

Algorithms for Computing Affine Equivalence and Self-Equivalence Groups

Joakim Sunde

University of Bergen
joakim.sunde@uib.no

Abstract. We study affine and linear equivalence of functions $F, G : \mathbb{F}_q^n \rightarrow \mathbb{F}_q^m$. We present practical equivalence algorithms that handle arbitrary functions (any algebraic degree, permutations and non-permutations) and are particularly effective on structured instances with nontrivial self-equivalences, covering most cryptographically relevant cases. We also give a dedicated algorithm for computing generators for the group of self-equivalences (A_1, A_2) satisfying $A_2 \circ F = F \circ A_1$, for which no comparably efficient general method was previously available.

Keywords: Affine equivalence · S-box · self-equivalence.

1 Introduction

Let q be a prime power, and let $\mathbb{F}_q$ denote the finite field with q elements. For functions $F, G : \mathbb{F}_q^n \rightarrow \mathbb{F}_q^m$, we say that F and G are *affine equivalent* if there exist affine bijections $A_1 : \mathbb{F}_q^n \rightarrow \mathbb{F}_q^n$ and $A_2 : \mathbb{F}_q^m \rightarrow \mathbb{F}_q^m$ such that

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

If A_1 and A_2 can be chosen linear, then F and G are *linearly equivalent*. Given F and G , deciding whether such maps exist is the *affine equivalence problem* (or the *linear equivalence problem*, respectively). These problems arise, for example, in the classification of S-boxes [17], S-box analysis and reverse engineering [9], and white-box cryptography [18]. Several algorithms are known: the methods in [8] work best when F and G are permutations, whereas the rank-table approach of [20] is most effective for random permutations of high algebraic degree.

A closely related task is the affine/linear *self-equivalence* problem: given F , find nontrivial affine/linear permutations (A_1, A_2) such that $A_2 \circ F = F \circ A_1$. Self-equivalences are central in commutative cryptanalysis [4], a generalization of differential cryptanalysis [3]. They also arise in equivalent representations of block ciphers [2, 31], searches for cryptographically strong functions [5, 7], and threshold implementations [33]. Beyond their direct applications, self-equivalences can be exploited to reduce search spaces via symmetry; we use this for pruning in our equivalence algorithms. Nevertheless, there is currently no efficient general algorithm dedicated to computing self-equivalence groups, and the standard approach is to apply an equivalence algorithm exhaustively (as in [4]). This

becomes costly when the self-equivalence group is large or when n is large. For instance, [4] adapts the algorithm of [8] to enumerate affine self-equivalences and implements it in SboxU [30]. In our experiments on the AES S-box, this enumeration takes 45 minutes, while our method computes generators for the same group in less than one second on a standard laptop. Table 1 reports our experiments on inverse functions in larger dimensions.

1.1 Our contributions

Our main contributions are:

- We give a practical algorithm for deciding affine and linear equivalence of arbitrary functions $F, G : \mathbb{F}_q^n \rightarrow \mathbb{F}_q^m$, including non-permutations.
- In our experiments, our implementation outperforms the algorithms of [8] on almost all instances (the exceptions being very small positive instances where both algorithms are very fast), with orders-of-magnitude speedups on structured functions with nontrivial self-equivalences.
- We present the first dedicated algorithms for computing generators for affine and linear self-equivalence groups, consisting of pairs (A_1, A_2) satisfying $A_2 \circ F = F \circ A_1$, using computational group-theoretic primitives that are especially effective when the self-equivalence group is large.

Implementations. We provide two implementations: **one specialized to $\mathbb{F}_2$ and one supporting general fields.**

1.2 Previous work and state of the art

For general S-boxes, a widely used approach to linear and affine equivalence is the algorithm of [8]. While it also applies to non-permutations, its complexity increases quickly outside the permutation setting. For random permutations of $\mathbb{F}_{2^n}$, the rank-table method of [20] has a stated complexity of about $n^3 2^n$ bit operations with very high probability. For random permutations it outperforms our approach. However, rank-table techniques are tailored to functions of high algebraic degree and often fail on structured instances (e.g., the inverse function).

Several algorithms are known for special cases. For k -to-1 functions satisfying additional structural conditions, see [22]. For quadratic vectorial Boolean functions, [12] gives an extended-affine (EA) equivalence recovery algorithm and ortho-derivative-based invariants for distinguishing quadratic APN functions. For the special case of an S-box layer, see, for example, [1, 16].

CCZ and EA equivalence are more general equivalence relations than affine and linear equivalence and are often used in classifying functions. Following the graph-based notion introduced in [15], CCZ equivalence is defined as follows: two functions $F, G : \mathbb{F}_q^n \rightarrow \mathbb{F}_q^m$ are CCZ equivalent if there exists an affine permutation A of $\mathbb{F}_q^{n+m}$ such that

$$A(\Gamma_F) = \Gamma_G,$$

where $\Gamma_F = \{(x, F(x)) : x \in \mathbb{F}_q^n\}$ denotes the graph of F . For vectorial Boolean functions, a dedicated partition-refinement algorithm was recently developed in [25]. Our algorithms here build on this work by applying similar ideas to the linear and affine equivalence problem. The main differences between the algorithms in [25] and the algorithm we present here are:

- We maintain separate ordered partitions for the domain and codomain and allow branching on either side.
- We branch on both points and hyperplanes.
- We use different signatures because those in [25] are too costly to compute for our purposes.

The algorithm in [25] could naturally also be applied to the affine and linear equivalence problem with limited modification, so we also include this algorithm in our runtime comparison.

1.3 Organization of the paper

Section 2 introduces the notation and basic objects, and Section 3 outlines the search framework. Section 4 defines signatures and explains how we use them for refinement. Section 5 then instantiates these signatures for points and hyperplanes. Section 6 explains orbit-based pruning using self-equivalences. The full equivalence and self-equivalence algorithms are presented in Section 7, with experiments in Section 8 and correctness in Section 9.

2 Notation and preliminaries

Let $\mathbb{F}_q$ be the finite field of order $q = p^s$, where p is prime, and view $\mathbb{F}_q^d$ as the d -dimensional vector space over $\mathbb{F}_q$. For a function $F : \mathbb{F}_q^n \rightarrow \mathbb{F}_q^m$, we write

$$F^{-1}(y) = \{x \in \mathbb{F}_q^n : F(x) = y\}.$$

An *affine permutation* of $\mathbb{F}_q^d$ is a bijection of the form $A(x) = L(x) + t$, where $L \in \text{GL}(d, q)$ is an invertible $\mathbb{F}_q$ -linear map and $t \in \mathbb{F}_q^d$. We denote the group of affine permutations by $\text{AGL}(d, q)$.

An *affine hyperplane* in $\mathbb{F}_q^d$ is a coset of a $(d-1)$ -dimensional linear subspace; equivalently, it has the form

$$H_{a,b} = \{x \in \mathbb{F}_q^d : a \cdot x = b\},$$

for some $a \in \mathbb{F}_q^d \setminus \{0\}$ and $b \in \mathbb{F}_q$, where $a \cdot x$ denotes the standard dot product. We write $\mathcal{H}_d$ for the set of all of these affine hyperplanes in $\mathbb{F}_q^d$.

Two functions $F, G : \mathbb{F}_q^n \rightarrow \mathbb{F}_q^m$ are *affine equivalent* if there exist $A_1 \in \text{AGL}(n, q)$ and $A_2 \in \text{AGL}(m, q)$ such that $A_2 \circ F = G \circ A_1$. If A_1 and A_2 can be chosen linear, then F and G are *linearly equivalent*.

2.1 Group theory

A *group* is a set $\mathcal{G}$ together with a binary operation $\cdot : \mathcal{G} \times \mathcal{G} \rightarrow \mathcal{G}$. We require the operation to be associative, to admit an identity element $e \in \mathcal{G}$, and we require every element $g \in \mathcal{G}$ to have an inverse $g^{-1} \in \mathcal{G}$. A group $\mathcal{G}$ is *abelian* if $g \cdot h = h \cdot g$ for all $g, h \in \mathcal{G}$. If $\mathcal{K}$ is a subgroup of $\mathcal{G}$, we write $\mathcal{K} \leq \mathcal{G}$.

A *(left) group action* of $\mathcal{G}$ on a set X is a map $\mathcal{G} \times X \rightarrow X$, $(g, x) \mapsto g \cdot x$, such that $e \cdot x = x$ and $(gh) \cdot x = g \cdot (h \cdot x)$ for all $g, h \in \mathcal{G}$ and $x \in X$. For $x \in X$, its *orbit* is $\mathcal{G} \cdot x := \{g \cdot x \mid g \in \mathcal{G}\}$. If $\mathcal{K} \leq \mathcal{G}$ is a subgroup acting on X by restriction, we write

$$X/\mathcal{K} := \{\mathcal{K} \cdot x \mid x \in X\}$$

for the set of $\mathcal{K}$ -orbits (equivalently, the quotient of X by the relation $x \sim y \iff \exists k \in \mathcal{K} : y = k \cdot x$). The *(point) stabilizer* of x is $\mathcal{G}_x := \{g \in \mathcal{G} \mid g \cdot x = x\} \leq \mathcal{G}$. More generally, for a subset $S \subseteq X$ we define the *setwise stabilizer*

$$\mathcal{G}_S := \{g \in \mathcal{G} \mid g \cdot S = S\} \leq \mathcal{G}.$$

A *group homomorphism* between groups $\mathcal{G}$ and H is a map $\varphi : \mathcal{G} \rightarrow H$ such that $\varphi(g_1 \cdot g_2) = \varphi(g_1) \cdot \varphi(g_2)$ for all $g_1, g_2 \in \mathcal{G}$. A *(complex) character* of a (finite abelian) group $\mathcal{G}$ is a homomorphism $\chi : \mathcal{G} \rightarrow \mathbb{C}^\times$.

Definition 1 (Self-equivalence group). Let $F : \mathbb{F}_q^n \rightarrow \mathbb{F}_q^m$. The set of affine self-equivalences (SE) of F forms a group, called the affine self-equivalence group of F , defined by

$$\text{Aut}(F) := \{(A_1, A_2) \in \text{AGL}(n, q) \times \text{AGL}(m, q) \mid A_2 \circ F = F \circ A_1\}.$$

Its linear self-equivalence group is the subgroup

$$\text{Aut}_{\text{lin}}(F) := \{(L_1, L_2) \in \text{GL}(n, q) \times \text{GL}(m, q) \mid L_2 \circ F = F \circ L_1\} \leq \text{Aut}(F).$$

Throughout this paper, $\text{Aut}(F)$ denotes the affine self-equivalence group. We explicitly distinguish linear and CCZ self-equivalence groups when these are considered. The group operation is componentwise composition:

$$(A_1, A_2) \cdot (B_1, B_2) := (A_1 \circ B_1, A_2 \circ B_2),$$

where $(A_1, A_2), (B_1, B_2) \in \text{Aut}(F)$.

We will use the natural action of $\text{AGL}(d, q)$ on points and affine hyperplanes. For $A(x) = L(x) + t \in \text{AGL}(d, q)$ and a point $x \in \mathbb{F}_q^d$, we set $A \cdot x := A(x)$. For an affine hyperplane $H \in \mathcal{H}_d$, we set

$$A \cdot H := A(H) := \{A(x) \mid x \in H\} \in \mathcal{H}_d.$$

In particular, if $H = H_{a,b} = \{x \in \mathbb{F}_q^d : a \cdot x = b\}$, then

$$A \cdot H_{a,b} = H_{L^{-\top}a, b + (L^{-\top}a) \cdot t},$$

where $L^{-T} = (L^{-1})^T$ is the transpose of the inverse.

For any $(A_1, A_2) \in \text{Aut}(F)$, A_1 acts on domain objects, while A_2 acts on codomain objects.

Definition 2 (Coordinate projections). For a pair $A = (A_1, A_2) \in \text{AGL}(n, q) \times \text{AGL}(m, q)$ we define the coordinate projections by $\pi_1(A) := A_1$ and $\pi_2(A) := A_2$. For a subgroup $\mathcal{K} \leq \text{AGL}(n, q) \times \text{AGL}(m, q)$ we set

$$\pi_1(\mathcal{K}) = \{\pi_1(A) \mid A \in \mathcal{K}\}, \quad \pi_2(\mathcal{K}) = \{\pi_2(A) \mid A \in \mathcal{K}\}.$$

2.2 Ordered partitions and branching decisions

Our algorithm is a partition refinement algorithm, relying on ordered partitions.

Definition 3 (Ordered partition). An ordered partition¹ of a finite set X is a sequence $\Pi = (B_1, \dots, B_k)$ of pairwise disjoint subsets of X whose union is X . The sets B_i are called the cells of Π .

Our algorithm proceeds by depth-first search (DFS). At each search node, we maintain partial information about the unknown mappings (A_1, A_2) , recorded explicitly.

Definition 4 (Branch decisions). A branch decision is one of the following constraints:

1. a domain point decision $A_1(x) = y$ for some $x, y \in \mathbb{F}_q^n$;
2. a domain hyperplane decision $A_1(U) = V$ for some $U, V \in \mathcal{H}_n$;
3. a codomain point decision $A_2(x) = y$ for some $x, y \in \mathbb{F}_q^m$; or
4. a codomain hyperplane decision $A_2(U) = V$ for some $U, V \in \mathcal{H}_m$.

A set of branch decisions is

$$\Delta := (\Delta_{\text{pt}}^{(1)}, \Delta_{\text{hyp}}^{(1)}, \Delta_{\text{pt}}^{(2)}, \Delta_{\text{hyp}}^{(2)}),$$

where $\Delta_{\text{pt}}^{(1)}$ and $\Delta_{\text{hyp}}^{(1)}$ contain decisions on A_1 , and $\Delta_{\text{pt}}^{(2)}$ and $\Delta_{\text{hyp}}^{(2)}$ contain decisions on A_2 .

For convenience, we write $\Delta \cup \{A_1(x) = y\}$ when adding the constraint $A_1(x) = y$ to Δ , and similarly for the other types of decisions.

Given a set of decisions, we consider the set of solutions that are consistent with them. That is, we consider all pairs of affine permutations that satisfy the decisions in Δ and witness an affine equivalence between F and G .

Definition 5 (Consistent completion). A pair of affine permutations (A_1, A_2) is a consistent completion of a set of branch decisions Δ for the functions F and G if

1. (A_1, A_2) witnesses an affine equivalence between F and G , i.e., $A_2 \circ F = G \circ A_1$; and
2. A_1 satisfies every constraint in $\Delta_{\text{pt}}^{(1)} \cup \Delta_{\text{hyp}}^{(1)}$, and A_2 satisfies every constraint in $\Delta_{\text{pt}}^{(2)} \cup \Delta_{\text{hyp}}^{(2)}$.

¹ Typically the cells are required to be non-empty; we allow empty cells for convenience.

For a set of branch decisions Δ and fixed functions F and G , let $\mathcal{C}(\Delta)$ be the set of consistent completions of Δ , i.e.,

$$\mathcal{C}(\Delta) = \{(A_1, A_2) : (A_1, A_2) \text{ is a consistent completion of } \Delta\}.$$

Definition 6 (Branch stabilizer). *Let $\Delta = (\Delta_{\text{pt}}^{(1)}, \Delta_{\text{hyp}}^{(1)}, \Delta_{\text{pt}}^{(2)}, \Delta_{\text{hyp}}^{(2)})$ be a set of branch decisions and $H \leq \text{Aut}(G)$. The associated branch stabilizer is the subgroup*

$$\begin{aligned} H_\Delta := \{ \varphi \in H \mid & \pi_1(\varphi)(y) = y \text{ for all } (A_1(x) = y) \in \Delta_{\text{pt}}^{(1)}, \\ & \pi_1(\varphi)(V) = V \text{ for all } (A_1(U) = V) \in \Delta_{\text{hyp}}^{(1)}, \\ & \pi_2(\varphi)(y) = y \text{ for all } (A_2(x) = y) \in \Delta_{\text{pt}}^{(2)}, \\ & \pi_2(\varphi)(V) = V \text{ for all } (A_2(U) = V) \in \Delta_{\text{hyp}}^{(2)} \}. \end{aligned}$$

Thus, every element of H_Δ fixes each target point individually and fixes each target hyperplane setwise. Any element $g = (g_1, g_2) \in H_\Delta$ acts on $\mathcal{C}(\Delta)$ by

$$g \cdot (A_1, A_2) = (g_1 \circ A_1, g_2 \circ A_2).$$

3 Overview of the algorithms

At a high level, our method belongs to the individualization-refinement, or partition-backtrack, family of algorithms used in graph isomorphism and permutation-group computation [27, 29]. It alternates invariant refinement of ordered partitions with individualization and depth-first search, while stabilizer orbits are used to prune equivalent branches. Our setting differs in maintaining four paired partitions over domain and codomain points and hyperplanes, and in using affine-incidence and function-transport signatures computed via finite-field Radon transforms.

Fix functions $F, G : \mathbb{F}_q^n \rightarrow \mathbb{F}_q^m$ throughout the rest of the paper. Our equivalence algorithm performs a depth-first search (DFS). The self-equivalence algorithm uses the same search with $F = G$, but repeatedly seeks a self-equivalence outside the currently known subgroup H , enlarging H until no such element remains. We maintain the following data (subject to invariants stated below):

Δ		the current set of branch decisions,
H	$\leq \text{Aut}(G)$	subgroup of the self-equivalence group of G ,
$H^{(1)}$	$\leq \pi_1(H_\Delta)$	domain-side stabilizer subgroup for the current decisions,
$H^{(2)}$	$\leq \pi_2(H_\Delta)$	codomain-side stabilizer subgroup for the current decisions,
$(\mathcal{P}_F, \mathcal{P}_G)$	partitions of $\mathbb{F}_q^n$,	input-point partitions for F and G ,
$(\mathcal{Q}_F, \mathcal{Q}_G)$	partitions of $\mathbb{F}_q^m$,	output-point partitions for F and G ,
$(\mathcal{L}_F, \mathcal{L}_G)$	partitions of $\mathcal{H}_n$,	domain-hyperplane partitions for F and G ,
$(\mathcal{R}_F, \mathcal{R}_G)$	partitions of $\mathcal{H}_m$,	codomain-hyperplane partitions for F and G .

Each pair $(\mathcal{P}_F, \mathcal{P}_G)$, $(\mathcal{Q}_F, \mathcal{Q}_G)$, $(\mathcal{L}_F, \mathcal{L}_G)$ and $(\mathcal{R}_F, \mathcal{R}_G)$ consists of two ordered partitions: one associated with F and one associated with G . Write these partitions explicitly as

$$\begin{aligned}\mathcal{P}_F &= (P_1^F, \dots, P_k^F), & \mathcal{P}_G &= (P_1^G, \dots, P_k^G), \\ \mathcal{Q}_F &= (Q_1^F, \dots, Q_{k'}^F), & \mathcal{Q}_G &= (Q_1^G, \dots, Q_{k'}^G), \\ \mathcal{L}_F &= (L_1^F, \dots, L_l^F), & \mathcal{L}_G &= (L_1^G, \dots, L_l^G), \\ \mathcal{R}_F &= (R_1^F, \dots, R_r^F), & \mathcal{R}_G &= (R_1^G, \dots, R_r^G).\end{aligned}$$

The algorithm maintains the following alignment invariants: every consistent completion (A_1, A_2) of Δ respects the paired ordered partitions.

Invariant 1 (Domain hyperplane invariant). A_1 respects the ordered domain-hyperplane partitions $\mathcal{L}_F$ and $\mathcal{L}_G$: for all indices i ,

$$A_1(L_i^F) = L_i^G.$$

Invariant 2 (Codomain hyperplane invariant). A_2 respects the ordered codomain-hyperplane partitions $\mathcal{R}_F$ and $\mathcal{R}_G$: for all indices i ,

$$A_2(R_i^F) = R_i^G.$$

Invariant 3 (Input point-cell invariant). A_1 respects the ordered input-point partitions $\mathcal{P}_F$ and $\mathcal{P}_G$: for all indices i ,

$$A_1(P_i^F) = P_i^G.$$

Invariant 4 (Output point-cell invariant). A_2 respects the ordered output-point partitions $\mathcal{Q}_F$ and $\mathcal{Q}_G$: for all indices i ,

$$A_2(Q_i^F) = Q_i^G.$$

At a high level, the DFS explores a tree of *guesses* about the unknown affine maps. The goal is to keep the search tree small by refining the partitions as much as possible between guesses. We select an object to branch on (typically a domain point or a domain hyperplane). Suppose we branch on domain points. We pick the smallest cell of $\mathcal{P}_F$ with at least two elements, say P_i^F , and choose an element $x \in P_i^F$. By the [Input point-cell invariant](#), any consistent completion (A_1, A_2) of Δ must map x to some point in P_i^G . Thus we only need to consider $|P_i^G|$ candidates; moreover, orbit pruning reduces this to one representative from each orbit in $P_i^G/H^{(1)}$ (Section 6).

If we choose a candidate $y \in P_i^G$, we add the decision $A_1(x) = y$ to Δ and update the partitions so that x and y become singleton cells in the corresponding positions. We also update the stabilizer projections $H^{(1)}$ and $H^{(2)}$. We then refine all partitions using the new information while maintaining the alignment invariants. For example, if $A_1(x) = y$, then any domain hyperplane $U \in \mathcal{H}_n$ containing x must be mapped to a hyperplane containing y , which can be used to split the domain-hyperplane partition further. More generally, refinement propagates back and forth between point and hyperplane partitions.

If any pair of corresponding ordered partitions differs in shape—that is, in the number of cells or the cardinalities of corresponding cells—then no consistent completion of Δ exists, and the branch can be pruned.

4 Signatures

Partition refinement aims to reduce cells to singletons where possible. Smaller cells leave fewer candidate images for the unknown maps and therefore reduce branching. We achieve this by assigning each object a signature derived from the current search state, with the requirement that corresponding objects have equal signatures under every consistent completion. Objects with different signature values cannot correspond under a consistent completion, so we refine each ordered partition by splitting its cells according to signature values. Crucially, the signatures are chosen so that they are invariant under any valid affine equivalence respecting Δ , ensuring that refinement preserves the alignment invariants from the previous section.

Definition 7 (Signatures). *Fix an integer $p \geq 1$. We will use four types of signatures, corresponding to the four kinds of objects in the search state, all taking values in $\mathbb{Z}_p$:*

$$\begin{aligned} \text{sig}_{\mathcal{P}} : \mathbb{F}_q^n &\rightarrow \mathbb{Z}_p && (\text{domain point signature}), \\ \text{sig}_{\mathcal{Q}} : \mathbb{F}_q^m &\rightarrow \mathbb{Z}_p && (\text{codomain point signature}), \\ \text{sig}_{\mathcal{L}} : \mathcal{H}_n &\rightarrow \mathbb{Z}_p && (\text{domain hyperplane signature}), \\ \text{sig}_{\mathcal{R}} : \mathcal{H}_m &\rightarrow \mathbb{Z}_p && (\text{codomain hyperplane signature}). \end{aligned}$$

We use a signature to split an ordered partition in the following way.

Definition 8 (Signature refinement of an ordered partition). *Let $\Pi = (B_1, \dots, B_k)$ be an ordered partition of a set X , and let $\text{sig} : X \rightarrow \mathbb{Z}_p$ be a signature. For $i \in \{1, \dots, k\}$ and $t \in \{0, 1, \dots, p-1\}$ set*

$$B_{i,t} := \{x \in B_i : \text{sig}(x) = t\},$$

which may be empty. The signature refinement is the ordered partition

$$\text{Ref}_{\text{sig}}(\Pi) := (B_{1,0}, \dots, B_{1,p-1}, B_{2,0}, \dots, B_{2,p-1}, \dots, B_{k,0}, \dots, B_{k,p-1}).$$

We keep empty cells only to ensure that partitions with the same signature values line up. In an implementation, one would naturally drop empty cells.

Definition 9 (Same shape). *Two ordered partitions $\Pi = (B_1, \dots, B_k)$ and $\Pi' = (B'_1, \dots, B'_{k'})$ are said to have the same shape if $k = k'$ and $|B_i| = |B'_i|$ for all $i \in \{1, \dots, k\}$.*

No bijection can map corresponding cells of ordered partitions with different shapes onto one another. Therefore, if the alignment invariant holds and a shape mismatch occurs, no consistent completion exists and the branch can be rejected.

We cannot refine the partitions of F and G using arbitrary signatures: we need the signatures to preserve the alignment invariants. This is a property of a pair of signatures $(\text{sig}_F, \text{sig}_G)$, since we refine F 's partition using sig_F and G 's partition using sig_G .

Definition 10 (Valid signature pairs). We say that a pair of signatures $(\text{sig}_F, \text{sig}_G)$ is valid for a given object type relative to Δ if the corresponding condition below holds for every consistent completion (A_1, A_2) extending the branch decisions Δ :

1. (Domain points) $\text{sig}_F, \text{sig}_G : \mathbb{F}_q^n \rightarrow \mathbb{Z}_p$ and $\text{sig}_F(x) = \text{sig}_G(A_1(x))$ for all $x \in \mathbb{F}_q^n$.
2. (Codomain points) $\text{sig}_F, \text{sig}_G : \mathbb{F}_q^m \rightarrow \mathbb{Z}_p$ and $\text{sig}_F(y) = \text{sig}_G(A_2(y))$ for all $y \in \mathbb{F}_q^m$.
3. (Domain hyperplanes) $\text{sig}_F, \text{sig}_G : \mathcal{H}_n \rightarrow \mathbb{Z}_p$ and $\text{sig}_F(U) = \text{sig}_G(A_1(U))$ for all $U \in \mathcal{H}_n$.
4. (Codomain hyperplanes) $\text{sig}_F, \text{sig}_G : \mathcal{H}_m \rightarrow \mathbb{Z}_p$ and $\text{sig}_F(V) = \text{sig}_G(A_2(V))$ for all $V \in \mathcal{H}_m$.

Signature validity depends on the branch decisions: a signature pair valid for one set of decisions need not be valid for another.

By definition, refinement using a valid signature pair preserves the alignment invariants (Definition 8).

Lemma 1 (Refinement preserves alignment). Let $(\text{sig}_F, \text{sig}_G)$ be a pair of signatures that is valid (for the relevant object type) relative to the current branch decisions Δ . If an aligned pair of ordered partitions is refined by refining each partition with its corresponding signature, then the corresponding alignment invariant is preserved. More precisely:

1. (Input points) If $(\mathcal{P}_F, \mathcal{P}_G)$ satisfies the *Input point-cell invariant* and we set $\mathcal{P}'_F := \text{Ref}_{\text{sig}_F}(\mathcal{P}_F)$ and $\mathcal{P}'_G := \text{Ref}_{\text{sig}_G}(\mathcal{P}_G)$, then $(\mathcal{P}'_F, \mathcal{P}'_G)$ also satisfies the *Input point-cell invariant*.
2. (Output points) If $(\mathcal{Q}_F, \mathcal{Q}_G)$ satisfies the *Output point-cell invariant* and we set $\mathcal{Q}'_F := \text{Ref}_{\text{sig}_F}(\mathcal{Q}_F)$ and $\mathcal{Q}'_G := \text{Ref}_{\text{sig}_G}(\mathcal{Q}_G)$, then $(\mathcal{Q}'_F, \mathcal{Q}'_G)$ also satisfies the *Output point-cell invariant*.
3. (Domain hyperplanes) If $(\mathcal{L}_F, \mathcal{L}_G)$ satisfies the *Domain hyperplane invariant* and we set $\mathcal{L}'_F := \text{Ref}_{\text{sig}_F}(\mathcal{L}_F)$ and $\mathcal{L}'_G := \text{Ref}_{\text{sig}_G}(\mathcal{L}_G)$, then $(\mathcal{L}'_F, \mathcal{L}'_G)$ also satisfies the *Domain hyperplane invariant*.
4. (Codomain hyperplanes) If $(\mathcal{R}_F, \mathcal{R}_G)$ satisfies the *Codomain hyperplane invariant* and we set $\mathcal{R}'_F := \text{Ref}_{\text{sig}_F}(\mathcal{R}_F)$ and $\mathcal{R}'_G := \text{Ref}_{\text{sig}_G}(\mathcal{R}_G)$, then $(\mathcal{R}'_F, \mathcal{R}'_G)$ also satisfies the *Codomain hyperplane invariant*.

This lemma follows directly from Definition 10.

5 Point and hyperplane signatures

We now introduce the specific signatures used by our algorithms. As discussed above, signature validity is defined relative to the current branch decisions Δ , so the signatures depend on the evolving search state. Our signatures are parameterized by one of the four ordered partitions in the search state: we use a partition (together with its alignment invariant) to define a valid signature pair for refining one of the other partitions. The first step is to assign a weight to each cell of an ordered partition.

Definition 11 (Weight functions). Let X be a finite set and let $\mathcal{B} = (B_1, \dots, B_t)$ be an ordered partition of X . Fix a weight vector $K = (k_1, \dots, k_t) \in \mathbb{Z}_p^t$. We define the weight function induced by $(\mathcal{B}, K)$ to be

$$w_{\mathcal{B}}^K : X \rightarrow \mathbb{Z}_p, \quad w_{\mathcal{B}}^K(x) = k_i \text{ where } x \in B_i.$$

Using weight functions, we define six kinds of signatures. We start with point signatures.

5.1 Point signatures

Let $p \in \mathbb{F}_q^n$ be any point, $\mathcal{L}$ a domain hyperplane partition, and let K be a weight vector for $\mathcal{L}$. We define the point signature

$$\text{sig}_{K, \mathcal{L}}^{(1)}(p) := \sum_{\substack{U \in \mathcal{H}_n \\ p \in U}} w_{\mathcal{L}}^K(U).$$

Similarly, for a point $q \in \mathbb{F}_q^m$, a codomain hyperplane partition $\mathcal{R}$, and a weight vector K for $\mathcal{R}$, we define

$$\text{sig}_{K, \mathcal{R}}^{(2)}(q) := \sum_{\substack{V \in \mathcal{H}_m \\ q \in V}} w_{\mathcal{R}}^K(V).$$

Thus, these signatures sum the weights of all hyperplanes containing the point. An equivalent way to write this is

$$\text{sig}_{K, \mathcal{L}}^{(1)}(p) = K \cdot (|\{U \in \mathcal{L}_1 : p \in U\}|, \dots, |\{U \in \mathcal{L}_t : p \in U\}|),$$

where $\mathcal{L} = (\mathcal{L}_1, \dots, \mathcal{L}_t)$.

Using the same weight vector on both sides ensures that these constructions yield a valid signature pair.

Lemma 2 (Point signature validity).

- (i) If a domain hyperplane partition pair $(\mathcal{L}_F, \mathcal{L}_G)$ satisfies the *Domain hyperplane invariant* relative to a set of branch decisions Δ , and K is an integer weight vector for $\mathcal{L}_F$ (equivalently $\mathcal{L}_G$), then $(\text{sig}_{K, \mathcal{L}_F}^{(1)}, \text{sig}_{K, \mathcal{L}_G}^{(1)})$ is a valid input-point signature pair relative to Δ .
- (ii) If a codomain hyperplane partition pair $(\mathcal{R}_F, \mathcal{R}_G)$ satisfies the *Codomain hyperplane invariant* relative to a set of branch decisions Δ , and K is an integer weight vector for $\mathcal{R}_F$ (equivalently $\mathcal{R}_G$), then $(\text{sig}_{K, \mathcal{R}_F}^{(2)}, \text{sig}_{K, \mathcal{R}_G}^{(2)})$ is a valid output-point signature pair relative to Δ .

Proof. (i) Let (A_1, A_2) be any consistent completion of Δ and $p \in \mathbb{F}_q^n$ any point. Then

$$\text{sig}_{K, \mathcal{L}_F}^{(1)}(p) = \sum_{\substack{U \in \mathcal{H}_n \\ p \in U}} w_{\mathcal{L}_F}^K(U) = \sum_{\substack{U \in \mathcal{H}_n \\ A_1(p) \in A_1(U)}} w_{\mathcal{L}_F}^K(U).$$

By the [Domain hyperplane invariant](#) we have that $w_{\mathcal{L}_F}^K(U) = w_{\mathcal{L}_G}^K(A_1(U))$, hence

$$= \sum_{\substack{U \in \mathcal{H}_n \\ A_1(p) \in A_1(U)}} w_{\mathcal{L}_G}^K(A_1(U)) = \text{sig}_{K, \mathcal{L}_G}^{(1)}(A_1(p)).$$

Thus the signature pair is valid relative to Δ by Definition 10.

(ii) The same argument applies with A_2 instead.

5.2 Hyperplane signatures

We use four hyperplane-signature families. The first two are analogous to the point signatures. Let $U \in \mathcal{H}_n$ be a domain hyperplane, $\mathcal{P}$ a domain point partition, and K an integer weight vector for $\mathcal{P}$. We define

$$\text{sig}_{K, \mathcal{P}}^{(3)}(U) := \sum_{p \in U} w_{\mathcal{P}}^K(p).$$

Similarly, for $V \in \mathcal{H}_m$, a codomain point partition $\mathcal{Q}$, and a weight vector K for $\mathcal{Q}$, we define

$$\text{sig}_{K, \mathcal{Q}}^{(4)}(V) := \sum_{q \in V} w_{\mathcal{Q}}^K(q).$$

These signatures sum the weights of the points in the hyperplane. We can also write this as

$$\text{sig}_{K, \mathcal{P}}^{(3)}(U) = K \cdot (|U \cap \mathcal{P}_1|, \dots, |U \cap \mathcal{P}_t|),$$

where $\mathcal{P} = (\mathcal{P}_1, \dots, \mathcal{P}_t)$ (and similarly for $\text{sig}^{(4)}$ with $\mathcal{Q}$).

Lemma 3 (Validity of $\text{sig}^{(3)}$ and $\text{sig}^{(4)}$).

- (i) If a domain point partition pair $(\mathcal{P}_F, \mathcal{P}_G)$ satisfies the [Input point-cell invariant](#) relative to a set of branch decisions Δ , and K is an integer weight vector for $\mathcal{P}_F$ (equivalently $\mathcal{P}_G$), then $(\text{sig}_{K, \mathcal{P}_F}^{(3)}, \text{sig}_{K, \mathcal{P}_G}^{(3)})$ is a valid domain-hyperplane signature pair relative to Δ .
- (ii) If a codomain point partition pair $(\mathcal{Q}_F, \mathcal{Q}_G)$ satisfies the [Output point-cell invariant](#) relative to a set of branch decisions Δ , and K is an integer weight vector for $\mathcal{Q}_F$ (equivalently $\mathcal{Q}_G$), then $(\text{sig}_{K, \mathcal{Q}_F}^{(4)}, \text{sig}_{K, \mathcal{Q}_G}^{(4)})$ is a valid codomain-hyperplane signature pair relative to Δ .

Proof. (i) Let (A_1, A_2) be any consistent completion of Δ and let $U \in \mathcal{H}_n$. Then

$$\text{sig}_{K, \mathcal{P}_F}^{(3)}(U) = \sum_{p \in U} w_{\mathcal{P}_F}^K(p)$$

due to the [Input point-cell invariant](#).

$$= \sum_{p \in U} w_{\mathcal{P}_G}^K(A_1(p)) = \sum_{q \in A_1(U)} w_{\mathcal{P}_G}^K(q) = \text{sig}_{K, \mathcal{P}_G}^{(3)}(A_1(U)).$$

(ii) The same argument applies with A_2 instead.

The first four signature families propagate information within the domain or within the codomain. The next two use the function to propagate information between the domain and codomain.

Let $\mathcal{R}$ be a codomain hyperplane partition, $U \in \mathcal{H}_n$, and let K be a weight vector for $\mathcal{R}$. Define

$$\text{sig}_{K,\mathcal{R}}^{(5)}(U) := \sum_{V \in \mathcal{H}_m} w_{\mathcal{R}}^K(V) \cdot |U \cap F^{-1}(V)|.$$

Similarly, let $\mathcal{L}$ be a domain hyperplane partition, $V \in \mathcal{H}_m$, and let K be a weight vector for $\mathcal{L}$. Define

$$\text{sig}_{K,\mathcal{L}}^{(6)}(V) := \sum_{U \in \mathcal{H}_n} w_{\mathcal{L}}^K(U) \cdot |F^{-1}(V) \cap U|.$$

Intuitively, $\text{sig}^{(5)}$ is a *transport* signature: it uses the function F to convert information about codomain hyperplanes into a number attached to a domain hyperplane.

Fix a domain hyperplane U . For each codomain hyperplane V , look at the fiber slice $U \cap F^{-1}(V) = \{p \in U : F(p) \in V\}$. The term $|U \cap F^{-1}(V)|$ counts how many points of U are mapped by F into V . We then take a weighted sum of these counts. The validity of these signatures uses the identity $A_2 \circ F = G \circ A_1$, which was not needed in the validity proofs for the first four signature families.

Lemma 4 (Validity of $\text{sig}^{(5)}$ and $\text{sig}^{(6)}$).

- (i) If $(\mathcal{R}_F, \mathcal{R}_G)$ satisfies the *Codomain hyperplane invariant* relative to a set of branch decisions Δ and K is a weight vector, then $(\text{sig}_{K,\mathcal{R}_F}^{(5)}, \text{sig}_{K,\mathcal{R}_G}^{(5)})$ is a valid domain-hyperplane signature pair relative to Δ .
- (ii) If $(\mathcal{L}_F, \mathcal{L}_G)$ satisfies the *Domain hyperplane invariant* relative to a set of branch decisions Δ and K is a weight vector, then $(\text{sig}_{K,\mathcal{L}_F}^{(6)}, \text{sig}_{K,\mathcal{L}_G}^{(6)})$ is a valid codomain-hyperplane signature pair relative to Δ .

Proof. (i) Let $(\mathcal{R}_F, \mathcal{R}_G)$ satisfy the *Codomain hyperplane invariant* relative to Δ , and let (A_1, A_2) be any consistent completion of Δ . For $U \in \mathcal{H}_n$,

$$\text{sig}_{K,\mathcal{R}_F}^{(5)}(U) = \sum_{V \in \mathcal{H}_m} w_{\mathcal{R}_F}^K(V) \cdot |U \cap F^{-1}(V)|$$

using the fact that A_1 is a permutation

$$= \sum_{V \in \mathcal{H}_m} w_{\mathcal{R}_F}^K(V) \cdot |A_1(U) \cap A_1(F^{-1}(V))|$$

Since $A_2 \circ F = G \circ A_1$, we have $A_1(F^{-1}(V)) = G^{-1}(A_2(V))$, so

$$= \sum_{V \in \mathcal{H}_m} w_{\mathcal{R}_F}^K(V) \cdot |A_1(U) \cap G^{-1}(A_2(V))|$$

By the *Codomain hyperplane invariant*, we have $w_{\mathcal{R}_F}^K(V) = w_{\mathcal{R}_G}^K(A_2(V))$, so

$$\begin{aligned}
 &= \sum_{V \in \mathcal{H}_m} w_{\mathcal{R}_G}^K(A_2(V)) \cdot |A_1(U) \cap G^{-1}(A_2(V))| \\
 &= \text{sig}_{\mathcal{R}_G}^{(5)}(A_1(U)).
 \end{aligned}$$

(ii) The proof is analogous to part (i).

5.3 Cost of direct signature computation

Now let us consider how to compute these six types of signatures. We could of course use the definition directly, so for example just compute the sum $\sum_{\substack{U \in \mathcal{H}_n \\ p \in U}} w_{\mathcal{L}}^K(U)$ for each point p to compute $\text{sig}^{(1)}$, but the time complexity of this would be around $\mathcal{O}(q^{2n})$. The same is true for the other signatures. However, using the fast Fourier transform we can compute $\text{sig}^{(1)}$ in time closer to $\mathcal{O}(q^n)$ (ignoring polynomial factors), which is a significant improvement as computing the signatures is generally the most computationally expensive part of the algorithms. We can obtain the same kind of speedup for the other signatures.

5.4 Radon transform

First we need some basic concepts from Fourier analysis.

Definition 12 (Fourier transform on $\mathbb{F}_q^d$). Fix the standard additive character

$$\psi(t) := \exp\left(\frac{2\pi i}{p} \text{Tr}_{\mathbb{F}_q/\mathbb{F}_p}(t)\right), \quad t \in \mathbb{F}_q.$$

For $\alpha \in \mathbb{F}_q^d$ define $\chi_\alpha(x) := \psi(\alpha \cdot x)$. For $f : \mathbb{F}_q^d \rightarrow \mathbb{C}$ define its Fourier transform by

$$\hat{f}(\alpha) := \sum_{x \in \mathbb{F}_q^d} f(x) \psi(-\alpha \cdot x), \quad \alpha \in \mathbb{F}_q^d,$$

and the inverse transform by

$$f(x) = \frac{1}{q^d} \sum_{\alpha \in \mathbb{F}_q^d} \hat{f}(\alpha) \psi(\alpha \cdot x), \quad x \in \mathbb{F}_q^d.$$

It is well known that both the forward and inverse transforms can be computed for all inputs in time $\mathcal{O}(q^d \log(q^d))$; see, for example, [19].

Definition 13 (Radon transform and backprojection on affine hyperplanes in $\mathbb{F}_q^d$). For $f : \mathbb{F}_q^d \rightarrow \mathbb{C}$ define its (hyperplane) Radon transform $\mathcal{T}f : \mathcal{H}_d \rightarrow \mathbb{C}$ by

$$(\mathcal{T}f)(H) := \sum_{x \in H} f(x), \quad H \in \mathcal{H}_d.$$

For $g : \mathcal{H}_d \rightarrow \mathbb{C}$ define the adjoint (backprojection) $\mathcal{T}^*g : \mathbb{F}_q^d \rightarrow \mathbb{C}$ by

$$(\mathcal{T}^*g)(x) := \sum_{H \in \mathcal{H}_d : x \in H} g(H), \quad x \in \mathbb{F}_q^d.$$

Using this, we can compute both the Radon transform and the backprojection efficiently due to the following facts.

Lemma 5. *For a function $f : \mathbb{F}_q^d \rightarrow \mathbb{C}$ and hyperplane $H_{a,b} = \{x \in \mathbb{F}_q^d : a \cdot x = b\}$*

$$(\mathcal{T}f)(H_{a,b}) = \frac{1}{q} \sum_{t \in \mathbb{F}_q} \widehat{f}(ta) \psi(tb), \quad b \in \mathbb{F}_q.$$

Lemma 6. *For $g : \mathcal{H}_d \rightarrow \mathbb{C}$, $a \in \mathbb{F}_q^d \setminus \{0\}$, and $t \in \mathbb{F}_q^\times$ we have*

$$\widehat{\mathcal{T}^*g}(ta) = q^{d-1} \sum_{b \in \mathbb{F}_q} g(H_{a,b}) \psi(-tb).$$

5.5 Computing signatures with Radon transforms

We now have everything needed to express our signatures using compositions of the Radon transform, backprojection, pushforward, and pullback.

Definition 14. *Let $F : \mathbb{F}_q^n \rightarrow \mathbb{F}_q^m$. For a function $w : \mathbb{F}_q^n \rightarrow \mathbb{C}$, define the pushforward along F by*

$$(F_{\rightarrow}w)(y) = \sum_{x:F(x)=y} w(x),$$

and for a function $h : \mathbb{F}_q^m \rightarrow \mathbb{C}$, define the pullback by

$$(F_{\leftarrow}h)(x) = h(F(x)).$$

The following lemma expresses all six signature families in terms of these operators, allowing efficient computation.

Lemma 7 (Radon transform realizes signatures). *Each of the six signature families can be expressed using Radon transforms and backprojections of appropriate weight functions, together with pushforward or pullback where needed. In particular,*

$$\begin{aligned} \text{sig}_{K,\mathcal{L}}^{(1)}(p) &= (\mathcal{T}^*w_{\mathcal{L}}^K)(p), \\ \text{sig}_{K,\mathcal{R}}^{(2)}(q) &= (\mathcal{T}^*w_{\mathcal{R}}^K)(q), \\ \text{sig}_{K,\mathcal{P}}^{(3)}(U) &= (\mathcal{T}w_{\mathcal{P}}^K)(U), \\ \text{sig}_{K,\mathcal{Q}}^{(4)}(V) &= (\mathcal{T}w_{\mathcal{Q}}^K)(V), \\ \text{sig}_{K,\mathcal{R}}^{(5)}(U) &= \mathcal{T}\left(F_{\leftarrow}(\mathcal{T}^*w_{\mathcal{R}}^K)\right)(U), \\ \text{sig}_{K,\mathcal{L}}^{(6)}(V) &= \mathcal{T}\left(F_{\rightarrow}(\mathcal{T}^*w_{\mathcal{L}}^K)\right)(V). \end{aligned}$$

Proof. We prove each identity by expanding definitions.

(1) **Points via $\mathcal{L}$.** For $p \in \mathbb{F}_q^n$,

$$\text{sig}_{K,\mathcal{L}}^{(1)}(p) = \sum_{\substack{U \in \mathcal{H}_n \\ p \in U}} w_{\mathcal{L}}^K(U) = (\mathcal{T}^* w_{\mathcal{L}}^K)(p).$$

(2) **Codomain points via $\mathcal{R}$.** For $q \in \mathbb{F}_q^m$,

$$\text{sig}_{K,\mathcal{R}}^{(2)}(q) = \sum_{\substack{V \in \mathcal{H}_m \\ q \in V}} w_{\mathcal{R}}^K(V) = (\mathcal{T}^* w_{\mathcal{R}}^K)(q).$$

(3) **Domain hyperplanes via $\mathcal{P}$.** For $U \in \mathcal{H}_n$,

$$\text{sig}_{K,\mathcal{P}}^{(3)}(U) = \sum_{p \in U} w_{\mathcal{P}}^K(p) = (\mathcal{T} w_{\mathcal{P}}^K)(U).$$

(4) **Codomain hyperplanes via $\mathcal{Q}$.** For $V \in \mathcal{H}_m$,

$$\text{sig}_{K,\mathcal{Q}}^{(4)}(V) = \sum_{q \in V} w_{\mathcal{Q}}^K(q) = (\mathcal{T} w_{\mathcal{Q}}^K)(V).$$

(5) **Domain hyperplanes via $\mathcal{R}$.** For $U \in \mathcal{H}_n$,

$$\begin{aligned} \text{sig}_{K,\mathcal{R}}^{(5)}(U) &= \sum_{V \in \mathcal{H}_m} w_{\mathcal{R}}^K(V) \cdot |U \cap F^{-1}(V)| = \sum_{p \in U} \sum_{\substack{V \in \mathcal{H}_m \\ F(p) \in V}} w_{\mathcal{R}}^K(V) \\ &= \sum_{p \in U} (\mathcal{T}^* w_{\mathcal{R}}^K)(F(p)) = \sum_{p \in U} (F_{\leftarrow}(\mathcal{T}^* w_{\mathcal{R}}^K))(p) = \mathcal{T}(F_{\leftarrow}(\mathcal{T}^* w_{\mathcal{R}}^K))(U). \end{aligned}$$

(6) **Codomain hyperplanes via $\mathcal{L}$.** For $V \in \mathcal{H}_m$,

$$\begin{aligned} \text{sig}_{K,\mathcal{L}}^{(6)}(V) &= \sum_{U \in \mathcal{H}_n} w_{\mathcal{L}}^K(U) \cdot |F^{-1}(V) \cap U| = \sum_{F(p) \in V} \sum_{\substack{U \in \mathcal{H}_n \\ p \in U}} w_{\mathcal{L}}^K(U) \\ &= \sum_{F(p) \in V} (\mathcal{T}^* w_{\mathcal{L}}^K)(p) = \sum_{q \in V} \sum_{p: F(p)=q} (\mathcal{T}^* w_{\mathcal{L}}^K)(p) = \sum_{q \in V} (F_{\rightarrow}(\mathcal{T}^* w_{\mathcal{L}}^K))(q) \\ &= \mathcal{T}(F_{\rightarrow}(\mathcal{T}^* w_{\mathcal{L}}^K))(V). \end{aligned}$$

The following algorithm computes all six signature families efficiently from the ordered partitions $(\mathcal{P}, \mathcal{Q}, \mathcal{L}, \mathcal{R})$.

Algorithm 1: Simultaneous computation of the six signature families from a tuple of ordered partitions $(\mathcal{P}, \mathcal{Q}, \mathcal{L}, \mathcal{R})$ and a tuple of weight vectors (K_P, K_Q, K_L, K_R) (one for each partition).

Input: A tuple of ordered partitions $(\mathcal{P}, \mathcal{Q}, \mathcal{L}, \mathcal{R})$ and weight vectors (K_P, K_Q, K_L, K_R) .

Output: The six signatures $(\text{sig}_{K_L, \mathcal{L}}^{(1)}(p))_{p \in \mathbb{F}_q^n}, (\text{sig}_{K_R, \mathcal{R}}^{(2)}(q))_{q \in \mathbb{F}_q^m},$
 $(\text{sig}_{K_P, \mathcal{P}}^{(3)}(U))_{U \in \mathcal{H}_n}, (\text{sig}_{K_Q, \mathcal{Q}}^{(4)}(V))_{V \in \mathcal{H}_m}, (\text{sig}_{K_R, \mathcal{R}}^{(5)}(U))_{U \in \mathcal{H}_n},$
 $(\text{sig}_{K_L, \mathcal{L}}^{(6)}(V))_{V \in \mathcal{H}_m}.$

```

 $\text{sig}_{K_L, \mathcal{L}}^{(1)} \leftarrow \mathcal{T}^* w_{\mathcal{L}}^{K_L}$ 
 $\text{sig}_{K_R, \mathcal{R}}^{(2)} \leftarrow \mathcal{T}^* w_{\mathcal{R}}^{K_R}$ 
 $\text{sig}_{K_P, \mathcal{P}}^{(3)} \leftarrow \mathcal{T} w_{\mathcal{P}}^{K_P}$ 
 $\text{sig}_{K_Q, \mathcal{Q}}^{(4)} \leftarrow \mathcal{T} w_{\mathcal{Q}}^{K_Q}$ 
foreach  $x \in \mathbb{F}_q^n$  do
   $\text{pullback}[x] \leftarrow \text{sig}_{K_R, \mathcal{R}}^{(2)}(F(x))$ 
 $\text{sig}_{K_R, \mathcal{R}}^{(5)} \leftarrow \mathcal{T} \text{pullback}$ 
foreach  $y \in \mathbb{F}_q^m$  do
   $\text{pushforward}[y] \leftarrow 0$ 
foreach  $x \in \mathbb{F}_q^n$  do
   $y \leftarrow F(x)$ 
   $\text{pushforward}[y] \leftarrow \text{pushforward}[y] + \text{sig}_{K_L, \mathcal{L}}^{(1)}(x)$ 
 $\text{sig}_{K_L, \mathcal{L}}^{(6)} \leftarrow \mathcal{T} \text{pushforward}$ 

```

6 Orbits

Our algorithms use known self-equivalences to reduce the search trees. Candidate images in the same orbit under the current stabilizer yield equivalent branches, so only one representative per orbit needs to be explored. We first consider point branches.

Lemma 8 (Orbit pruning and transport of point completions). *Let $F, G : \mathbb{F}_q^n \rightarrow \mathbb{F}_q^m$, let Δ be the current branch decisions, and let $H \leq \text{Aut}(G)$.*

Then, for every $\phi = (\phi_1, \phi_2) \in H_\Delta$, the following hold.

1. *For all $x, y \in \mathbb{F}_q^n$,*

$$\phi \cdot (\mathcal{C}(\Delta \cup \{A_1(x) = y\})) = \mathcal{C}(\Delta \cup \{A_1(x) = \phi_1(y)\})$$

2. *For all $z, w \in \mathbb{F}_q^m$,*

$$\phi \cdot (\mathcal{C}(\Delta \cup \{A_2(z) = w\})) = \mathcal{C}(\Delta \cup \{A_2(z) = \phi_2(w)\})$$

Proof. We prove part (1); part (2) is analogous. Let $\Delta = (\Delta_{\text{pt}}^{(1)}, \Delta_{\text{hyp}}^{(1)}, \Delta_{\text{pt}}^{(2)}, \Delta_{\text{hyp}}^{(2)})$, $\phi = (\phi_1, \phi_2) \in H_\Delta$, and $x, y \in \mathbb{F}_q^n$.

First let $(\widehat{A}_1, \widehat{A}_2) \in \phi \cdot (\mathcal{C}(\Delta \cup \{A_1(x) = y\}))$. So

$$(\widehat{A}_1, \widehat{A}_2) = (\phi_1 \circ A_1, \phi_2 \circ A_2),$$

where (A_1, A_2) is a consistent completion of $\Delta \cup \{A_1(x) = y\}$. We have that

$$(\phi_2 \circ A_2) \circ F = G \circ (\phi_1 \circ A_1).$$

Since $\phi \in H_\Delta$ we see that ϕ_1 fixes every object in $\Delta_{\text{pt}}^{(1)}$ and $\Delta_{\text{hyp}}^{(1)}$, and the same is true for ϕ with respect to $\Delta_{\text{pt}}^{(2)}$ and $\Delta_{\text{hyp}}^{(2)}$. This means that $\phi \cdot (A_1, A_2) \in \mathcal{C}(\Delta)$. Since $A_1(x) = y$ we also know that $\phi_1 \circ A_1(x) = \phi_1(y)$, so we get that $\phi \cdot (A_1, A_2) \in \mathcal{C}(\Delta \cup \{A_1(x) = \phi_1(y)\})$.

Conversely, suppose $(A_1, A_2) \in \mathcal{C}(\Delta \cup \{A_1(x) = \phi_1(y)\})$. Then define

$$(\widetilde{A}_1, \widetilde{A}_2) := (\phi_1^{-1} \circ A_1, \phi_2^{-1} \circ A_2).$$

Since $\phi \in H_\Delta$, its inverse ϕ^{-1} also fixes every object in $\Delta_{\text{pt}}^{(1)}$ and $\Delta_{\text{hyp}}^{(1)}$ and likewise for $\Delta_{\text{pt}}^{(2)}$ and $\Delta_{\text{hyp}}^{(2)}$, hence $(\widetilde{A}_1, \widetilde{A}_2) \in \mathcal{C}(\Delta)$. Finally,

$$\widetilde{A}_1(x) = \phi_1^{-1}(A_1(x)) = \phi_1^{-1}(\phi_1(y)) = y,$$

so $(\widetilde{A}_1, \widetilde{A}_2) \in \mathcal{C}(\Delta \cup \{A_1(x) = y\})$. Therefore

$$(A_1, A_2) = \phi \cdot (\widetilde{A}_1, \widetilde{A}_2) \in \phi \cdot \mathcal{C}(\Delta \cup \{A_1(x) = y\}),$$

which completes the proof.

The analogous statements hold for hyperplane branches.

Lemma 9 (Orbit pruning and transport of hyperplane completions).

Let $F, G : \mathbb{F}_q^n \rightarrow \mathbb{F}_q^m$, let Δ be the current branch decisions, and let $H \leq \text{Aut}(G)$.

Then, for every $\phi = (\phi_1, \phi_2) \in H_\Delta$, the following hold.

1. For all $U, V \in \mathcal{H}_n$,

$$\phi \cdot (\mathcal{C}(\Delta \cup \{A_1(U) = V\})) = \mathcal{C}(\Delta \cup \{A_1(U) = \phi_1(V)\})$$

2. For all $W, X \in \mathcal{H}_m$,

$$\phi \cdot (\mathcal{C}(\Delta \cup \{A_2(W) = X\})) = \mathcal{C}(\Delta \cup \{A_2(W) = \phi_2(X)\})$$

In our algorithm we will keep two separate subgroups in addition to the subgroup of the affine self-equivalence group. These are $H^{(1)} = \pi_1(H)$ and $H^{(2)} = \pi_2(H)$. If we for example are going to branch on the values of A_1 at a point x , and find that y and y' are in the same orbit under $\pi_1(H_\Delta)$, then by Lemma 8 if $\phi = (\phi_1, \phi_2)$ is the witness such that $\phi_1(y) = y'$ we have that for any consistent extension of Δ , (A_1, A_2) where $A_1(x) = y$, there exists a consistent extension $(\widehat{A}_1, \widehat{A}_2)$ also of Δ where $\widehat{A}_1(x) = y'$ and where $(\widehat{A}_1, \widehat{A}_2) = \phi \cdot (A_1, A_2)$. For our self-equivalence algorithm it means that it is enough to find $\mathcal{C}(\Delta \cup \{A_1(x) = y\})$, and then just multiply with H_Δ . For equivalence search, if one representative branch has no consistent completion, then no other branch in the same orbit has a consistent completion.

6.1 Computing the stabilizer

Now for a group $H \leq \text{Aut}(G)$ we must be able to actually compute H_Δ for a set of branch decisions Δ . We will not use the entire subgroup $H \leq \text{Aut}(G)$, but instead the two parts $H^{(1)} = \pi_1(H)$ and $H^{(2)} = \pi_2(H)$. So these are what we use for orbit pruning. We need to be able to efficiently compute H_Δ however as we need to update the groups $H^{(1)}$ and $H^{(2)}$. To do this, we will use standard methods from computational group theory, namely stabilizer chains (BSGS) and the Schreier–Sims algorithm [32, Chapter 4]. We will not go into detail on this here, but this allows us to compute pointwise stabilizer subgroups efficiently. This means that for all the point decisions $A_1(x) = y$ in Δ if $H' = H_\Delta$ we will simply compute $H^{(1)} = \pi_1(H'_y)$, and likewise for point decisions $A_2(x) = y$ we compute $H^{(2)} = \pi_2(H'_y)$. Now for hyperplanes we want the setwise stabilizer, which can be a bit more computationally expensive to compute, so we rely on the following well-known result called Schreier’s lemma, see for example Chapter 4 in [32].

Lemma 10 (Stabilizer of a hyperplane). *Let $H = \langle S \rangle \leq \text{AGL}(d, q)$ act on the set $\mathcal{H}_d$ of affine hyperplanes in $\mathbb{F}_q^d$. Fix $U_0 \in \mathcal{H}_d$. For each $U \in H \cdot U_0$, choose an element $\tau_U \in H$ such that*

$$\tau_U(U_0) = U,$$

with $\tau_{U_0} = \text{id}$. Then the setwise stabilizer of U_0 in H is

$$H_{U_0} = \langle \tau_\sigma^{-1} \sigma \tau_U : U \in H \cdot U_0, \sigma \in S \cup S^{-1} \rangle.$$

This lemma provides an explicit generating set for the setwise stabilizer H_{U_0} , which we use to compute H_Δ .

7 The equivalence and self-equivalence algorithms

The refinement algorithm repeatedly applies the signatures computed by Algorithm 1 until the partitions stop changing.

Algorithm 2: Refine the ordered partitions until no more refinement occurs, or both A_1 and A_2 can be determined

Input: Domain point partitions $(\mathcal{P}_F, \mathcal{P}_G)$, codomain point partitions $(\mathcal{Q}_F, \mathcal{Q}_G)$, domain hyperplane partitions $(\mathcal{L}_F, \mathcal{L}_G)$, and codomain hyperplane partitions $(\mathcal{R}_F, \mathcal{R}_G)$
Output: Refined partitions $(\mathcal{P}'_F, \mathcal{P}'_G)$, $(\mathcal{Q}'_F, \mathcal{Q}'_G)$, $(\mathcal{L}'_F, \mathcal{L}'_G)$, $(\mathcal{R}'_F, \mathcal{R}'_G)$, or **false** if any partition pair mismatches in shape

repeat
 Choose K_P, K_Q, K_L, K_R
 Compute all signatures for F and G using Algorithm 1
 $P_F \leftarrow \text{Ref}_{\text{sig}_{K_L, L_F}}^{(1)}(P_F)$, $P_G \leftarrow \text{Ref}_{\text{sig}_{K_L, L_G}}^{(1)}(P_G)$
 $Q_F \leftarrow \text{Ref}_{\text{sig}_{K_R, R_F}}^{(2)}(Q_F)$, $Q_G \leftarrow \text{Ref}_{\text{sig}_{K_R, R_G}}^{(2)}(Q_G)$
 $L_F \leftarrow \text{Ref}_{\text{sig}_{K_P, P_F}}^{(3)}(L_F)$, $L_G \leftarrow \text{Ref}_{\text{sig}_{K_P, P_G}}^{(3)}(L_G)$
 $L_F \leftarrow \text{Ref}_{\text{sig}_{K_R, R_F}}^{(5)}(L_F)$, $L_G \leftarrow \text{Ref}_{\text{sig}_{K_R, R_G}}^{(5)}(L_G)$
 $R_F \leftarrow \text{Ref}_{\text{sig}_{K_Q, Q_F}}^{(4)}(R_F)$, $R_G \leftarrow \text{Ref}_{\text{sig}_{K_Q, Q_G}}^{(4)}(R_G)$
 $R_F \leftarrow \text{Ref}_{\text{sig}_{K_L, L_F}}^{(6)}(R_F)$, $R_G \leftarrow \text{Ref}_{\text{sig}_{K_L, L_G}}^{(6)}(R_G)$
 if some pair of partitions does not have the same shape **then**
 | **return false**
 if the current singleton cells determine A_1 **then**
 | Individualize (P_F, P_G) according to A_1
 if the current singleton cells determine A_2 **then**
 | Individualize (Q_F, Q_G) according to A_2
until no partition changes
return $(P_F, P_G), (Q_F, Q_G), (L_F, L_G), (R_F, R_G)$

A round may leave every partition unchanged even when another choice of weight vectors would produce further refinement.

As discussed above, our overall strategy is a depth-first search (DFS). At each search node we choose an object type to branch on—domain points, codomain points, domain hyperplanes, or codomain hyperplanes. We then choose a non-singleton cell in the current partitions and select a distinguished *source* element on one side. Rather than branching on every element of the corresponding cell on the other side, we branch only on one representative from each orbit (under the current stabilizer) acting on that cell. For each such representative, we create a child node by individualizing both partitions so as to commit to the tentative pairing of the source with that representative, run the refinement procedure, and recurse. If refinement detects an inconsistency—for example, if the resulting partition shapes/cardinalities on the two sides no longer agree—then no consistent completion exists along that branch, so we prune it and backtrack.

As explained in Section 3, both algorithms maintain a subgroup $H \leq \text{Aut}(G)$. At a search node with branch decisions Δ , the projections $H^{(1)} = \pi_1(H_\Delta)$ and

$H^{(2)} = \pi_2(H_\Delta)$ are used for orbit pruning as described in Section 6. In the equivalence algorithm, H is supplied as input and may be nontrivial. In the self-equivalence algorithm, where $G = F$, H is the subgroup generated by the self-equivalences found so far; only this algorithm rejects a completed solution that already belongs to H . Unless a subgroup is supplied in advance, the self-equivalence search starts with the trivial group, so its two stabilizer projections are initially trivial as well.

Algorithm 3: $\text{DFS}(\Delta, (P_F, P_G), (Q_F, Q_G), (L_F, L_G), (R_F, R_G), H^{(1)}, H^{(2)}, H)$

Input: Branch decisions Δ , paired partitions $(P_F, P_G), (Q_F, Q_G), (L_F, L_G), (R_F, R_G)$, local stabilizers $H^{(1)}, H^{(2)}$, and $H \leq \text{Aut}(G)$.
Output: A consistent completion $(A_1, A_2) \notin H$, or **false** if no such (A_1, A_2) exists.

$S \leftarrow$ output of Algorithm 2 on $(P_F, P_G), (Q_F, Q_G), (L_F, L_G), (R_F, R_G)$
if $S = \text{false}$ **then**
 return false
 $((P_F, P_G), (Q_F, Q_G), (L_F, L_G), (R_F, R_G)) \leftarrow S$
if Δ determines A_1 and A_2 **then**
 if $A_2 \circ F \neq G \circ A_1$ **then**
 return false
 if $(A_1, A_2) \in H$ **then**
 // This step is not done for the equivalence algorithm
 return false
 return (A_1, A_2)
Choose an unresolved cell of smallest size among P_F, Q_F, L_F, R_F
if the chosen cell is P_i^F **then**
 choose $x \in P_i^F$
 $\mathcal{B} \leftarrow \{A_1(x) = y : y \in \text{Rep}(P_i^G/H^{(1)})\}$
if the chosen cell is Q_i^F **then**
 choose $x \in Q_i^F$
 $\mathcal{B} \leftarrow \{A_2(x) = y : y \in \text{Rep}(Q_i^G/H^{(2)})\}$
if the chosen cell is L_i^F **then**
 choose $U \in L_i^F$
 $\mathcal{B} \leftarrow \{A_1(U) = V : V \in \text{Rep}(L_i^G/H^{(1)})\}$
if the chosen cell is R_i^F **then**
 choose $U \in R_i^F$
 $\mathcal{B} \leftarrow \{A_2(U) = V : V \in \text{Rep}(R_i^G/H^{(2)})\}$
for $\delta \in \mathcal{B}$ **do**
 $(\tilde{P}_F, \tilde{P}_G), (\tilde{Q}_F, \tilde{Q}_G), (\tilde{L}_F, \tilde{L}_G), (\tilde{R}_F, \tilde{R}_G) \leftarrow$ individualize the current partitions according to δ
 $(\tilde{H}^{(1)}, \tilde{H}^{(2)}) \leftarrow (\pi_1(H_{\Delta \cup \{\delta\}}), \pi_2(H_{\Delta \cup \{\delta\}}))$
 $B \leftarrow \text{DFS}(\Delta \cup \{\delta\}, (\tilde{P}_F, \tilde{P}_G), (\tilde{Q}_F, \tilde{Q}_G), (\tilde{L}_F, \tilde{L}_G), (\tilde{R}_F, \tilde{R}_G), \tilde{H}^{(1)}, \tilde{H}^{(2)}, H)$
 if $B \neq \text{false}$ **then**
 return B
return false

The equivalence and self-equivalence algorithms share the same search procedure. In self-equivalence search, we reject solutions already contained in H . For both equivalence and self-equivalence we use H to update the local stabilizer subgroups when necessary.

Algorithm 4: Affine equivalence algorithm**Input:** Functions $F, G : \mathbb{F}_q^n \rightarrow \mathbb{F}_q^m$, and a subgroup $H \leq \text{Aut}(G)$.**Output:** An affine equivalence (A_1, A_2) from F to G , or **false**.

$$\Delta \leftarrow (\emptyset, \emptyset, \emptyset, \emptyset)$$

$$(P_F, P_G) \leftarrow ((\mathbb{F}_q^n,), (\mathbb{F}_q^n,))$$

$$(Q_F, Q_G) \leftarrow ((\mathbb{F}_q^m,), (\mathbb{F}_q^m,))$$

$$(L_F, L_G) \leftarrow ((\mathcal{H}_n,), (\mathcal{H}_n,))$$

$$(R_F, R_G) \leftarrow ((\mathcal{H}_m,), (\mathcal{H}_m,))$$

$$H^{(1)} \leftarrow \pi_1(H), \quad H^{(2)} \leftarrow \pi_2(H)$$
return DFS($\Delta, (P_F, P_G), (Q_F, Q_G), (L_F, L_G), (R_F, R_G), H^{(1)}, H^{(2)}, H$)

Algorithm 5: Affine self-equivalence algorithm**Input:** A function $F : \mathbb{F}_q^n \rightarrow \mathbb{F}_q^m$.**Output:** The affine self-equivalence group $\text{Aut}(F)$, represented by a generating set.

$$H \leftarrow \langle (\text{id}_{\mathbb{F}_q^n}, \text{id}_{\mathbb{F}_q^m}) \rangle$$
repeat

$$\Delta \leftarrow \emptyset$$

$$(P_F, P_G) \leftarrow ((\mathbb{F}_q^n,), (\mathbb{F}_q^n,))$$

$$(Q_F, Q_G) \leftarrow ((\mathbb{F}_q^m,), (\mathbb{F}_q^m,))$$

$$(L_F, L_G) \leftarrow ((\mathcal{H}_n,), (\mathcal{H}_n,))$$

$$(R_F, R_G) \leftarrow ((\mathcal{H}_m,), (\mathcal{H}_m,))$$

$$H^{(1)} \leftarrow \pi_1(H), \quad H^{(2)} \leftarrow \pi_2(H)$$

$$B \leftarrow \text{DFS}(\Delta, (P_F, P_G), (Q_F, Q_G),$$

$$(L_F, L_G), (R_F, R_G), H^{(1)}, H^{(2)}, H)$$
if $B \neq \text{false}$ **then**

$$\quad H \leftarrow \langle H, B \rangle$$
until $B = \text{false}$
return H

7.1 Linear equivalence

All our results and algorithms directly apply to linear equivalence. We only need to check that both A_1 and A_2 are linear permutations before returning the pair. We can, however, do slightly better by taking advantage of the fact that A_1 and A_2 must be linear. This is simply done by adding $\{A_1(0) = 0\}$ and $\{A_2(0) = 0\}$ to the initial set of branch decisions. We also have the fact that $A_2(F(0)) = G(0)$ which adds another element to the initial set of branch decisions if $F(0) \neq 0$. After adding these initial branch decisions we just run the search as normal for both the equivalence and self-equivalence algorithms.

8 Experimental results

We now present runtime experiments comparing several algorithms for affine and linear equivalence. For ease of reference, we use the following names:

- AFFINEEQ-BACKFORTH and LINEAREQ-BACKFORTH: the affine and linear equivalence algorithms of [8]. We use the implementation from [30].
- AFFINEEQ-RANKTABLE: the affine equivalence algorithm of [20]. This is only reliable for random functions so we only test it for those.
- AFFINEEQ-OURS and LINEAREQ-OURS: our affine and linear equivalence algorithms. The reported runtimes include precomputation of the target function’s self-equivalence group for the corresponding equivalence relation.
- CCZEQ and CCZSELFEQ: the CCZ equivalence and self-equivalence algorithms from [25]. The reported CCZEQ runtimes include precomputation of the target function’s CCZ self-equivalence group. The available implementation supports only dimensions $n < 16$, as the memory requirements become very large at higher dimensions.

Timing notation. In the tables below, ‘–’ denotes not available (no implementation / not supported) and † denotes timeout (did not finish within the time limit). We allowed each run at least 10 hours, usually much longer, before reporting a timeout. We test both positive and negative cases: equivalent and nonequivalent pairs.

Table 1. Runtime comparison (in seconds) on the inverse-function *positive* instances for dimensions $n = 10$ to $n = 20$. The first block reports *linear* equivalence runtimes, and the second block reports *affine* equivalence runtimes. We generated random instances equivalent to the inverse function.

Algorithm	Dimension n											
	10	11	12	13	14	15	16	17	18	19	20	
<i>Linear equivalence</i>												
LINEAREQ-BACKFORTH	0.1	0.5	2.9	5.8	9.8	15	729	2220	7493	13850	†	
LINEAREQ-OURS	0.2	0.3	2.0	2.2	6.6	17	69	68	354	370	1641	
CCZEQ	2.4	11	65	175	1025	4333	–	–	–	–	–	
<i>Affine equivalence</i>												
AFFINEEQ-BACKFORTH	228	981	36769	88995	1900664	†	†	†	†	†	†	
AFFINEEQ-OURS	0.3	0.4	2.1	1.9	7.0	15	67	70	310	398	1935	
CCZEQ	2.1	11	81	178	1018	4188	–	–	–	–	–	

For negative tests, inequivalence can often be established by comparing invariants. However, these may themselves be expensive to compute and do not always distinguish inequivalent functions. We therefore select a variety of function

pairs that share several commonly used invariants, providing more challenging benchmarks for equivalence testing.

Table 2. Runtime comparison (median seconds) on specific *non-equivalent* pairs (F, G) . Each entry is the median time to conclude inequivalence (return **false**) over 10 runs, including the relevant self-equivalence-group precomputation.

n	Pair (F, G)	AFFINEEQ OURS	AFFINEEQ BACKFORTH	CCZ EQ
8	[23] $F = x^{173} + u^{182}x^{88} + u^3x^3, G = x^{173} + u^{39}x^{88} + u^{29}x^3$	0.008	1.58	0.065
9	[6] APN perms over $\mathbb{F}_8^3$: $u^3 + u + 1 = 0$ $F = (x^3 + uy^2z, y^3 + uxxz^2, z^3 + ux^2y)$ $G = (x^3 + (u+1)y^2z, y^3 + (u+1)xz^2, z^3 + (u+1)x^2y)$	4.2	18.6	167.5
9	Trace APN non-perms [11] $F = x^3 + \text{Tr}((u^2 + u)x), G = x^3 + \text{Tr}(x^9 + x)$	0.0	–	1794.5
10	[13] Gen. open-butterfly 4-uniform perms: $u^5 + u^2 + 1 = 0$ $\alpha = 1, \beta_F = u + 1, \beta_G = u^2 + u + 1$ (cube-root exp. 21)	0.1	113	1.9
11	APN non-perms over $\mathbb{F}_{2^{11}}$: $u^{11} + u^2 + 1 = 0$ $F = x^3 + \text{Tr}(u^3x), G = x^3 + \text{Tr}(x^9 + x)$ [11]	0.0	–	†
12	[28] Li–Wang ($i = 2$) versus Li–Wang ($i = 3$)	5.1	5241.7	20.8
13	Kasami vs Welch (APN perms, deg. 3) [14] $F(x) = x^{13}, G(x) = x^{67}$	43.9	32751.0	†
14	[13] Gen. open-butterfly 4-uniform perms over $\mathbb{F}_{2^7}$: $u^7 + u + 1 = 0$ $F = B_{u+1}, G = B_{u^3}$	1.5	†	922.7
15	BCL-versus-Gold (both APN perms) [10, 14] $F(x) = x^3 + u^{31}x^{1088}, G(x) = x^5$	1159.4	†	†
16	[24] Modified inverse (4-uniform perms, deg. 15) $\beta_F = u^5 + u^4 + u^2, \beta_G = u^8 + u^7 + u^4 + u + 1$ $\mathbb{F}_{2^{16}} = \mathbb{F}_2[u]/(u^{16} + u^5 + u^3 + u^2 + 1)$	17907.3	†	–

Table 3. Runtime comparison (average seconds) on positive instances for the explicitly specified base functions F . Each base function is compared with ten independently generated affine equivalent functions

n	Function	AFFINEEq OURS	AFFINEEq BACKFORTH	CCZ Eq
8	Pott–Zhou APN non-permutation [34] ($k = 1, s = 0$) $F(x, y) = (x^3 + uy^3, xy), (x, y) \in \mathbb{F}_{2^4}^2$ $u^4 + u + 1 = 0$	0.0	–	0.3
8	4-uniform permutation trinomial [23] $F(x) = x^{173} + u^{182}x^{88} + u^3x^3, x \in \mathbb{F}_{2^8}$ $u^8 + u^4 + u^3 + u^2 + 1 = 0$	0.0	1.3	0.1
9	Sporadic APN permutation [6] $F(x, y, z) = (x^3 + uy^2z, y^3 + uxz^2, z^3 + ux^2y)$ $(x, y, z) \in \mathbb{F}_8^3, u^3 + u + 1 = 0$	4.6	307	156
9	Gold plus linear trace (APN non-permutation) [14, 21] $F(x) = x^3 + \text{Tr}((u^2 + u)x), x \in \mathbb{F}_{2^9}$ $u^9 + u^4 + 1 = 0$	0.0	–	6.8
10	4-uniform open butterfly [13] $F(x, y) = ((y + t)^3 + (u + 1)t^3, t)$ $t = (x + (u + 1)y^3)^{21} + y, (x, y) \in \mathbb{F}_{2^5}^2$ $u^5 + u^2 + 1 = 0$	0.1	53	2.3
11	Gold plus linear trace (APN non-permutation) [14, 21] $F(x) = x^3 + \text{Tr}(u^3x), x \in \mathbb{F}_{2^{11}}$ $u^{11} + u^2 + 1 = 0$	0.0	–	739
12	Li–Wang 4-uniform permutation ($i = 2$) [28] $F(x) = x^{6553} + x^{1639} + x, F : H \rightarrow H$ $H = \{x \in \mathbb{F}_{2^{13}} : \text{Tr}(x) = 0\}$	5.3	3827	43
12	Pott–Zhou APN non-permutation [34] ($k = 1, s = 0$) $F(x, y) = (x^3 + uy^3, xy), (x, y) \in \mathbb{F}_{2^6}^2$ $u^6 + u^4 + u^3 + u + 1 = 0$	0.6	–	222
13	Kasami APN permutation [14, 26] $F(x) = x^{13}, x \in \mathbb{F}_{2^{13}}$	49	†	†
13	Gold plus linear trace (APN non-permutation) [14, 21] $F(x) = x^3 + \text{Tr}((u^3 + u^2 + u)x), x \in \mathbb{F}_{2^{13}}$ $u^{13} + u^4 + u^3 + u + 1 = 0$	0.1	–	†
14	4-uniform open butterfly [13] $F(x, y) = ((y + t)^3 + (u + 1)t^3, t)$ $t = (x + (u + 1)y^3)^{85} + y, (x, y) \in \mathbb{F}_{2^7}^2$ $u^7 + u + 1 = 0$	1.5	†	1615

For a concrete negative benchmark, we compare the inequivalent functions $F(x) = x^3$ and $G(x) = x^5$ over $\mathbb{F}_{2^n}$. For odd n , both are quadratic APN permutations [14], so the ortho-derivative-based invariants of [12] provide a specialized approach to testing their inequivalence. For even n , the functions have different differential uniformities (2 and 4, respectively) and hence are inequivalent. This pair provides a benchmark for comparing the runtime of general-purpose equivalence algorithms.

Table 4. Runtime comparison (average seconds) on the negative experiment x^3 versus x^5 in dimension n .

Algorithm	Dimension n											
	9	10	11	12	13	14	15	16	17	18	19	20
<i>Linear equivalence</i>												
LINEAREQ-OURS	0.2	0.4	0.7	3.6	4.1	12	31	94	37	484	763	3207
LINEAREQ-BACKFORTH	110	–	6821	–	401601	–	†	–	†	–	†	–
CCZEQ	6.7	61	461	119	38214	†	†	–	–	–	–	–
<i>Affine equivalence</i>												
AFFINEEQ-OURS	0.2	0.5	1.7	3.7	17	30	418	89	2354	2596	40934	3176
AFFINEEQ-BACKFORTH	86	–	853	–	98936	–	†	–	†	–	†	–
CCZEQ	6.7	61	461	119	38214	†	†	–	–	–	–	–

On some of these instances, SboxU tests affine equivalence faster than linear equivalence. The two routines use different strategies: the affine routine computes canonical representatives under linear equivalence for translated versions of the functions and searches for a match. On these instances, this approach is faster than the direct linear-equivalence search, despite requiring many canonicalization calls. See the algorithm in [8] and its SboxU implementation [30].

9 Correctness

Lemma 11 (Correctness of DFS for self-equivalence). *Assume $F = G$ and let $H \leq \text{Aut}(F)$. Assume that the current paired partitions satisfy the four alignment invariants and that the local stabilizers are*

$$H^{(1)} = \pi_1(H_\Delta), \quad H^{(2)} = \pi_2(H_\Delta).$$

Then DFS returns an element of

$$\mathcal{C}(\Delta) \setminus H$$

if such an element exists, and returns false otherwise.

Proof. The proof is by induction on the remaining depth of the search tree.

First consider the refinement step. Every signature pair used by the refinement algorithm is valid relative to the current branch decisions Δ . Hence refinement preserves the four alignment invariants. If, after refinement, some pair of corresponding ordered partitions does not have the same shape, then no affine permutation can map one partition to the other while respecting the ordering. Therefore

$$\mathcal{C}(\Delta) = \emptyset,$$

and returning false is correct.

Now suppose that the current branch decisions determine both affine maps A_1 and A_2 . The algorithm explicitly checks whether $A_2 \circ F = F \circ A_1$. It also checks whether $(A_1, A_2) \in H$. Thus, in the terminal case, the algorithm returns exactly when the determined pair lies in $\mathcal{C}(\Delta) \setminus H$, and otherwise correctly returns false.

It remains to justify the branching step. We prove the case where the algorithm branches on a domain point; the arguments for codomain points, domain hyperplanes, and codomain hyperplanes are identical, using the corresponding orbit-transport lemmas.

Suppose that the algorithm branches on a point $x \in P_i^F$, and suppose that

$$(A_1, A_2) \in \mathcal{C}(\Delta) \setminus H.$$

By the input point-cell invariant, $A_1(x) \in P_i^G$. Let r be the chosen representative of the $H^{(1)}$ -orbit of $A_1(x)$ in P_i^G . Then there exists $h_1 \in H^{(1)}$ such that

$$h_1(A_1(x)) = r.$$

Since $H^{(1)} \leq \pi_1(H_\Delta)$, there exists $\varphi = (\varphi_1, \varphi_2) \in H_\Delta$ such that $\varphi_1 = h_1$. Hence

$$\varphi_1(A_1(x)) = r.$$

By Lemma 8,

$$(\varphi_1 \circ A_1, \varphi_2 \circ A_2) \in \mathcal{C}(\Delta \cup \{A_1(x) = r\}).$$

We now show that this transported completion is still outside H . Suppose, for contradiction, that

$$(\varphi_1 \circ A_1, \varphi_2 \circ A_2) \in H.$$

Since $\varphi \in H_\Delta \leq H$, we have $\varphi^{-1} \in H$. Because H is a subgroup, it is closed under composition. Therefore

$$(A_1, A_2) = \varphi^{-1} \cdot (\varphi_1 \circ A_1, \varphi_2 \circ A_2) \in H,$$

contradicting $(A_1, A_2) \notin H$. Thus the representative branch $\Delta \cup \{A_1(x) = r\}$ contains an element of $\mathcal{C}(\Delta \cup \{A_1(x) = r\}) \setminus H$.

By the induction hypothesis, the recursive call on that branch returns an element of

$$\mathcal{C}(\Delta \cup \{A_1(x) = r\}) \setminus H.$$

Therefore, whenever $\mathcal{C}(\Delta) \setminus H$ is nonempty, at least one recursive branch returns such an element. Conversely, every element returned by a recursive call is checked at the terminal step and belongs to $\mathcal{C}(\Delta) \setminus H$. Hence, if all recursive calls return false, then

$$\mathcal{C}(\Delta) \setminus H = \emptyset.$$

This proves the claim.

The correctness of the complete self-equivalence algorithm then follows simply by applying Lemma 11 to the initial search state. The proof for the equivalence algorithm follows the same argument, with the subgroup-membership rejection omitted.

Acknowledgments. We used Codex to generate code and experiments, particularly for Section 8.

Disclosure of Interests. The author has no competing interests.

References

1. Baek, C.H., Cheon, J.H., Hong, H.: White-box AES implementation revisited. *Journal of Communications and Networks* **18**(3), 273–287 (2016). <https://doi.org/10.1109/JCN.2016.000043>
2. Barkan, E., Biham, E.: In how many ways can you write Rijndael? In: *Advances in Cryptology – ASIACRYPT 2002. Lecture Notes in Computer Science*, vol. 2501, pp. 160–175. Springer (2002). https://doi.org/10.1007/3-540-36178-2_10
3. Baudrin, J., Beierle, C., Felke, P., Leander, G., Neumann, P., Perrin, L., Stennes, L.: Commutative cryptanalysis as a generalization of differential cryptanalysis. *Designs, Codes and Cryptography* **93**(8), 3243–3281 (2025). <https://doi.org/10.1007/s10623-025-01625-9>
4. Baudrin, J., Felke, P., Leander, G., Neumann, P., Perrin, L., Stennes, L.: Commutative cryptanalysis made practical. *IACR Transactions on Symmetric Cryptology* **2023**(4), 299–329 (2023). <https://doi.org/10.46586/tosc.v2023.i4.299-329>
5. Beierle, C., Brinkmann, M., Leander, G.: Linearly self-equivalent APN permutations in small dimension. *IEEE Transactions on Information Theory* **67**(7), 4863–4875 (2021). <https://doi.org/10.1109/TIT.2021.3071533>
6. Beierle, C., Carlet, C., Leander, G., Perrin, L.: A further study of quadratic APN permutations in dimension nine. *Finite Fields and Their Applications* **81**, 102049 (2022). <https://doi.org/10.1016/j.ffa.2022.102049>
7. Beierle, C., Leander, G.: New instances of quadratic APN functions. *IEEE Transactions on Information Theory* **68**(1), 670–678 (2022). <https://doi.org/10.1109/TIT.2021.3120698>
8. Biryukov, A., De Cannière, C., Braeken, A., Preneel, B.: A toolbox for cryptanalysis: Linear and affine equivalence algorithms. In: *Advances in Cryptology – EUROCRYPT 2003. Lecture Notes in Computer Science*, vol. 2656, pp. 33–50. Springer (2003). https://doi.org/10.1007/3-540-39200-9_3
9. Bonnetain, X., Perrin, L., Tian, S.: Anomalies and vector space search: Tools for S-box analysis. In: *Advances in Cryptology – ASIACRYPT 2019. Lecture Notes in Computer Science*, vol. 11921, pp. 196–223. Springer (2019). https://doi.org/10.1007/978-3-030-34578-5_8
10. Budaghyan, L., Carlet, C., Leander, G.: Two classes of quadratic APN binomials inequivalent to power functions. *IEEE Transactions on Information Theory* **54**(9), 4218–4229 (2008). <https://doi.org/10.1109/TIT.2008.928275>
11. Budaghyan, L., Carlet, C., Leander, G.: Constructing new APN functions from known ones. *Finite Fields and Their Applications* **15**(2), 150–159 (2009). <https://doi.org/10.1016/j.ffa.2008.10.001>
12. Canteaut, A., Couvreur, A., Perrin, L.: Recovering or testing extended-affine equivalence. *IEEE Transactions on Information Theory* **68**(9), 6187–6206 (2022). <https://doi.org/10.1109/TIT.2022.3166692>
13. Canteaut, A., Duval, S., Perrin, L.: A generalisation of Dillon’s APN permutation with the best known differential and nonlinear properties for all fields of size 2^{4k+2} . *IEEE Transactions on Information Theory* **63**(11), 7575–7591 (2017). <https://doi.org/10.1109/TIT.2017.2676807>

14. Carlet, C.: Boolean Functions for Cryptography and Coding Theory. Cambridge University Press (2021). <https://doi.org/10.1017/9781108606806>
15. Carlet, C., Charpin, P., Zinoviev, V.: Codes, bent functions and permutations suitable for DES-like cryptosystems. *Designs, Codes and Cryptography* **15**(2), 125–156 (1998). <https://doi.org/10.1023/A:1008344232130>
16. Cheon, J.H., Hong, H., Lee, J., Lee, J.: An efficient affine equivalence algorithm for multiple S-boxes and a structured affine layer. In: *Selected Areas in Cryptography – SAC 2016. Lecture Notes in Computer Science*, vol. 10532, pp. 299–316. Springer (2017). https://doi.org/10.1007/978-3-319-69453-5_17
17. De Meyer, L., Bilgin, B.: Classification of balanced quadratic functions. *IACR Transactions on Symmetric Cryptology* **2019**(2), 169–192 (2019). <https://doi.org/10.46586/tosc.v2019.i2.169-192>
18. De Mulder, Y., Roelse, P., Preneel, B.: Cryptanalysis of the Xiao–Lai white-box AES implementation. In: *Selected Areas in Cryptography – SAC 2012. Lecture Notes in Computer Science*, vol. 7707, pp. 34–49. Springer (2013). https://doi.org/10.1007/978-3-642-35999-6_3
19. Diaconis, P., Rockmore, D.: Efficient computation of the Fourier transform on finite groups. *Journal of the American Mathematical Society* **3**(2), 297–332 (1990). <https://doi.org/10.1090/S0894-0347-1990-1030655-4>
20. Dinur, I.: An improved affine equivalence algorithm for random permutations. In: *Advances in Cryptology – EUROCRYPT 2018*. pp. 413–442. *Lecture Notes in Computer Science*, Springer (2018). https://doi.org/10.1007/978-3-319-78381-9_16
21. Gold, R.: Maximal recursive sequences with 3-valued recursive cross-correlation functions. *IEEE Transactions on Information Theory* **14**(1), 154–156 (1968). <https://doi.org/10.1109/TIT.1968.1054106>
22. Ivkovic, I., Kaleyski, N.: Deciding and reconstructing linear equivalence of uniformly distributed functions. *Cryptology ePrint Archive*, Paper 2022/666 (2022), <https://eprint.iacr.org/2022/666>
23. Jeong, J., Kim, C.H., Koo, N., Kwon, S., Lee, S.: On cryptographic parameters of permutation polynomials of the form $x^r h(x^{(q-1)/d})$. *Cryptology ePrint Archive*, Paper 2019/767 (2019), <https://eprint.iacr.org/2019/767>
24. Jeong, J., Koo, N., Kwon, S.: New differentially 4-uniform permutations from modifications of the inverse function. *Finite Fields and Their Applications* **77**, 101931 (2022). <https://doi.org/10.1016/j.ffa.2021.101931>
25. Kaleyski, N., Sunde, J.: Efficiently deciding and recovering CCZ and EA equivalence for arbitrary vectorial Boolean functions using the partition refinement framework. *Cryptology ePrint Archive*, Paper 2026/940 (2026), <https://eprint.iacr.org/2026/940>
26. Kasami, T.: The weight enumerators for several classes of subcodes of the 2nd order binary Reed–Muller codes. *Information and Control* **18**(4), 369–394 (1971). [https://doi.org/10.1016/S0019-9958\(71\)90473-6](https://doi.org/10.1016/S0019-9958(71)90473-6)
27. Leon, J.S.: Permutation group algorithms based on partitions, I: Theory and algorithms. *Journal of Symbolic Computation* **12**(4-5), 533–583 (1991). [https://doi.org/10.1016/S0747-7171\(08\)80103-4](https://doi.org/10.1016/S0747-7171(08)80103-4)
28. Li, Y., Wang, M.: Constructing differentially 4-uniform permutations over $\mathbb{F}_{2^{2m}}$ from quadratic APN permutations over $\mathbb{F}_{2^{2m+1}}$. *Designs, Codes and Cryptography* **72**(2), 249–264 (2014). <https://doi.org/10.1007/s10623-012-9760-9>
29. McKay, B.D., Piperno, A.: Practical graph isomorphism, II. *Journal of Symbolic Computation* **60**, 94–112 (2014). <https://doi.org/10.1016/j.jsc.2013.09.003>

30. Perrin, L., Alich, J., Baudrin, J., Boeuf, A., Bonnetain, X., Couvreur, A., Fruchon, M., Joly, M., Galissant, P., Stennes, L.: sboxU: black-box analysis of discrete functions. Available online at <https://github.com/lpp-crypto/sboxU/> (2026)
31. Raddum, H.: More dual Rijndaels. In: Advanced Encryption Standard – AES 2004. Lecture Notes in Computer Science, vol. 3373, pp. 142–147. Springer (2005). https://doi.org/10.1007/11506447_12
32. Seress, Á.: Permutation Group Algorithms. Cambridge University Press (2003). <https://doi.org/10.1017/CB09780511546549>
33. Varici, K., Nikova, S., Nikov, V., Rijmen, V.: Constructions of S-boxes with uniform sharing. *Cryptography and Communications* **11**(3), 385–398 (2019). <https://doi.org/10.1007/s12095-018-0345-y>
34. Zhou, Y., Pott, A.: A new family of semifields with 2 parameters. *Advances in Mathematics* **234**, 43–60 (2013). <https://doi.org/10.1016/j.aim.2012.10.014>