F(x)) = (x', G(x')) \quad \text{where} \quad A(x, F(x)) = (x', G(x')).$$

If A is a partial affine map with domain D, an immediate generalization of the above definition gives a partial map $\pi(A) : D \cap \Gamma_F \to \Gamma_G$.

We write $\mathrm{Aut}(F) \leq \mathrm{Sym}(\mathbb{F}_2^{n+m})$[1] for the automorphism group of F, i.e. the group of affine permutations $A : \mathbb{F}_2^{n+m} \to \mathbb{F}_2^{n+m}$ such that $A(\Gamma_F) = \Gamma_F$. Any $A \in \mathrm{Aut}(F)$ stabilizes Γ_F, so restricting to Γ_F gives a group homomorphism

$$\pi : \mathrm{Aut}(F) \to \mathrm{Sym}(\Gamma_F), \qquad \pi(A)(p) = A(p) \quad (p \in \Gamma_F).$$

For a subgroup $H \leq \mathrm{Aut}(F)$ we write $Q := \pi(H) \leq \mathrm{Sym}(\Gamma_F)$ for the induced permutation group and $\ker(\pi|_H)$ for its kernel. Also let $\mathrm{Aut}_{EA}(F)$ be the automorphism

[1] Since the elements of $\mathrm{Aut}(F)$ are affine permutations of $\mathbb{F}_2^{n+m}$, one could equivalently view $\mathrm{Aut}(F)$ as a subgroup of the affine general linear group $\mathrm{AGL}(n+m, 2)$. We use $\mathrm{Sym}(\mathbb{F}_2^{n+m})$ only as a convenient ambient permutation group.

group of F restricted to EA-mappings. Denote $\pi(\mathrm{Aut}(F))$ by $Aut^*(F)$. If $H \leq Aut^*(F)$ then we let $Stab_A(H) = \{g \in H : g(q) = q \text{ for all } q \in A(D \cap \Gamma_F)\}$. For an element $g \in Aut^*(F)$ we say that $\bar{g} \in Aut(F)$ is a *lift* if g and $\bar{g}$ agree on Γ_F, such a lifting is always possible by the definition of $Aut^*(F)$. Finally let $1_{\Gamma_F} \in \mathrm{Sym}(\Gamma_F)$ be the identity mapping.

2.4 Ordered partitions

Our algorithms operate by exploiting the structure of certain partitions that remain invariant under CCZ- and EA-equivalence. Furthermore, the order of the elements in these partitions is preserved by the equivalence relations as well, which motivates the following definition.

For any set S, an *ordered partition*[2] of S is an ordered k-tuple

$$\Phi = (B_1, B_2, \ldots, B_k)$$

for some positive integer k that satisfies the following conditions:

1. *pairwise disjointness:* $B_i \cap B_j = \emptyset$;
2. *cells in universe* for all i, $B_i \subseteq S$.

The partitions are ordered, so two partitions are considered distinct if they have the same blocks but in different order. We say that two partitions $\Phi = (B_1, \ldots, B_k)$ and $\Phi' = (C_1, \ldots, C_k)$ have the same *shape* if $|B_i| = |C_i|$ for all i, and say that B_i and C_i are *corresponding cells*. We denote the singleton cells of an ordered partition Φ by

$$I(\Phi) = \{P \in \Phi : |P| = 1\}.$$

An ordered partition $\widehat{\Phi}$ is called a *refinement* of an ordered partition Φ (of the same set S) if every cell of $\widehat{\Phi}$ is contained in some cell of Φ, i.e.

$$\widehat{P} \in \widehat{\Phi} \implies \exists\, P \in \Phi \text{ with } \widehat{P} \subseteq P.$$

If for some $\widehat{P} \in \widehat{\Phi}$ we have a strict containment $\widehat{P} \subsetneq P$, we say that the cell P has been *split*. Finally, we will use the shorthand notation $\bigcup \Phi = \bigcup_i B_i$ when $\Phi = (B_1, \ldots, B_k)$ is an ordered partition.

3 The partition refinement algorithm

The partition refinement framework was introduced in [20] for the graph isomorphism problem, but has now been adapted for various other problems including code isomorphism [17]. The basic structure of the algorithm is a backtracking search where each search node has one or more ordered partitions (see Section 2.4), which restricts the possible extensions of the current partial isomorphism.

[2] In much of the literature this would be called an ordered partial partition, since we do not require the cells to cover S. For our purposes we simply call it an ordered partition.

In our case the isomorphism we are looking for is the affine permutation $A : \mathbb{F}_2^{n+m} \to \mathbb{F}_2^{n+m}$ satisfying $A(\Gamma_F) = \Gamma_G$, where $F, G : \mathbb{F}_2^n \to \mathbb{F}_2^m$ are VBFs. Our algorithm will branch by picking some so-far unmapped point $p \in \Gamma_F$ and mapping it to some point $q \in \Gamma_G$. The goal is then to reduce the number of such guesses we have to make and thus reduce the size of the search tree. Throughout this section and the rest of this paper F and G will denote some VBFs $F, G : \mathbb{F}_2^n \to \mathbb{F}_2^m$.

Our algorithm will maintain the following data in each search node.

- A partial affine map $A : D \subseteq \mathbb{F}_2^{n+m} \to \mathbb{F}_2^{n+m}$.
- A pair of ordered partitions (Π_F, Π_G), where $\Pi_F = (P_1^F, \ldots, P_k^F)$ is an ordered partition of Γ_F and $\Pi_G = (P_1^G, \ldots, P_k^G)$ is an ordered partition of Γ_G. We refer to (Π_F, Π_G) as the *point partitions*.
- A pair of ordered partitions (Λ_F, Λ_G), where $\Lambda_F = (S_1^F, \ldots, S_l^F)$ and $\Lambda_G = (S_1^G, \ldots, S_l^G)$ are ordered partitions of $\mathrm{Hyp}(\mathbb{F}_2^{n+m})$. We refer to (Λ_F, Λ_G) as the *hyperplane partitions*.

For these objects the following invariants will be maintained.

Invariant property 1 Any extension $\widehat{A}$ of the partial affine map A satisfies $\widehat{A}(P_i^F) = P_i^G$ for all $i \in [k]$.

Invariant property 2 Any extension $\widehat{A}$ of the partial affine map A satisfies $\widehat{A}(S_i^F) = S_i^G$ for all $i \in [l]$.

Invariant property 1 allows us to reduce the number of guesses we have to make for a point $p \in P_i^F$, since it can only be mapped to a point q in the corresponding cell P_i^G. We would then like to make the cells as small as possible to reduce the number of guesses. To achieve this, we repeatedly refine the point partition pair (Π_F, Π_G) (see Section 2.4), while maintaining Invariant property 1. Informally the basic structure of the algorithm is as follows:

BacktrackSearch$(A, (\Pi_F, \Pi_G))$

```
Refine(Π_F, Π_G)
Let P_i^F be the smallest cell of Π_F by cardinality
Let x be any element of P_i^F \ Domain(A)
for y ∈ P_i^G do
    A(x) ← y
    BacktrackSearch(A, (Π_F, Π_G))
```

The reason for maintaining Invariant property 2 is so that we can use it to perform the $\textsc{Refine}(\Pi_F, \Pi_G)$ step. We will also refine the partitions Λ_F and Λ_G, because as they become more refined, they better help us refine Π_F and Π_G. Since the point partitions are the partitions we branch on, the sizes of these partitions determine the size of the search tree. The algorithm also uses the automorphism group of G for the branching step, which we will discuss further in Section 5.1.

3.1 Signatures

We now describe the main step of the algorithm, i.e. partition refinement. The basic idea is that we take all the cells in the point partitions or the hyperplane partitions and try to split them. We do this by defining signatures of points and hyperplanes. Let Σ be any ordered finite set of elements. We refer to a mapping $\text{sig} : \Gamma_F \to \Sigma$ as a *point signature* and to a mapping $\text{sig} : \text{Hyp}(\mathbb{F}_2^{n+m}) \to \Sigma$ as a *hyperplane signature*.

Valid signatures. Let A be a partial affine map. We say that the pair of signatures $(\text{sig}_F, \text{sig}_G)$ is a *valid point signature pair* if, for every extension $\widehat{A}$ of A, we have

$$\widehat{A}(p) = q \quad \Longrightarrow \quad \text{sig}_F(p) = \text{sig}_G(q)$$

for all $p \in \Gamma_F, q \in \Gamma_G$.

Similarly, we say that $(\text{sig}_F, \text{sig}_G)$ is a *valid hyperplane signature pair* if, for every extension $\widehat{A}$ of A, we have

$$\widehat{A}(U) = V \quad \Longrightarrow \quad \text{sig}_F(U) = \text{sig}_G(V),$$

for all hyperplanes $U \in \text{Hyp}(\mathbb{F}_2^{n+m})$ and $V \in \text{Hyp}(\mathbb{F}_2^{n+m})$.

The purpose of these signatures is to refine the point partitions and hyperplane partitions. Let $\Phi = (B_1, \ldots, B_k)$ be an ordered partition of a set Ω (which can be either a point partition or a hyperplane partition), and let $\Sigma = \{\sigma_1, ..., \sigma_t\}$ be any ordered set so that $\sigma_i < \sigma_{i+1}$ for all i. The refinement of Φ induced by the signature $\text{sig} : \Omega \to \Sigma$ is the ordered partition

$$\text{Refine}(\Phi, \text{sig}) := \big(B_{i,\sigma_j} \;\big|\; 1 \le i \le k,\ 1 \le j \le t,\ B_{i,\sigma_j} \neq \varnothing \big),$$

where we list the nonempty cells in lexicographic order by (i, j) (i.e., $B_{1,\sigma_1}, \ldots, B_{1,\sigma_t}, B_{2,\sigma_1}, \ldots$), omitting any empty cells. For each i, the cells $B_{i,\sigma_1}, \ldots, B_{i,\sigma_t}$ are exactly the subsets of B_i consisting of elements with the same signature value. We thus keep the order of the original blocks B_i, but subdivide them into smaller sets. Formally

$$\begin{aligned} a \in B_{i,\sigma_j} \iff & \ a \in B_i \text{ and } \text{sig}(a) = \sigma_j, \\ & B_{i,\sigma_j} \subseteq B_i. \end{aligned} \tag{2}$$

Given a pair of ordered partitions $(\Phi_F = (B_1^F, \ldots, B_a^F), \Phi_G = (B_1^G, \ldots, B_b^G))$ and a pair of signatures $(\text{sig}_F, \text{sig}_G)$, we say that Φ_F and Φ_G have the same *shape* relative to (sig_F, sig_G) if $a = b$ and for each $i \in [a]$

$$\forall x \in B_i^F, \forall y \in B_i^G \ \ sig_F(x) = sig_G(y)$$

and

$$|B_i^F| = |B_i^G|.$$

This then guarantees that performing refinement by valid signatures will maintain Invariant property 1 and Invariant property 2. This is formalized in the following two lemmas.

Lemma 1 *Let A be a partial affine map, and let (Π_F, Π_G) satisfy Invariant property 1 (with respect to a partial map A). Let $(\mathrm{sig}_F, \mathrm{sig}_G)$ be a valid point signature pair. Compute the refinements*

$$\Pi'_F := \mathrm{Refine}(\Pi_F, \mathrm{sig}_F), \qquad \Pi'_G := \mathrm{Refine}(\Pi_G, \mathrm{sig}_G).$$

Then (Π'_F, Π'_G) also satisfies Invariant property 1 with respect to A if (Π'_F, Π'_G) have the same shape relative to (sig_F, sig_G).

Proof Let $\widehat{A}$ be any extension of A, let $p \in \Gamma_F$ and let $\widehat{A}(p) = q$. We must then show that p and q are in corresponding cells in Π'_F and Π'_G respectively. Since (Π_F, Π_G) satisfies Invariant property 1, p and q must lie in corresponding cells with respect to Π_F and Π_G. Then since (sig_F, sig_G) is a valid point signature pair we must have $sig_F(p) = sig_G(q)$, so we see by (2) that p and q must be in corresponding cells with respect to (Π'_F, Π'_G) since (Π'_F, Π'_G) have the same shape relative to (sig_F, sig_G). □

The same argument applies to hyperplane signatures. The proof is analogous to that of Lemma 1.

Lemma 2 *Let A be a partial affine map, and let (Λ_F, Λ_G) satisfy Invariant property 2 (with respect to A). Let $(\mathrm{sig}_F, \mathrm{sig}_G)$ be a valid hyperplane signature pair. Let*

$$\Lambda'_F := \mathrm{Refine}(\Lambda_F, \mathrm{sig}_F), \qquad \Lambda'_G := \mathrm{Refine}(\Lambda_G, \mathrm{sig}_G).$$

Then (Λ'_F, Λ'_G) also satisfies Invariant property 2 relative to A if they have the same shape relative to (sig_F, sig_G).

3.1.1 Point signatures

We now discuss the concrete signatures that we use throughout our algorithm. Let us first consider how we define the signatures on the points. Fix any ordered hyperplane partition $\Lambda = (S_1, \ldots, S_l)$. For a point $p \in \Gamma_F$, define its signature as the vector of incidence counts

$$\mathrm{sig}_\Lambda(p) := \big(|\{U \in S_1 : p \in U\}|, \ldots, |\{U \in S_l : p \in U\}| \big).$$

In other words, the i-th coordinate of the signature records how many hyperplanes in the cell S_i contain p.

The following lemma shows that we can obtain a valid point signature pair from any pair of hyperplane partitions satisfying Invariant property 2.

Lemma 3 *If (Λ_F, Λ_G) is a pair of hyperplane partitions satisfying Invariant property 2 for the partial affine map A, then $(sig_{\Lambda_F}, sig_{\Lambda_G})$ is a valid point signature pair.*

Proof Let $\widehat{A}$ be any extension of A, and let $\widehat{A}(p) = q$. Fix $i \in \{1, \ldots, l\}$. By Invariant property 2 the extension $\widehat{A}$ maps the i-th block of the hyperplane partition to a corresponding block, i.e. $\widehat{A}(S_i^F) = S_i^G$.

Consider the map $U \mapsto \widehat{A}(U)$, which is a bijection from S_i^F to S_i^G. Clearly, for any two points p and $q = \widehat{A}(p)$,

$$p \in U \quad \iff \quad q = \widehat{A}(p) \in \widehat{A}(U).$$

Therefore the number of hyperplanes in S_i^F containing p equals the number of hyperplanes in S_i^G containing q, i.e.

$$|\{U \in S_i^F : p \in U\}| = |\{V \in S_i^G : q \in V\}|.$$

Since this holds for each i, we get $\mathrm{sig}_{\Lambda_F}(p) = \mathrm{sig}_{\Lambda_G}(q)$, so $(\mathrm{sig}_{\Lambda_F}, \mathrm{sig}_{\Lambda_G})$ is valid. □

3.1.2 Hyperplane signatures

We now discuss how we define our hyperplane signatures. For any affine hyperplane $U \in \mathrm{Hyp}(\mathbb{F}_2^{n+m})$ and a point partition $\Pi = (P_1, ..., P_k)$ we define the signature of U as

$$sig_{\Pi}(U) = (|U \cap P_1|, |U \cap P_2|, \ldots, |U \cap P_k|).$$

Thus, the signature of the hyperplane is the intersection count with the point cells.

The following lemma establishes the validity of the hyperplane signature.

Lemma 4 *If (Π_F, Π_G) is a point partition pair satisfying Invariant property 1 for the partial affine map A, then $(\mathrm{sig}_{\Pi_F}, \mathrm{sig}_{\Pi_G})$ is a valid hyperplane signature pair.*

Proof Let $\widehat{A}$ be any extension of A. Let $U \in \mathrm{Hyp}(\mathbb{F}_2^{n+m})$ and assume $\widehat{A}(U) = V$. Fix $i \in \{1, \ldots, k\}$. By Invariant property 1 we have $\widehat{A}(P_i^F) = P_i^G$. Using that $\widehat{A}$ is a permutation, we compute

$$|V \cap P_i^G| = |\widehat{A}(U) \cap \widehat{A}(P_i^F)| = |\widehat{A}(U \cap P_i^F)| = |U \cap P_i^F|.$$

Thus the i-th coordinates of $\mathrm{sig}_{\Pi_F}(U)$ and $\mathrm{sig}_{\Pi_G}(V)$ agree for every i, so $\mathrm{sig}_{\Pi_F}(U) = \mathrm{sig}_{\Pi_G}(V)$. This proves that $(\mathrm{sig}_{\Pi_F}, \mathrm{sig}_{\Pi_G})$ is valid. □

3.2 Refining with signatures

We will discuss how to compute signatures efficiently in more detail in Section 4. The point and hyperplane signatures can be used to perform refinement as follows.

From hyperplanes to points. Given a hyperplane partition pair (Λ_F, Λ_G), we use the induced point signatures sig_{Λ_F} and sig_{Λ_G} to compute $\mathrm{Refine}(\Pi_F, \mathrm{sig}_{\Lambda_F})$ and $\mathrm{Refine}(\Pi_G, \mathrm{sig}_{\Lambda_G})$.

From points to hyperplanes. Given a point partition pair (Π_F, Π_G), we use the induced hyperplane signatures sig_{Π_F} and sig_{Π_G} to compute $\mathrm{Refine}(\Lambda_F, \mathrm{sig}_{\Pi_F})$ and $\mathrm{Refine}(\Lambda_G, \mathrm{sig}_{\Pi_G})$.

We repeat these two steps until neither (Π_F, Π_G) nor (Λ_F, Λ_G) can be split further by refinement, or until the shapes of the corresponding ordered partitions no longer match.
The first step in the refinement process is to initialize the partition pairs (Π_F, Π_G) and (Λ_F, Λ_G). For the point partitions, we start with the trivial (one-block) partitions

$$\Pi_F = (\Gamma_F,), \qquad \Pi_G = (\Gamma_G,).$$

Then clearly Invariant property 1 is maintained since any extension $\widehat{A}$ of the empty partial map must necessarily satisfy $\widehat{A}(\Gamma_F) = \Gamma_G$ by definition.

For the hyperplane partitions, we simply set

$$\Lambda_F = \Lambda_G = (\mathrm{Hyp}(\mathbb{F}_2^{n+m}),),$$

which clearly satisfies Invariant property 2 since an affine map maps an affine hyperplane to another affine hyperplane.

4 Computing signatures

In the previous section we defined signatures for hyperplanes and points, and we discussed how these can be used in the refinement process. The question is then, how to compute these signatures efficiently. A naive implementation directly from the definitions is, of course, possible, but would be rather inefficient. For example, point signatures could be computed as follows:

Naive computation of point signatures

Input: Point set Γ_F and hyperplane partition $\Lambda_F = (S_1, \ldots, S_\ell)$
Output: Signatures $\mathrm{sig}_{\Lambda_F}(p) \in \mathbb{Z}^\ell$ for all $p \in \Gamma_F$
foreach $p \in \Gamma_F$ **do**
  for $i \leftarrow 1$ **to** ℓ **do**
    $\mathrm{sig}_{\Lambda_F}(p)_i \leftarrow 0$
    foreach $U \in S_i$ **do**
      if $p \in U$ **then**
        $\mathrm{sig}_{\Lambda_F}(p)_i \leftarrow \mathrm{sig}_{\Lambda_F}(p)_i + 1$

The complexity of this approach is approximately $\mathcal{O}(|\Gamma_F| \cdot |\mathrm{Hyp}(\mathbb{F}_2^{n+m})|) = \mathcal{O}(2^{2n+m})$ ignoring polynomial factors. We get the same complexity for computing hyperplane signatures naively. However, we can do much better.

For a function $f : \mathbb{F}_2^d \to \mathbb{Z}$ the Fourier-Hadamard transform is defined by

$$\widehat{f}(u) = \sum_{x \in \mathbb{F}_2^d} f(x)(-1)^{u \cdot x}.$$

It is well known that one can compute a lookup table of all values of $\widehat{f}$ in time $\mathcal{O}(d2^d)$ using the Fast Fourier-Hadamard transform (FFHT), see for example [21]. We now discuss how the Fourier-Hadamard transform can be used to perform the refinement procedure more efficiently.

4.1 Point refinement

Suppose that we are given ordered partitions of points and hyperplanes $\Pi = (P_1, \ldots, P_k)$ and $\Lambda = (S_1, \ldots, S_l)$. While ideally we would like to compute the full point signature

$$\mathrm{sig}_\Lambda(p) := \big(|\{U \in S_1 : p \in U\}|, \ldots, |\{U \in S_l : p \in U\}|\big),$$

as explained above this is quite expensive. Instead, we will compute a weighted version. Let $K = (s_1, s_2, \ldots, s_l)$ be a tuple of integers, one for each cell of Λ. Then we define a weight function $h_{K,\Lambda} : \mathrm{Hyp}(\mathbb{F}_2^{n+m}) \to \mathbb{Z}$ by

$$h_{K,\Lambda}(U) = \begin{cases} s_i, & \text{if } U \in S_i \text{ for some } i \in [l], \\ 0, & \text{if } U \notin \bigcup \Lambda. \end{cases}$$

The signature of a point p induced by the weight function $h_{K,\Lambda}$ is defined as

$$\mathrm{sig}_{h_{K,\Lambda}}(p) = \sum_{i=1}^{l} s_i \cdot |\{U \in S_i : p \in U\}|.$$

This weighted version of the signature is weaker than the original (so potentially less distinguishing for points) as it is a scalar value instead of the full incidence vector of $sig_\Lambda(p)$. However, it can still be used to refine partitions in practice, at a much more efficient computational cost than the full signature. The next lemma follows from the simple fact that

$$\mathrm{sig}_{\Lambda_F}(p) = \mathrm{sig}_{\Lambda_G}(q) \implies \mathrm{sig}_{h_{K,\Lambda_F}}(p) = \mathrm{sig}_{h_{K,\Lambda_G}}(q).$$

Lemma 5 *Let (Λ_F, Λ_G) be a hyperplane partition pair satisfying Invariant property 2 for the current partial affine map A, where $\Lambda_F = (S_1^F, \ldots, S_l^F)$ and $\Lambda_G = (S_1^G, \ldots, S_l^G)$. Let $K = (s_1, \ldots, s_l)$ be a tuple of integers. Then $\big(\mathrm{sig}_{h_{K,\Lambda_F}}, \mathrm{sig}_{h_{K,\Lambda_G}}\big)$ is a valid point signature pair.*

The advantage of this weighted signature is that it can be computed more efficiently using the FFHT. We now show how to do this. For any $\alpha \in \mathbb{F}_2^{n+m} \backslash \{0\}$ and any weight function $h_{K,\Lambda}$, define the difference function

$$d_{h_{K,\Lambda}}(\alpha) = h_{K,\Lambda}(L_{\alpha,0}) - h_{K,\Lambda}(L_{\alpha,1}), \tag{3}$$

and let $d_{h_{K,\Lambda}}(0) = 0$. We have the following connection between this function and the signature induced by the weight function.

Lemma 6 *Let K be a tuple of integers and Λ a hyperplane partition. Then for the weight function $h_{K,\Lambda}$ we have*

$$\widehat{d}_{h_{K,\Lambda}}(p) \;=\; 2 \cdot \mathrm{sig}_{h_{K,\Lambda}}(p) - C_{K,\Lambda},$$

where

$$C_{K,\Lambda} \;=\; \sum_{\alpha \in \mathbb{F}_2^{n+m} \setminus \{0\}} \big(h_{K,\Lambda}(L_{\alpha,0}) + h_{K,\Lambda}(L_{\alpha,1})\big).$$

Proof Fixing any $p \in \mathbb{F}_2^{n+m}$, we have

$$\widehat{d}_{h_{K,\Lambda}}(p) \;=\; \sum_{\alpha \in \mathbb{F}_2^{n+m} \setminus \{0\}} d_{h_{K,\Lambda}}(\alpha)(-1)^{\alpha \cdot p}.$$

Furthermore,

$$\begin{aligned} d_{h_{K,\Lambda}}(\alpha)(-1)^{\alpha \cdot p} &= \big(h_{K,\Lambda}(L_{\alpha,0}) - h_{K,\Lambda}(L_{\alpha,1})\big)(-1)^{\alpha \cdot p} \\ &= 2h_{K,\Lambda}\big(L_{\alpha,\alpha \cdot p}\big) - \big(h_{K,\Lambda}(L_{\alpha,0}) + h_{K,\Lambda}(L_{\alpha,1})\big). \end{aligned}$$

so we get

$$\begin{aligned} \widehat{d}_{h_{K,\Lambda}}(p) &= \sum_{\alpha \in \mathbb{F}_2^{n+m} \setminus \{0\}} d_{h_{K,\Lambda}}(\alpha)(-1)^{\alpha \cdot p} \\ &= \sum_{\alpha \in \mathbb{F}_2^{n+m} \setminus \{0\}} \Big(2h_{K,\Lambda}\big(L_{\alpha,\alpha \cdot p}\big) - \big(h_{K,\Lambda}(L_{\alpha,0}) + h_{K,\Lambda}(L_{\alpha,1})\big)\Big) \\ &= 2 \cdot \left(\sum_{\alpha \in \mathbb{F}_2^{n+m} \setminus \{0\}} h_{K,\Lambda}\big(L_{\alpha,\alpha \cdot p}\big) \right) - C_{K,\Lambda}. \end{aligned}$$

For each nonzero α, the hyperplane $L_{\alpha,\alpha \cdot p}$ is the unique affine hyperplane with defining element α that contains p. Hence

$$\sum_{\alpha \neq 0} h_{K,\Lambda}\big(L_{\alpha,\alpha \cdot p}\big) \;=\; \sum_{U \colon p \in U} h_{K,\Lambda}(U) \;=\; \mathrm{sig}_{h_{K,\Lambda}}(p).$$

Substituting this into the previous derivation gives

$$\widehat{d}_{h_{K,\Lambda}}(p) = 2 \cdot \sum_{\alpha \neq 0} h_{K,\Lambda}\big(L_{\alpha,\alpha \cdot p}\big) - C_{K,\Lambda} = 2 \cdot \mathrm{sig}_{h_{K,\Lambda}}(p) - C_{K,\Lambda},$$

as claimed. □

The overhead for considering all the hyperplane cells in (Λ_F, Λ_G) is quite large, since we have to perform some work for every hyperplane that we include in the computations. Therefore, as we go deeper in the search tree, we try to discard the largest cells as soon as possible. Thus, we can simply delete some arbitrary large cells from Λ_F and their corresponding cells from Λ_G. This means that we will no longer have a full partition, but will still maintain the invariant property. The complexity of this remains the same

regardless of how many hyperplanes remain if we use the FFHT. So at some point it becomes faster to compute the signatures directly, i.e. directly computing the sum.

$$\begin{aligned}\widehat{d}_{h_{K,\Lambda}}(p) &= \sum_{\alpha\in\mathbb{F}_2^{n+m}\setminus\{0\}} d_{h_{K,\Lambda}}(\alpha)\,(-1)^{\alpha\cdot p} \\ &= \sum_{\alpha\in\mathbb{F}_2^{n+m}\setminus\{0\}} h_{K,\Lambda}(L_{\alpha,0})\,(-1)^{\alpha\cdot p} - \sum_{\alpha\in\mathbb{F}_2^{n+m}\setminus\{0\}} h_{K,\Lambda}(L_{\alpha,1})\,(-1)^{\alpha\cdot p}\end{aligned}$$

by going through all the hyperplanes in Λ and either adding or subtracting the weight to get the final sum. This is done for all points in time $\mathcal{O}(2^n \cdot |\bigcup \Lambda|)$. Thus, when the number of hyperplanes in $\bigcup \Lambda$ is less than around $(n+m)2^m$, it becomes more efficient to compute the point signatures without using the FFHT. We denote the maximum number of hyperplanes in the hyperplane partitions before we start doing direct summation as `DirectThreshold`. The full algorithm for refining the points is then

Algorithm 1: RefinePoints(Π, Λ, K)

Input: Ordered point partition $\Pi = (P_1, \ldots, P_k)$, ordered hyperplane partition $\Lambda = (S_1, \ldots, S_l)$ and list of integers $K = (s_1, \ldots, s_l)$.
Output: (`refined`, Π') where `refined` is **true** if at least one cell has been split or **false** otherwise, and Π' is a refinement of Π.

if $|\bigcup_{i=1}^{l} S_i| >$ *`DirectThreshold`* **then**
  Compute $\widehat{d}_{h_{K,\Lambda}}(p) = \sum_\alpha d_{h_{K,\Lambda}}(\alpha)(-1)^{\alpha\cdot p}$ for all p using FFHT
else
  Compute $\widehat{d}_{h_{K,\Lambda}}(p)$ directly by summing over the hyperplanes in Λ.
$C_{K,\Lambda} \leftarrow \sum_{\alpha\in\mathbb{F}_2^{n+m}\setminus\{0\}} \big(h_{K,\Lambda}(L_{\alpha,0}) + h_{K,\Lambda}(L_{\alpha,1})\big)$
$\mathrm{sig}(p) \leftarrow \big(\widehat{d}_{h_{K,\Lambda}}(p) + C_{K,\Lambda}\big)/2$ for all p
$\Pi' \leftarrow ()$, `refined` $\leftarrow$ **false**
for $P \in \Pi$ **do**
  Compute the nonempty blocks $B_{t_1}, \ldots, B_{t_r}$ where $B_t := \{p \in P : \mathrm{sig}(p) = t\}$ and $t_i < t_{i+1}$
  if $r > 1$ **then**
    `refined` $\leftarrow$ **true**
  Append $B_{t_1}, \ldots, B_{t_r}$ to Π'
return (*`refined`*, Π')

The correctness of Algorithm 1 is justified in the following lemma, which clearly follows from the above discussion.

Lemma 7 *If Π is a point partition, Λ a hyperplane partition and K a list of integers, Algorithm 1 with inputs (Π, Λ, K) computes* Refine$(\Pi, sig_{h_{K,\Lambda}})$.

4.2 Hyperplane refinement

Now we discuss a similar procedure that can be applied to the hyperplane signatures. For any point partition $\Pi = (P_1, \ldots, P_k)$ and tuple of integers $K = (s_1, \ldots, s_k)$, we can define a weight function $w_{K,\Pi} : \Gamma_F \to \mathbb{Z}$ by

$$w_{K,\Pi}(p) := \begin{cases} s_i, & \text{if } p \in P_i \text{ for some } i \in [k], \\ 0, & \text{if } p \notin \bigcup \Pi. \end{cases}$$

For an affine hyperplane $L \in \mathrm{Hyp}(\mathbb{F}_2^{n+m})$, define its signature induced by $w_{K,\Pi}$ as

$$\mathrm{sig}_{w_{K,\Pi}}(L) := \sum_{x \in L \cap \Gamma_F} w_{K,\Pi}(x).$$

As with the point partitions, it is not difficult to observe that these weight functions can be used to obtain valid signature pairs.

Lemma 8 *Let (Π_F, Π_G) be a point partition pair satisfying Invariant property 1 for a partial affine map A, where $\Pi_F = (P_1^F, \ldots, P_k^F)$ and $\Pi_G = (P_1^G, \ldots, P_k^G)$. Let $K = (s_1, \ldots, s_k)$ be a tuple of integers. Then $\left(\mathrm{sig}_{w_{K,\Pi_F}}, \mathrm{sig}_{w_{K,\Pi_G}}\right)$ is a valid hyperplane signature pair.*

Proof Let $\widehat{A}$ be an extension of A and assume $\widehat{A}(U) = V$. By Invariant property 1, for each i we have $\widehat{A}(P_i^F) = P_i^G$, hence $\widehat{A}$ restricts to a bijection $U \cap P_i^F \to V \cap P_i^G$.

Then

$$\sum_{x \in U \cap P_i^F} w_{K,\Pi_F}(x) \;=\; s_i \,|U \cap P_i^F| \;=\; s_i \,|V \cap P_i^G| \;=\; \sum_{y \in V \cap P_i^G} w_{K,\Pi_G}(y).$$

Summing over i gives $\mathrm{sig}_{w_{K,\Pi_F}}(U) = \mathrm{sig}_{w_{K,\Pi_G}}(V)$, proving validity. □

Similarly to our approach for the point signatures from Lemma 6, we can compute these signatures using the FFHT using the following connection.

Lemma 9 *Let Π be a point partition of Γ_F and K a list of integers, then*

$$\mathrm{sig}_{w_{K,\Pi}}(L_{\alpha,c}) = \frac{S + (-1)^c \widehat{w_{K,\Pi}}(\alpha)}{2}$$

for any $\alpha \in \mathbb{F}_2^{n+m} \backslash \{0\}, c \in \mathbb{F}_2$ where $S = \sum_{p \in \Gamma_F} w_{K,\Pi}(p)$.

Proof Fix $\alpha \neq 0$. We see that

$$\mathrm{sig}_{w_{K,\Pi}}(L_{\alpha,0}) + \mathrm{sig}_{w_{K,\Pi}}(L_{\alpha,1}) = \sum_{p \in \Gamma_F} w_{K,\Pi}(p) = S.$$

We now have

$$\widehat{w_{K,\Pi}}(\alpha) = \sum_{p \in \Gamma_F} w_{K,\Pi}(p)(-1)^{\alpha \cdot p}$$

$$= \sum_{\substack{p\in\Gamma_F \\ \alpha\cdot p=0}} w_{K,\Pi}(p) - \sum_{\substack{p\in\Gamma_F \\ \alpha\cdot p=1}} w_{K,\Pi}(p)$$
$$= \mathrm{sig}_{w_{K,\Pi}}(L_{\alpha,0}) - \mathrm{sig}_{w_{K,\Pi}}(L_{\alpha,1}).$$

Solving the linear system

$$\begin{cases} \mathrm{sig}_{w_{K,\Pi}}(L_{\alpha,0}) + \mathrm{sig}_{w_{K,\Pi}}(L_{\alpha,1}) = S, \\ \mathrm{sig}_{w_{K,\Pi}}(L_{\alpha,0}) - \mathrm{sig}_{w_{K,\Pi}}(L_{\alpha,1}) = \widehat{w_{K,\Pi}}(\alpha), \end{cases}$$

gives $\mathrm{sig}_{w_{K,\Pi}}(L_{\alpha,0}) = \frac{S+\widehat{w_{K,\Pi}}(\alpha)}{2}$ and $\mathrm{sig}_{w_{K,\Pi}}(L_{\alpha,1}) = \frac{S-\widehat{w_{K,\Pi}}(\alpha)}{2}$. □

This means that the weighted signature is fully determined by the Fourier-Hadamard transform. The same way as in RefinePoints, if the number of hyperplanes in the hyperplane partition is below `DirectThreshold` we switch to just directly computing $\widehat{w}(\alpha) = \sum_{x\in\Gamma_F} w(x)(-1)^{\alpha\cdot x}$ for all relevant α . We note that we use the same threshold value for both the point and the hyperplane partitions.

The resulting refinement procedure is outlined in the following algorithm:

Algorithm 2: RefineHyperplanes(Π, Λ, K)

Input: Ordered point partition $\Pi = (P_1, \ldots, P_k)$, ordered hyperplane partition $\Lambda = (S_1, \ldots, S_l)$, and key vector $K = (s_1, \ldots, s_k)$

Output: $(\texttt{refined}, \Lambda')$ where `refined` is true if at least one cell has been split, and Λ' is a refinement of Λ

if $|\bigcup S_i| >$ *DirectThreshold* **then**
  Compute $\widehat{w_{K,\Pi}}(\alpha) = \sum_{x\in\mathbb{F}_2^{n+m}} w_{K,\Pi}(x)(-1)^{\alpha\cdot x}$ for all α using FFHT

else
  for $\alpha \in \{\alpha : L_{\alpha,0}$ *or* $L_{\alpha,1}$ *is contained in* $\bigcup\Lambda\}$ **do**
    Compute $\widehat{w_{K,\Pi}}(\alpha)$ directly.

Compute $S = \sum_{p\in\Gamma_F} w_{K,\Pi}(p)$
Compute $\mathrm{sig}(L_{\alpha,c}) \leftarrow \big(S + (-1)^c \widehat{w_{K,\Pi}}(\alpha)\big)/2$ for all $L_{\alpha,c} \in \bigcup\Lambda$
$\Lambda' \leftarrow ()$, `refined` $\leftarrow$ **false**
for $S_i \in \Lambda$ **do**
  Compute the nonempty blocks $B_{t_1}, \ldots, B_{t_r}$ where $B_t := \{L \in S_i : \mathrm{sig}(L) = t\}$ and $t_i < t_{i+1}$
  if $r > 1$ **then**
    `refined` $\leftarrow$ `true`
  Append $B_{t_1}, \ldots, B_{t_r}$ to Λ'
return (*refined*, Λ')

Once again, it is not difficult to justify the correctness of the above procedure.

Lemma 10 *If* Π *is a point partition,* Λ *a hyperplane partition and* K *a list of integers Algorithm 2 with input* (Π, Λ, K) *computes* $\mathrm{Refine}(\Lambda, \mathrm{sig}_{w_{K,\Pi}})$.

4.3 Refinement algorithm

We now describe the full back-and-forth refinement algorithm. We maintain a point partition pair (Π_F, Π_G) and a hyperplane partition pair (Λ_F, Λ_G). We alternate between refining the point partitions and refining the hyperplane partitions until no further occurs. We also discard some of the largest cells in the hyperplane partition pair, while making sure to keep enough hyperplanes to still be able to perform meaningful refinement. Denote the minimum number of hyperplanes we keep by `HyperplaneFloor`. Furthermore, for a signature sig and point cell P_i^F let $sig(P_i^F)$ be defined as the value of $sig(p)$ for any point $p \in P_i^F$, which is well defined when all the points have the same signature. We define the signature of a cell the same way for hyperplane partitions.

Putting everything together, we obtain the following general refinement procedure.

Algorithm 3: REFINE$\big((\Pi_F, \Pi_G), (\Lambda_F, \Lambda_G)\big)$

Input: Point partition pair (Π_F, Π_G) and hyperplane partition pair (Λ_F, Λ_G).
Output: $(\texttt{consistent}, (\Pi_F, \Pi_G), (\Lambda_F, \Lambda_G))$ where `consistent` is **false** if shapes mismatch relative to the signature at any point, otherwise it is **true**.

```
refining ← true
while refining do
    K_1 ← (s_1, ..., s_l) where s_i is a random integer and l = |Λ_F|
    (refined-points, Π_F) ← RefinePoints(Π_F, Λ_F, K_1)
    (_, Π_G) ← RefinePoints(Π_G, Λ_G, K_1)
    if (Π_F, Π_G) does not have the same shape relative to
     (sig_{h_{K_1,Λ_F}}, sig_{h_{K_1,Λ_G}}) then
        return false
    K_2 ← (s_1, ..., s_k) where s_i are random integers and k = |Π_F|
    (refined-hyperplanes, Λ_F) ← RefineHyperplanes(Π_F, Λ_F, K_2)
    (_, Λ_G) ← RefineHyperplanes(Π_G, Λ_G, K_2)
    if (Λ_F, Λ_G) does not have the same shape relative to
     (sig_{w_{K_2,Π_F}}, sig_{w_{K_2,Π_G}}) then
        return false
    refining ← refined-points ∨ refined-hyperplanes
Order the partitions Λ_F and Λ_G by cell size (increasing)
Λ'_F ← ()
Λ'_G ← ()
num-hyperplanes ← 0
i ← 1
while num-hyperplanes < HyperplaneFloor do
    Λ'_F ← (Λ'_F, S_i^F)
    Λ'_G ← (Λ'_G, S_i^G)
    num-hyperplanes ← num-hyperplanes + |S_i^F|
    i ← i + 1
return (true, (Π_F, Π_G), (Λ'_F, Λ'_G))
```

Remark 1 In the last part, where we sort the hyperplane partition pair, we first sort Λ_F and re-arrange the cells in Λ_G accordingly so that the indices of corresponding cells are preserved.

Remark 2 We just return false if the shapes mismatch at any point since in that case the concrete values of the partitions are not relevant for further computations.

Lemma 11 *If A is a partial map and the point partition pair (Π_F, Π_G) satisfies Invariant property 1 and the hyperplane partition pair (Λ_F, Λ_G) satisfies Invariant property 2 both relative to A, then if Algorithm 3 returns false on input $((\Pi_F, \Pi_G), (\Lambda_F, \Lambda_G))$, then there does*

not exist any extension of A. Conversely, if Algorithm 3 returns true and $(\Pi'_F, \Pi'_G), (\Lambda'_F, \Lambda'_G)$ *then these satisfy Invariant property 1 and Invariant property 2 respectively.*

Proof We first prove inductively that as long as Algorithm 3 does not return false that the invariant properties are maintained. The algorithm alternates between refining the points and the hyperplanes. Let us start with the points. By Lemma 7 Algorithm 1 computes $\Pi'_F = \text{Refine}(\Pi_F, sig_{h_{K_1,\Lambda_F}})$ and $\Pi'_G = \text{Refine}(\Pi_G, sig_{h_{K_1,\Lambda_G}})$. By Lemma 5 $(sig_{h_{K_1,\Lambda_F}}, sig_{h_{K_1,\Lambda_G}})$ is a valid point signature pair and so by Lemma 1 (Π'_F, Π'_G) satisfies Invariant property 1.

Now the algorithm refines the hyperplanes using Algorithm 2 which by Lemma 10 computes $\Lambda'_F = \text{Refine}(\Lambda_F, \text{sig}_{w_{K_2,\Pi_F}})$ and $\Lambda'_G = \text{Refine}(\Lambda_G, \text{sig}_{w_{K_2,\Pi_G}})$. By Lemma 8 $(\text{sig}_{w_{K_2,\Pi_F}}, \text{sig}_{w_{K_2,\Pi_G}})$ is a valid hyperplane signature pair and so by Lemma 2 (Λ'_F, Λ'_G) satisfies Invariant property 2.

Now for the second part, assume that the algorithm returns false due to mismatched shapes on the point partitions. The algorithm computes $\Pi'_F = \text{Refine}(\Pi_F, sig_{h_{K_1,\Lambda_F}})$ and $\Pi'_G = \text{Refine}(\Pi_G, sig_{h_{K_1,\Lambda_G}})$ and (Π'_F, Π'_G) does not have the same shape relative to $(sig_{h_{K_1,\Lambda_F}}, sig_{h_{K_1,\Lambda_G}})$ which is a valid point signature pair by Lemma 5. Therefore, for any extension $\widehat{A}$ of A we must have for all $p \in \Gamma_F$

$$\widehat{A}(p) = q \quad \Longrightarrow \quad sig_{h_{K_1,\Lambda_F}}(p) = sig_{h_{K_1,\Lambda_G}}(q)$$

but since (Π'_F, Π'_G) does not have the same shape relative to this signature pair, this is clearly not possible for all p, so no extension of A can exist.
Now if the algorithm returns false due to the hyperplane partitions, the algorithm computes $\Lambda'_F = \text{Refine}(\Lambda_F, \text{sig}_{w_{K_2,\Pi_F}})$ and $\text{Refine}(\Lambda_G, \text{sig}_{w_{K_2,\Pi_G}})$ and (Λ'_F, Λ'_G) does not have the same shape relative to $(\text{sig}_{w_{K_2,\Pi_F}}, \text{sig}_{w_{K_2,\Pi_G}})$ which is a valid hyperplane signature pair by Lemma 8. So for any extension $\widehat{A}$ of A we must have for all $U \in \text{Hyp}(\mathbb{F}_2^{n+m})$

$$\widehat{A}(U) = V \Longrightarrow \text{sig}_{w_{K_2,\Pi_F}}(U) = \text{sig}_{w_{K_2,\Pi_G}}(V)$$

but since Λ'_F and Λ_G does not have the same shape relative to this signature this is clearly impossible for all $U \in \bigcup \Lambda_F$, so no extension of A exists.

□

Example 1 Let us go through an example of what the ordered partitions look like at different depths of the search tree for a concrete pair of functions. Let $F(x) = x^3$ and $G(x) = x^5$ over $\mathbb{F}_{2^7}$. These functions are not CCZ-equivalent, so let us trace through the algorithm as it branches and see how the ordered partitions look like. Here we are not going to discard any hyperplane cells as we do in the implemented algorithm. We describe the first few steps in Figure 1.

depth d	points cell sizes (after refine)	hyperplanes cell sizes (after refine)
0	`[128]`	`[8128,16510,8128]`
1	`[1,127]`	`[4572,3556,16510,4572,3556]`
2	`[1,1,126]`	`[2016,2556,1540,2016,8320,8190,2016,2556, 1540,2016]`
3	—	L: `[3,20,44,69,128,239,270,363,1,1,17,17,28, 99,117,175,198,227,1,6,11,47,77,93,201,240, 336,408,7,15,37,61,143, ...]` R: `[5,17,47,67,113,251,317,319,4,17,18,34, 82,105,185,196,239,1,6,11,47,77,93,201,240, 336,408,7,15,37,61,143,...]`

Fig. 1 CCZ-equivalence refinement trace for $F(x) = x^3$ vs. $G(x) = x^5$ over $\mathbb{F}_{2^7}$, recording only the multisets of cell sizes (after refinement) up to depth 3.

At depths $0, 1$, and 2 the partitions have the same shape on both sides. Initially, at depth 0, we have three hyperplane cells but the point partition pair remains trivial. We then pick any source point in the (unique) point cell and branch on it. This gives point-cell sizes $[1, 127]$, where the singleton cell contains the chosen source point. After refinement, the point partition pair still does not split, although the hyperplane partition pair does, and we again pick a source point in the 127-point cell to branch on. The same repeats at depth 2 before we arrive at depth 3, where we obtain point-cell sizes $[1, 1, 1, 125]$. We then refine the hyperplane partition pair and immediately see that the shapes do not match, so this depth-3 search node aborts. The number of children for a search node is the size of the smallest non-singleton cell in the current point partition, and we see that in this example we could end up with a very large search tree. We will come back to this example in Section 5.1 to see how automorphism groups help us deal with this problem.

5 Automorphism groups

The basic operation of our algorithm has already been described in detail in Sections 3 and 4. In this section, we discuss how knowledge of the automorphism groups of the functions being tested for equivalence can be used to speed up the computations, and how these automorphism groups can be computed using our algorithmic framework.

5.1 Branching on orbits

If a function F has a large automorphism group, we can significantly reduce the number of guesses the algorithm has to make. In fact, we do not even need to know the full automorphism group in order to achieve a significant speedup.

Lemma 12 *Let $H \leq \mathrm{Aut}^*(G)$, $S_F \subseteq \Gamma_F$ and A a partial map where $\mathrm{Domain}(A) = \mathrm{Aff}(S_F)$ and $A(\mathrm{Domain}(A) \cap \Gamma_F) \subseteq \Gamma_G$. Fix $p \in \Gamma_F \backslash \mathrm{Domain}(A)$ and let $q, q' \in \Gamma_G$ be points in the same orbit under $\mathrm{Stab}_A(H)$. If there exists an extension $\widehat{A}$ of A with $\widehat{A}(p) = q$, then there also exists an extension $\widetilde{A}$ of A with $\widetilde{A}(p) = q'$. Furthermore, we have $\pi(\widehat{A}) = h \circ \pi(\widetilde{A})$ for some $h \in \mathrm{Stab}_A(H)$.*

Proof Let $\widehat{A}$ be an extension of A where $\widehat{A}(p) = q$. Assume $q \in \Gamma_G$ and $q' \in \Gamma_G$ lie in the same $\mathrm{Stab}_A(H)$-orbit, so there exists $g \in \mathrm{Stab}_A(H)$ with $g(q) = q'$. Now lift g to a map $\bar{g} \in \mathrm{Aut}(G)$. Let $\widetilde{A} = \bar{g} \circ \widehat{A}$. We see that

$$\widetilde{A}(p) = \bar{g} \circ \widehat{A}(p) = \bar{g}(q) = q'$$

So $\widetilde{A}(p) = q'$.

Now if $x \in \mathrm{Domain}(A)$ and $A(x) = y$ then

$$\begin{aligned} \widetilde{A}(x) &= \bar{g} \circ \widehat{A}(x) \\ &= \bar{g}(y). \end{aligned}$$

Since $S_F \subseteq \mathrm{Domain}(A) \cap \Gamma_F$ we see that $g \in \mathrm{Stab}_A(H)$ fixes every point of $A(S_F)$. Then since $\bar{g}$ is affine it fixes every point in the affine span $\mathrm{Aff}(A(S_F))$. Since $A(\mathrm{Domain}(A)) \subseteq \mathrm{Aff}(A(S_F))$ we see that $\bar{g}$ fixes y.

$$= y$$

So we got $\widetilde{A}(x) = A(x)$. Which means $\widetilde{A}$ is an extension of A.

Finally $\widehat{A} = \bar{g}^{-1} \circ \widetilde{A}$. Which clearly means that

$$\begin{aligned} \pi(\widehat{A}) &= \pi(\bar{g}^{-1} \circ \widetilde{A}) \\ &= \pi(\bar{g}^{-1}) \circ \pi(\widetilde{A}) \\ &= g^{-1} \circ \pi(\widetilde{A}) \end{aligned}$$

Letting $h = g^{-1}$ the final result follows. □

This means that for a fixed source point p instead of having to try all target points q in the corresponding point cell, we can just try one target point q per orbit in the corresponding cell. This can significantly reduce branching when the function has a nontrivial automorphism group. This is true for the majority of practically interesting functions, and in particular is conjectured to hold for all APN functions [22].

Example 2 Let us do the same kind of trace as in Example 1, but now illustrate how automorphism orbits can reduce the size of the search tree. We again test CCZ-equivalence between x^3 and x^5 over $\mathbb{F}_{2^7}$. The computations are illustrated in Figure 2

depth d		
0	points	`[128]`
	orbits	C_1: `[128]`
1	points	`[1,127]`
	orbits	C_1: `[1]`, C_2: `[127]`
2	points	`[1,1,126]`
	orbits	C_1: `[1]`, C_2: `[1]`, C_3: `[7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7]`

Fig. 2 Refinement trace with orbit information: the first column is the point-partition size multiset, and the last column records the orbit-size multiset inside the corresponding point partition

Here we record the size of the orbits inside the point partitions. This is done with respect to the orbits under the current stabilizer subgroup of the target function's automorphism group, so x^5 in this case. Starting at depth 0 we have only the trivial

point partition, but we see that all of the points are in fact in the same orbit since the orbit has size 128. This means that when we pick a source point p, any two images q and q' are "equivalent", so we can pick one of them arbitrarily and avoid any guessing. The same happens at depth 1. When we reach depth 2, the point cell of size 126 splits into 18 orbits of size 7. We again pick some source point in this cell and then branch on a representative for each of these 18 orbits, so this node would have 18 children. If we assume that all search nodes abort at depth 3 like in Example 1, we can estimate the number of nodes in the tree with and without using the orbits as follows:

- **with orbits:** in this case, we just have 18 leaf nodes, plus the single depth 0, depth 1 and depth 2 nodes, for a total of 21 nodes;
- **without orbits:** the depth 0 node has 128 children, the depth 1 nodes have 127 children and the depth 2 nodes have 126 children, for a total of $1 + 128 + 128 \cdot 127 + 128 \cdot 127 \cdot 126 = 2,064,641$ nodes.

We would like to do the same in our algorithm for computing automorphisms, but we cannot, of course, assume that we know the automorphism group at the start of the search. So in this case we will update the orbit partitions whenever the automorphism group gets updated.[3]

5.2 Representing the group

We could view $\mathrm{Aut}(F)$ as a subgroup of $Sym(\mathbb{F}_2^{n+m})$, but then the degree of the group would be 2^{n+m}, which would make our group operations quite expensive. We would instead want to consider it as a subgroup of $Sym(\Gamma_F)$, which then would make the degree just 2^n. Of course, an affine map $A : \mathbb{F}_2^{n+m} \to \mathbb{F}_2^{n+m}$ is not necessarily uniquely determined by its images on Γ_F, so in order to represent $Aut(F)$ as a subgroup of $Sym(\Gamma_F)$, we first need to deal with this issue. The way that we treat this depends on the concrete application. Recall that π was the induced action of an element from $\mathrm{Aut}(F) \leq \mathrm{Sym}(\mathbb{F}_2^{n+m})$ to an element in $\mathrm{Aut}^*(F) \leq \mathrm{Sym}(\Gamma_F)$.

- For our equivalence algorithms we can just view the automorphism group as a subgroup of $\mathrm{Sym}(\Gamma_G)$. This does not cause any issues, since correctness is guaranteed even if we only have a subgroup of the automorphism group.
- In our automorphism algorithms, we will also represent the group as a subgroup of $\mathrm{Sym}(\Gamma_F)$ in the search, and will then lift it to the full automorphism group afterwards.

To do the lifting we rely on the following Lemma.

Lemma 13 *Let $H \leq \mathrm{Aut}(F)$, with $Q := \pi(H)$ and $K := \ker(\pi|_H)$. If $Q = \langle q_1, \ldots, q_r \rangle$ and $g_1, \ldots, g_r \in H$ satisfy $\pi(g_i) = q_i$, then*

$$H = \langle K, g_1, \ldots, g_r \rangle.$$

[3] More precisely, when the current stabilizer subgroup of the current node grows.

Proof Let $g \in H$. By assumption, $Q = \langle \pi(g_1), \ldots, \pi(g_r) \rangle = \pi(\langle g_1, \ldots, g_r \rangle)$. Hence there exists $h \in \langle g_1, \ldots, g_r \rangle$ with $\pi(h) = \pi(g)$. Then $\pi(gh^{-1}) = 1$, so $gh^{-1} \in K$ and therefore $g \in \langle K, g_1, \ldots, g_r \rangle$. □

So the procedure in our automorphism algorithms is then:

1. Compute $Aut^*(F)$.
2. Compute $\ker(\pi)$, which is the subgroup of all elements of $\mathrm{Aut}(F)$ fixing every single element of Γ_F point-wise. In other words, this consists of all A where $A(p) = p$ for all $p \in \Gamma_F$. This is just a set of linear equations that we can easily solve. We can then extract the invertible solutions.
3. Lift a set of generators for $Aut^*(F)$ that have been found by the algorithm, then add these to the generators of the kernel of π.

When computing the $EA-$automorphism group we must add the linear constraints forcing the top-right block of A to be 0.

5.3 Quadratic functions

Quadratic functions (that is, functions of algebraic degree 2) represent an important special case in testing equivalence, both from a theoretical and a practical point of view. For example, the vast majority of mathematical constructions and search procedures for APN functions (whose investigation is one of the primary applications for equivalence testing algorithms) only produce quadratic functions, making this a particularly prominent use case for our algorithms. Furthermore, the prevalence of quadratic functions among the known instances of APN functions can be attributed to the relatively nice structural properties enjoyed by those functions, making their analysis in terms of cryptographic properties simpler. In the same vein, working with quadratic functions allows us to assume some structural properties that make their treatment with our algorithms easier. In particular, quadratic functions have a known automorphism subgroup, and we have the following well-known result [23].

Lemma 14 *Let $F : \mathbb{F}_2^n \to \mathbb{F}_2^m$ be a quadratic function and define*

$$\tau_a(x, y) = (x + a, y + D_a F(x)).$$

Then $\tau_a \in Aut_{EA}(F)$ for all $a \in \mathbb{F}_2^n$.

This means that all points in Γ_F are in the same orbit under the subgroup $\{\tau_a\}_{a \in \mathbb{F}_2^n}$, as formalized in the following lemma, which easily follows from Lemma 14.

Lemma 15 *Let $F : \mathbb{F}_2^n \to \mathbb{F}_2^m$ be a quadratic function and $H = \{\tau_a\}_{a \in \mathbb{F}_2^n}$ be a subgroup of $\mathrm{Aut}_{EA}(F)$, then $|\Gamma_F / H| = 1$.*

6 Algorithms

Now we are ready to present the complete algorithms for CCZ/EA-equivalence/automorphism. Since the CCZ and EA algorithms are almost identical

we will just give the pseudocode for the CCZ-algorithms. The only difference between the EA and CCZ versions is just checking the final affine map A at the end, the lifting step and the input automorphism group to the equivalence algorithm. We will use a couple of helper functions here.

- ValidEA(A) checks that A is a CCZ-mapping whose linear part has the block form required for an EA-mapping. More precisely, write

$$A(z) = Lz + t, \qquad z \in \mathbb{F}_2^{n+m},$$

  with $L \in \mathrm{GL}(n+m, 2)$ and $t \in \mathbb{F}_2^{n+m}$. Then A is an EA-mapping if the linear part L has the form

$$L = \begin{pmatrix} L_1 & 0 \\ L_3 & L_2 \end{pmatrix},$$

  where $L_1 \in \mathrm{GL}(n, 2)$, $L_2 \in \mathrm{GL}(m, 2)$, and $L_3 : \mathbb{F}_2^n \to \mathbb{F}_2^m$ is linear. Equivalently, this is the condition that [12]

$$L\big(\{0\} \times \mathbb{F}_2^m\big) \subseteq \{0\} \times \mathbb{F}_2^m.$$

- Update(A, p, q) just takes a partial affine map A and tries to map p to q, if this is possible it returns true and the new affine map, otherwise it returns false.
- OrbitPartition(P, H) for a point cell P and subgroup $H \leq \mathrm{Aut}^*(F)$ computes and returns P/H.

We say that we *individualize* a point if we put it into a singleton point cell, this happens whenever we branch on a point.

In this section, we give the full pseudocode for our algorithms, namely we present our algorithms for:

- Testing CCZ/EA-equivalence, Algorithm 7.
- Computing CCZ/EA automorphism groups, Algorithm 5.

6.1 Computing automorphism groups

We start with the automorphism algorithms. The difference from the equivalence algorithms is that we discover the automorphisms during the algorithm, so we need to update our data structures to reflect that. Algorithm 4 is the depth first search part of the automorphism algorithm. The current subgroup that we have found of $\mathrm{Aut}(F)$ is denoted H_{glob} and is updated whenever we find a new automorphism. H_{stab} is the current stabilizer subgroup, that is $H_{stab} = \mathrm{Stab}_A(H_{glob})$. We store this so that we do not have to recompute it at each node.

Algorithm 4: DFS-Auto$\big(A, (\Pi_F, \Pi_G), (\Lambda_F, \Lambda_G), H_{\text{stab}}, H_{\text{glob}}\big)$

Input: Partial affine map A, a point partition pair (Π_F, Π_G) and a hyperplane partition pair (Λ_F, Λ_G) (for the same function), a current stabilizer subgroup $H_{\text{stab}} = \text{Stab}_A(H_{glob})$ for orbit pruning, and a global accumulator $H_{\text{glob}} \leq \text{Aut}^*(F)$.

Output: The updated global accumulator H_{glob}.

$(\texttt{is-consistent}, \Pi_F, \Pi_G, \Lambda_F, \Lambda_G) \leftarrow \text{Refine}\big((\Pi_F, \Pi_G), (\Lambda_F, \Lambda_G)\big)$
if *is-consistent* = ***false*** **then**
  return H_{glob}
for $P_i^F \in I(\Pi_F)$ **do**
  $p, q \leftarrow$ the elements of P_i^F and P_i^G respectively
  $H_{stab} = \text{Stab}_q(H_{stab})$
  $(\texttt{is-consistent}, A) \leftarrow \text{Update}(A, p, q)$
  if *is-consistent* = ***false*** **then**
    return H_{glob}
if *A is fully determined on* Γ_F **then**
  if ValidCCZ(A) **then**
    $H_{\text{glob}} \leftarrow \langle H_{\text{glob}}, \pi(A) \rangle$
  return H_{glob}
Let P_i^F be the smallest cell in Π_F where $|P_i^F \setminus \text{Domain}(A)| > 1$
$p \leftarrow$ any element of $P_i^F \setminus \text{Domain}(A)$
$\mathcal{O} = (O_1, \ldots, O_s) \leftarrow \text{OrbitPartition}(P_i^G, H_{\text{stab}})$
$B \leftarrow \emptyset$ `// touched branch points (previously chosen q's)`
$Y \leftarrow$ one element from each orbit $O_j \in \mathcal{O}$
while $Y \neq \emptyset$ **do**
  Choose any $q \in Y$
  $Y \leftarrow Y \backslash \{q\}$
  $B \leftarrow B \cup \{q\}$
  $(\texttt{is-consistent}, A') \leftarrow \text{Update}(A, p, q)$
  if *is-consistent* **then**
    Individualize p and q in Π_F and Π_G to get Π'_F and Π'_G
    $H_{\text{glob}} \leftarrow \text{DFS-Auto}(A', \Pi'_F, \Pi'_G, \Lambda_F, \Lambda_G, \text{Stab}_q(H_{\text{stab}}), H_{\text{glob}})$
    if $Stab_A(H_{\text{glob}}) \supsetneq H_{\text{stab}}$ **then**
      $H_{\text{stab}} \leftarrow \text{Stab}_A(H_{\text{glob}})$
      $\mathcal{O} = (O_1, \ldots, O_s) \leftarrow \text{OrbitPartition}(P_i^G, H_{\text{stab}})$
      $Y \leftarrow$ one element from each orbit $O_j \in \mathcal{O}$ with $O_j \cap B = \emptyset$
return H_{glob}

Remark 3 (Branching on orbit representatives) In the branching step (the **while**-loop), we only branch on orbit representatives in the current target cell. During the search the stabilizer subgroup may grow, so the orbit partition of the cell must be recomputed; however, we do not have to start branching from scratch.

- B stores the branch points already tried (previously chosen y's).
- Y is the queue of candidates to branch on next.

Initially, Y contains one element from each orbit. Each time we branch on some $y \in Y$, we add it to B. Whenever the stabilizer subgroup increases, we recompute the orbit partition and refill Y with one representative from each *new* orbit that does not intersect B.

The CCZ automorphism algorithm is fairly straightforward now. We have observed that it is faster in practice to discard some hyperplanes already before the start of the refinement algorithms. We perform this by looking at the intersection count of each hyperplane with the graph, create the initial cells based on that and then discard the largest cells. This intersection count is given by the Walsh values and can be easily computed using FFHT. We also note that we do not necessarily need to let the algorithm finish, as we might not need to verify that we have found the entire automorphism group (finding just a subgroup can also be useful in many circumstances). In our implementation we therefore have the option to run this algorithm with a time limit.

Algorithm 5: CCZ-Automorphism(F)

Input: Function $F : \mathbb{F}_2^n \to \mathbb{F}_2^m$
Output: A generating set for $\mathrm{Aut}_{CCZ}(F)$ found by the search.
$\Pi_F \leftarrow (\Gamma_F,)$
$\Pi_G \leftarrow (\Gamma_F,)$
`// Initialize and truncate the hyperplane partitions using intersection numbers with` Γ_F
Let $S_t \leftarrow \{U \in \mathrm{Hyp}(\mathbb{F}_2^{n+m}) : |U \cap \Gamma_F| = t\}$
Let $T \leftarrow \{t : S_t \neq \emptyset\}$
Order T by decreasing $|2t - 2^n|$ (tie-break by increasing $|S_t|$)
$\Lambda_F \leftarrow (,)$
`num-hyperplanes` $\leftarrow 0$
for $t \in T$ *while* `num-hyperplanes` $<$ `HyperplaneFloor` **do**
  $\Lambda_F \leftarrow (\Lambda_F, S_t)$
  `num-hyperplanes` $\leftarrow$ `num-hyperplanes` $+ |S_t|$
$\Lambda_G \leftarrow \Lambda_F$
$A \leftarrow$ partial map $\mathbb{F}_2^{n+m} \to \mathbb{F}_2^{n+m}$
if F *is quadratic* **then**
  $_, A \leftarrow$ Update$(A, (0, F(0)), (0, F(0)))$
$H \leftarrow$ DFS-Auto$\big(A, (\Pi_F, \Pi_G), (\Lambda_F, \Lambda_G), \{1_{\Gamma_F}\}, \{1_{\Gamma_F}\}\big)$
if F *is quadratic* **then**
  $\tau_a : \ (x, y) \mapsto \big(x + a, \ y + D_a F(x)\big)$
  $H \leftarrow \langle H, \{\pi(\tau_a)\}_{a \in \mathbb{F}_2^n} \rangle$
`// Lifting step: recover the full subgroup of` $\mathrm{Aut}(F) \leq \mathrm{Sym}(\mathbb{F}_2^{n+m})$
Compute $K \leftarrow \{A \in \mathrm{Aut}(F) : A(p) = p \text{ for all } p \in \Gamma_F\}$ by solving the induced linear system
For each generator $h \in H$ (viewed as a permutation of Γ_F), extend it to an affine map $\widetilde{h} : \mathbb{F}_2^{n+m} \to \mathbb{F}_2^{n+m}$ with $\widetilde{h}(p) = h(p)$ for all $p \in \Gamma_F$
$H \leftarrow \langle K, \{\widetilde{h} : h \text{ a generator of } H\} \rangle$
return H

6.2 CCZ and EA equivalence

The DFS search for the equivalence algorithm is simpler since we do not have to update the orbit partitions the same way as in Algorithm 4 during the search. In Algorithm 6 we just return the first witness of a CCZ-equivalence we find. If one wants all CCZ-equivalences between two functions this is completely determined by the automorphism group of one of the functions and a single witness of the CCZ-equivalence between the two functions. In the following, H is the stabilizer subgroup of the current partial map.

Algorithm 6: $\mathrm{DFS}\big(A, (\Pi_F, \Pi_G), (\Lambda_F, \Lambda_G), H\big)$

Input: Partial affine map A, a point partition pair (Π_F, Π_G), a hyperplane partition pair (Λ_F, Λ_G), and a current H for orbit pruning.
Output: **false** or an affine permutation A fully determined on Γ_F witnessing CCZ-equivalence

$(\texttt{is-consistent}, \Pi_F, \Pi_G, \Lambda_F, \Lambda_G) \leftarrow \textsc{Refine}\big((\Pi_F, \Pi_G), (\Lambda_F, \Lambda_G)\big)$
if *is-consistent* = ***false*** **then**
  return ***false***
for $P_i^F \in I(\Pi_F)$ **do**
  $p, q \leftarrow$ the elements of P_i^F and P_i^G respectively
  $H \leftarrow \mathrm{Stab}_q(H)$
  $(\texttt{is-consistent}, A) \leftarrow \textsc{Update}(A, p, q)$
  if *is-consistent* = ***false*** **then**
    return ***false***
if *A is fully determined on Γ_F* **then**
  if $\textsc{ValidCCZ}(A)$ **then**
    return A
  else
    return ***false***
Let P_i^F be the smallest cell in Π_F where $|P_i^F \setminus \mathrm{Domain}(A)| > 1$
$p \leftarrow$ any element of $P_i^F \setminus \mathrm{Domain}(A)$
$\mathcal{O} = (O_1, \ldots, O_s) \leftarrow \textsc{OrbitPartition}(P_i^G, H)$
for $O_j \in \mathcal{O}$ **do**
  $q \leftarrow$ any element of O_j
  $(\texttt{is-consistent}, A') \leftarrow \textsc{Update}(A, p, q)$
  if *is-consistent* **then**
    Individualize p and q in Π_F and Π_G to get Π'_F and Π'_G
    $A_{\mathrm{eq}} \leftarrow \mathrm{DFS}\big(A', (\Pi'_F, \Pi'_G), (\Lambda_F, \Lambda_G), \mathrm{Stab}_q(H)\big)$
    if $A_{\mathrm{eq}} \neq$ ***false*** **then**
      return A_{eq}
return ***false***

The final part of our equivalence algorithm is then just the initialization. As we explain in Example 2, the automorphism group of G is very helpful for the equivalence algorithm, and so we will take this as an optional input. In our implementation of the equivalence algorithms we will first run the automorphism algorithm with an optimal time limit on both function, and then seed the equivalence algorithm with whatever group is largest (We not not do this in parallel in the experiments in Section 7). We also allow a subgroup of the automorphism group to be explicitly given, this can be very useful for example when testing equivalence to a power function, where the entire automorphism group is known [24] and can be given to the algorithm before doing any computation.

Algorithm 7: CCZ-EQUIVALENCE(F, G, H)

Input: Functions $F, G : \mathbb{F}_2^n \to \mathbb{F}_2^m$ and an optional pruning subgroup $H \leq \mathrm{Aut}^*(G)$.

Output: An affine permutation A with $A(\Gamma_F) = \Gamma_G$, or **false** if none exists.

$\Pi_F \leftarrow (\Gamma_F,)$

$\Pi_G \leftarrow (\Gamma_G,)$

`// Initialize and truncate the hyperplane partitions using intersection numbers (as in CCZ-AUTOMORPHISM)`

Let $S_t^F \leftarrow \{U \in \mathrm{AGr}(n+m-1, n+m) : |U \cap \Gamma_F| = t\}$

Let $S_t^G \leftarrow \{U \in \mathrm{AGr}(n+m-1, n+m) : |U \cap \Gamma_G| = t\}$

`// Abort early if the hyperplane ''shape'' differs`

Let $N_t^F \leftarrow |S_t^F|$ and $N_t^G \leftarrow |S_t^G|$

if $\exists t$ *such that* $N_t^F \neq N_t^G$ **then**

  return ***false***

Let $T \leftarrow \{t : N_t^F > 0\}$

Order T by decreasing $|2t - 2^n|$ (tie-break by increasing $|S_t^F|$)

$\Lambda_F \leftarrow (,)$

$\Lambda_G \leftarrow (,)$

`num-hyperplanes` $\leftarrow 0$

for $t \in T$ *while* `num-hyperplanes` $<$ `HyperplaneFloor` **do**

  $\Lambda_F \leftarrow (\Lambda_F, S_t^F)$

  $\Lambda_G \leftarrow (\Lambda_G, S_t^G)$

  `num-hyperplanes` $\leftarrow$ `num-hyperplanes` $+ |S_t^F|$

$A \leftarrow$ empty partial map, $\mathbb{F}_2^{n+m} \to \mathbb{F}_2^{n+m}$

$A \leftarrow \mathrm{DFS}\big(A, (\Pi_F, \Pi_G), (\Lambda_F, \Lambda_G), H\big)$

if $A =$ ***false*** **then**

  return ***false***

Extend A to a full mapping if it's a partial map

return A

7 Computational results

In order to empirically evaluate the efficiency and utility of our algorithms, and to compare them against existing methods, we generate test data and compare the performance. We tested the following algorithms.

- Algorithm 5 for automorphism and Algorithm 7 for equivalence. We just test the CCZ algorithms as according to our experiments, the EA algorithms perform very similarly.
- SboxU [25] has a CCZ-equivalence function which is simply reducing the problem to code equivalence and then calling the Sage code equivalence algorithm. Sage uses a partition refinement algorithm to solve this problem. This however was much slower than doing the same using Magma.

- GAP: we tried using GAP, but the code equivalence algorithm here crashes already for low dimensions.
- For quadratic functions we tested the EA-equivalence algorithm from [12], which is implemented in [25]. This algorithm is not at all efficient for APN functions however.
- Reducing the problem to code equivalence and solving that using Magma [26], here we can also compute the automorphism group. We note that Magma is proprietary software and not publicly available.

For APN functions the only competitive algorithms seem to be Magma and our algorithms. Due to this we only compare these two algorithms for the APN functions.

The table entries are left blank whenever computation was not able to finish after running multiple days on just a single case. All experiments are run on a single core.

7.1 APN functions

We start with testing quadratic APN functions which is probably the most common use case, and represents one of the hardest cases for equivalence testing algorithms. We start by testing the list of quadratic APN functions in dimension 7 from [27]. We report the runtime for:

- equivalence testing on **negative** (inequivalent) pairs, so just picking some of the functions from the list and then testing equivalence pairwise.
- equivalence testing on **positive** (equivalent) pairs, which is done by picking functions from the list and generating a CCZ-equivalent quadratic function using the methods in [28] implemented in SboxU.
- Computing the entire automorphism group.

For both the positive and negative examples we do not include the time to find the automorphism group since we track this time separately. So to get the total time for the algorithms we should add this time. Both Magma and our equivalence algorithms first compute the automorphism group of one of the functions.

Table 1 Average runtime (in seconds) for instances sampled from [27] (dimension $n = 7$).

Algorithm	Negative	Positive	Automorphism
Algorithm 5 and 7	12	6	36
Magma	6500	1	157

For $n = 8$ we use the list from [29], which reports about 1.6 million pairwise inequivalent quadratic APN functions. We do the experiment the same way as in the $n = 7$ case.

Table 2 Average runtime (in seconds) for instances sampled from [29] (dimension $n = 8$).

Algorithm	Negative	Positive	Automorphisms
Algorithm 5 and 7	1	0	1
Magma	73	0	0

Now we pick 10 APN functions from among the known CCZ-equivalence classes from infinite families of APN functions for different dimensions. These include non-quadratic APN functions. We then run the following tests

- We generate 4 functions that are EA-equivalent to them via $A_1 \circ F \circ A_2 + A$. So we have 50 functions in total with groups of 5 that are CCZ equivalent.
- We test the functions pairwise for equivalence.
- We compute the automorphism group.

For the negative and positive case we do not include the automorphism computation time. Since the runtime for the quadratic and non quadratic test cases vary a lot we separate their runtime, the case of equivalence between quadratic and non-quadratic counts as non-quadratic here. The times marked with * means we could not finish the entire experiment (so on all the functions that we tested the other algorithm on) giving a partial average, while for the entries not filled out we could not even finish a single function.

Table 3 Runtime (average, in seconds; quadratic/non-quadratic) across dimensions $n \in \{7, 8, 9, 10, 11, 12\}$ for CCZ equivalence testing and automorphism-group computations on APN functions.

Algorithm	$n = 7$	$n = 8$	$n = 9$	$n = 10$	$n = 11$	$n = 12$
Positive tests (s), mean (quad/non-quad)						
Algorithm 7	1/1	0/0	174/0	0/2	38/–	247/640
Magma	7/20	0/0	3512*/0	9/51	3012/–	1846*/19217
Negative tests (s), mean (quad/non-quad)						
Algorithm 7	4/2	0/0	436/0	1/0	63/8950[4]	746/669
Magma	6626/6458	153/182	–/0	–/–	–/–	–/–
Automorphism tests (s), mean (quad/non-quad)						
Algorithm 5	8/13	0/0	530/0	1/8	177/346902[5]	1237/3359
Magma	24/15	0/0	18994/0	16/61	6887/–	12371/20211

[4] the high runtime here is due to the nonq vs nonq tests, if we just consider quadratic vs nonq the average is just around 1 minute

[5] this is only run on 1 non-quadratic function as this took around 4 days

Table 4 Average peak memory usage (in MB; quadratic/non-quadratic) across dimensions $n \in \{7, 8, 9, 10, 11, 12\}$ for CCZ equivalence testing and automorphism-group computations on APN functions.

Algorithm	$n = 7$	$n = 8$	$n = 9$	$n = 10$	$n = 11$	$n = 12$
Positive tests (MB), (quad/non-quad)						
Algorithm 7	19/19	23/21	40/25	77/64	308/–	661/596
Magma	34/33	32/32	245*/36	175/99	7990/–	9857*/9846
Negative tests (MB), (quad/non-quad)						
Algorithm 7	19/19	22/21	38/35	71/68	283/355	628/582
Magma	35/34	32/32	–/–	–/–	–/–	–/–
Automorphism tests (MB), (quad/non-quad)						
Algorithm 5	19/19	22/20	40/24	76/62	343/365	1059/700
Magma	30/30	26/31	219/33	170/100	5021/–	6147/6112

We see here that the result can vary a lot by the specific function we choose, the $n = 9$ non-quadratic functions we choose for example happen to be very easy for both algorithms. It seems like $n = 12$ is the limit of any equivalence testing between APN for magma, especially when it comes to memory usage (many cases are not really feasible already before this in terms of time usage). While for our algorithm, especially for quadratic functions it seems feasible for all cases all the way up to dimension 15 (which is the maximum our implementation support) on a normal laptop.

7.2 The inverse function

the make the runtime more concrete we also test the inverse function x^{-1} over $\mathbb{F}_{2^n}$ for various values of n. This is APN only for odd n. We use sboxU to generate CCZ-equivalent functions and then report both the results for running the automorphism algorithms for all these functions and the positive equivalence test.

Table 5 Inverse function benchmark (non-APN). For each $n \in \{10, 11, 12, 13, 14, 15\}$ we generate CCZ-equivalent functions to $x \mapsto x^{-1}$ over $\mathbb{F}_{2^n}$ using sboxU and report (i) mean runtime (in seconds) and (ii) average peak memory (in MB) for the positive CCZ-equivalence test and the automorphism-group computation.

Algorithm	$n = 10$	$n = 11$	$n = 12$	$n = 13$	$n = 14$	$n = 15$
Runtimes (s)						
Positive equivalence test, mean (s)						
Algorithm 7	0	2	10	40	183	612
Magma	2	10	70	128	1596	–
Automorphism computation, mean (s)						
Algorithm 5	1	4	32	70	432	1371
Magma	3	11	78	131	1643	–
Memory (MB)						
Positive equivalence test, peak (MB)						
Algorithm 7	44	86	296	800	2315	24689
Magma	114	283	1405	1557	24757	–
Automorphism computation, peak (MB)						
Algorithm 5	43	102	474	1256	6314	37313
Magma	91	183	899	969	15277	–

Just for comparison we tried sage code equivalence for the positive $n = 10$ case here and it had not finished within an hour.

8 Complexity and correctness

8.1 Correctness

Lemma 16 *Let A be a partial map fully determined by a set of graph points $S_F \subseteq \Gamma_F$ and where $A(\mathrm{Domain}(A) \cap \Gamma_F) \subseteq \Gamma_F$, $H_{glob} \leq \mathrm{Aut}^*(F)$, $H_{stab} = \mathrm{Stab}_A(H_{glob})$, (Π_F, Π_G) a point partition pair satisfying Invariant property 1 and (Λ_F, Λ_G) be a hyperplane partition pair satisfying Invariant property 2. Then if*

$$H_{out} = \text{DFS-AUTO}(A, (\Pi_F, \Pi_G), (\Lambda_F, \Lambda_G), H_{stab}, H_{glob})$$

and $\mathrm{Ext}(A) = \{\pi(\widehat{A}) : \widehat{A}$ extends $A\}$ we have

$$\langle H_{glob}, \mathrm{Ext}(A)\rangle = H_{out}.$$

Proof We prove that $\langle H_{glob}, \mathrm{Ext}(A)\rangle \leq H_{out}$. by induction on

$$r(A) = |\Gamma_F \backslash \mathrm{Domain}(A)|.$$

The recursive calls always have a smaller value of $r(A)$.

We start with the base case $r(A) = 0$. Then A is fully determined on Γ_F. There are two possibilities. If A is not a CCZ-mapping, then $\mathrm{Ext}(A) = \emptyset$ and so the algorithm just returns H_{glob} and the claim follows.
If A is a CCZ-mapping then every extension of $\widehat{A}$ induces the same permutation on Γ_F, which is $\pi(A)$. Thus $\mathrm{Ext}(A) = \{\pi(A)\}$. In this case the algorithms returns

$$H_{out}\langle H_{glob}, \pi(A)\rangle$$

and so we see that

$$\langle H_{glob}, \mathrm{Ext}(A)\rangle \leq H_{out}$$

which concludes the base case.

Now assume A is a partial map and that the results holds for any partial map A' where $r(A') < r(A)$. If there does not exist any extension of A then the result clearly holds so assume $\widehat{A}$ is any extension of A. We want to show that $\pi(\widehat{A}) \in H_{out}$. The algorithms first refines the ordered partitions using Algorithm 3. If this returns False then by Lemma 11 there does not exist any extension of A. otherwise the invariant properties are preserved also by Lemma 11. The algorithm then updates any forced mappings $p \to q$ from $I(\Pi_F)$, by Invariant property 1 $\widehat{A}$ must satisfy $\widehat{A}(p) = q$ for all such p, q. If A is now fully determined then $\pi(A) = \pi(\widehat{A})$ and we return $\langle H_{glob}, \pi(A)\rangle$ which then clearly contains $\pi(\widehat{A})$. If A is not fully determined on Γ_F then the algorithm starts the process. Let $p \in P_i^F \setminus \mathrm{Domain}(A)$ be the branching point chosen by the algorithm, and let P_i^G be the corresponding target cell. Let $B \subseteq P_i^G$ be the set of target points that are selected in the while-loop. By construction of the while-loop, after it terminates, B intersects every orbit of P_i^G under $\mathrm{Stab}_A(H_{\mathrm{out}})$.

Now let $\pi(\widehat{A}) \in \mathrm{Ext}(A)$, and put $q = \widehat{A}(p)$. By Invariant property 1, we have $q \in P_i^G$. Hence there exists $q' \in B$ such that q and q' lie in the same $\mathrm{Stab}_A(H_{\mathrm{out}})$-orbit. By Lemma 12, there exists an extension $\widetilde{A}$ of A such that $\widetilde{A}(p) = q'$ and $\pi(\widehat{A}) = h \circ \pi(\widetilde{A})$ for some $h \in \mathrm{Stab}_A(H_{\mathrm{out}})$. In particular,

$$h \in H_{\mathrm{out}}.$$

Since $\widetilde{A}$ is an extension of A with $\widetilde{A}(p) = q'$, the update $\mathrm{Update}(A, p, q')$ cannot return false. Let A' be the partial affine map returned by this update. Then $\widetilde{A}$ extends A', so

$$\pi(\widetilde{A}) \in \mathrm{Ext}(A').$$

The recursive call corresponding to the branch $p \mapsto q'$ has smaller value of r, so by the induction hypothesis its returned group contains $\pi(\widetilde{A})$. Since the global accumulator only grows during the algorithm, this returned group is contained in H_{out}. Therefore

$$\pi(\widetilde{A}) \in H_{\mathrm{out}}.$$

Thus

$$\pi(\widehat{A}) = h \circ \pi(\widetilde{A}) \in H_{\mathrm{out}}.$$

Since $\pi(\widehat{A}) \in \mathrm{Ext}(A)$ was arbitrary, we have

$$\mathrm{Ext}(A) \subseteq H_{\mathrm{out}}.$$

Moreover $H_{\mathrm{glob}} \leq H_{\mathrm{out}}$, since the algorithm only enlarges the global accumulator. Hence

$$\langle H_{\mathrm{glob}}, \mathrm{Ext}(A)\rangle \leq H_{\mathrm{out}}.$$

For showing that $\langle H_{glob}, \mathrm{Ext}(A)\rangle \geq H_{out}$. We observe that every added generator to the global accumulator H_{glob} is on the form $\pi(\widehat{A})$ where $\widehat{A}$ is an extension of A since it returns true on ValidCCZ. So every generator added lies in $\mathrm{Ext}(A)$.

□

Lemma 17 *For any function* $F : \mathbb{F}_2^n \to \mathbb{F}_2^m$ *Algorithm 5 returns* $\mathrm{Aut}(F)$.

Proof Let H be the group returned by Algorithm 5.

We first need to verify that Invariant property 1 and Invariant property 2 hold when the algorithm calls Algorithm 4. The values of Π_F and Π_G are initially set to $(\Gamma_F,)$ so Invariant property 1 clearly holds. For the hyperplane partitions we do not necessarily keep all of the hyperplanes, but removing some cells does not change anything, since we always remove pairs of correspond cells so this still maintains Invariant property 2. Now there are two cases to consider depending on whether F is quadratic or not.

- If F is not quadratic, we just apply (4) directly with a partial map with empty domain and empty H_{glob}. Lemma 16 then gives us that

$$\langle Ext(A) \rangle = H_{out}$$

- If F is quadratic, we map $(0, F(0))$ to $(0, F(0))$ before calling Algorithm 4. Let A' be this partial map. Then by Lemma 16 we have

$$\langle \mathrm{Ext}(A') \rangle = H_{out}.$$

  Let $\widehat{A}$ be any extension of the empty partial map. By Lemma 15 all points in Γ_F are in the same orbit under the subgroup $H' = \{\tau_a\}_{a \in \mathbb{F}_2^n}$ and therefore also under $\pi(H')$. So by Lemma 12 applied to A there exists an extension $\widehat{A'}$ of A' and element $\tau_a \in \pi(H')$ such that $\pi(\widehat{A}) = \tau_a \circ \pi(\widehat{A'})$. So then

$$\pi(\widehat{A}) \in \langle \mathrm{Ext}(A'), \pi(H') \rangle$$

  .

So in both cases we have found the entire subgroup $\pi(\mathrm{Aut}(F)) \leq \mathrm{Sym}(\Gamma_F)$. Denote this group by H. We then lift this to the full automorphism group, and the correctness here follows by Lemma 13.

The analogous Lemmas for the correctness of Algorithm 6 and Algorithm 7 can be proved exactly the same way, so we do not include this. For the EA-versions of these algorithms also not much changes, only that we now verify that we have an EA-mapping instead, and the final lifting to the full $Aut_{EA}(F)$ is a bit different.

□

8.2 Group operations

Our algorithms maintain subgroups of automorphisms of the graph Γ_F. We represent each automorphism as the induced permutation of Γ_F (of size 2^n), and use standard computational group theory machinery (a base/strong generating set with Schreier–Sims) for all group computations [19].

At a high level, we only need to support three operations during the search:

- compute orbit partitions of points under a subgroup (to select representatives for branching);

- incrementally add newly found automorphisms to the global group (to improve pruning);
- compute stabilizers of points/partial maps (to restrict the allowed targets in deeper recursion).

In practice, these group operations are lightweight compared to the refinement and validation steps, so a straightforward implementation suffices.

8.3 Runtime and memory

We will not try to give a detailed complexity analysis of our algorithm, as it varies heavily based on which function is tested. We just note that the partition refinement step, which runs using the FFHT in about $\mathcal{O}((n+m)2^{n+m})$. This is where the algorithm spends the majority of the time. The total runtime of the algorithm will be around this times the number of search nodes.

One of the main advantages of our algorithm is the low memory consumption. For each partial map we store the hyperplanes (just a list of incidences of which hyperplanes are in which cell). There are $\mathcal{O}(2^{n+m})$ affine hyperplanes, so this will dominate our memory cost compared to the other objects we need to store (partial map, point partitions, stabilizer subgroup, etc). The total memory cost is then around $\mathcal{O}(d2^{n+m})$ where d is the max-depth of the search tree.

9 Conclusion

We have presented algorithms for testing CCZ and EA equivalence and computing the automorphism groups of vectorial Boolean functions under CCZ and EA equivalence. In contrast to the code isomorphism test, our methods consume significantly less memory and time, and in contrast to the other existing approaches, they can be applied to any pair of functions, regardless of their algebraic degree, image set size, and other properties. This allows us to test CCZ and EA equivalence between an arbitrary pair of functions in higher dimensions than were previously possible.

Some interesting future work is in optimizing the actual implementation of the algorithm and parallelizing it. Some practical applications of the algorithm could involve applying it to known corpora of functions that have not been tractable before, and using it to compute the automorphism groups of the known representatives.

References

[1] Gregor Leander and Axel Poschmann. On the classification of 4 bit s-boxes. In *International workshop on the arithmetic of finite fields*, pages 159–176. Springer, 2007.

[2] Dušan Božilov, Begül Bilgin, and Hacı Ali Sahin. A note on 5-bit quadratic permutations' classification. *IACR Transactions on Symmetric Cryptology*, pages 398–404, 2017.

[3] Marco Calderini. On the EA-classes of known APN functions in small dimensions. *Cryptography and Communications*, 12(5):821–840, 2020.

[4] Konstantin Kalgin and Valeriya Idrisova. The classification of quadratic APN functions in 7 variables. *Cryptology ePrint Archive*, 2020.

[5] Alex Biryukov, Christophe De Canniere, An Braeken, and Bart Preneel. A toolbox for cryptanalysis: Linear and affine equivalence algorithms. In *International conference on the theory and applications of cryptographic techniques*, pages 33–50. Springer, 2003.

[6] Itai Dinur. An improved affine equivalence algorithm for random permutations. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 413–442. Springer, 2018.

[7] Ivana Ivkovic and Nikolay Kaleyski. Deciding and reconstructing linear equivalence of uniformly distributed functions. *Cryptology ePrint Archive*, 2022.

[8] Chung Hun Baek, Jung Hee Cheon, and Hyunsook Hong. White-box aes implementation revisited. *Journal of Communications and Networks*, 18(3):273–287, 2016.

[9] Jung Hee Cheon, Hyunsook Hong, Joohee Lee, and Jooyoung Lee. An efficient affine equivalence algorithm for multiple s-boxes and a structured affine layer. In *International Conference on Selected Areas in Cryptography*, pages 299–316. Springer, 2016.

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

[11] Yves Edel and Alexander Pott. On the equivalence of nonlinear functions. In *Enhancing cryptographic primitives with techniques from error correcting codes*, pages 87–103. IOS Press, 2009.

[12] Anne Canteaut, Alain Couvreur, and Léo Perrin. Recovering or testing extended-affine equivalence. *IEEE Transactions on Information Theory*, 68(9):6187–6206, 2022.

[13] Nikolay Kaleyski. Deciding EA-equivalence via invariants. *Cryptography and Communications*, 14(2):271–290, 2022.

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

[15] Gábor P Nagy. On the minimum hamming distance between vectorial boolean and affine functions. *Cryptography and Communications*, 17(6):1703–1720, 2025.

[16] K.A. Browning, J.F. Dillon, R.E. Kibler, and M.T. McQuistan. APN polynomials and related codes. *Journal of Combinatorics, Information System Sciences*, 34, 01 2009.

[17] Jeffrey Leon. Computing automorphism groups of error-correcting codes. *IEEE Transactions on Information Theory*, 28(3):496–511, 1982.

[18] Satoshi Yoshiara. Equivalences of quadratic APN functions. *Journal of Algebraic Combinatorics*, 35(3):461–475, 2012.

[19] Charles C Sims. Computational methods in the study of permutation groups. In *Computational problems in abstract algebra*, pages 169–183. Elsevier, 1970.

[20] Boris Weisfeiler and Andrei Leman. The reduction of a graph to canonical form and the algebra which appears therein. *nti, Series*, 2(9):12–16, 1968.

[21] Claude Carlet. Boolean functions for cryptography and coding theory. 2021.

[22] Christof Beierle, Marcus Brinkmann, and Gregor Leander. Linearly self-equivalent APN permutations in small dimension. *IEEE Transactions on Information Theory*, 67(7):4863–4875, 2021.

[23] Carl Bracken, Eimear Byrne, Gary McGuire, and Gabriele Nebe. On the equivalence of quadratic APN functions. *Designs, Codes and Cryptography*, 61(3): 261–272, 2011.

[24] Ulrich Dempwolff. Automorphism groups of power functions. *Journal of Group Theory*. doi: doi:10.1515/jgth-2023-0071. URL https://doi.org/10.1515/jgth-2023-0071.

[25] sboxU: black-box analysis of discrete functions. Available online at https://github.com/lpp-crypto/sboxU/, 2026.

[26] Wieb Bosma, John Cannon, and Catherine Playoust. The Magma algebra system. I. The user language. *J. Symbolic Comput.*, 24(3-4):235–265, 1997. ISSN 0747-7171. doi: 10.1006/jsco.1996.0125. URL http://dx.doi.org/10.1006/jsco.1996.0125. Computational algebra and number theory (London, 1993).

[27] Konstantin Kalgin and Valeriya Idrisova. The classification of quadratic APN functions in 7 variables and combinatorial approaches to search for APN functions. *Cryptography and Communications*, 15(2):239–256, 2023.

[28] Anne Canteaut and Léo Perrin. On ccz-equivalence, extended-affine equivalence, and function twisting. *Finite Fields and Their Applications*, 56:209–246, 2019.

[29] Christof Beierle, Philippe Langevin, Gregor Leander, Alexandr Polujan, and Shahram Rasoolzadeh. Millions of inequivalent quadratic APN functions in eight variables. *arXiv preprint arXiv:2508.04644*, 2025.