Generic Framework for Key-Guessing Improvements

Marek Broll[1], Federico Canale[1], Antonio Flórez-Gutiérrez[2], Gregor Leander[1], and María Naya-Plasencia[2]

[1] Horst Görtz Institute for IT Security, Ruhr University Bochum, Bochum, Germany, {marek.broll,federico.canale,gregor.leander}@rub.de
[2] Inria, France, {antonio.florez-gutierrez,maria.naya_plasencia}@inria.fr

Abstract. We propose a general technique to improve the key-guessing step of several attacks on block ciphers. This is achieved by defining and studying some new properties of the associated S-boxes and by representing them as a special type of decision trees that are crucial for finding fine-grained guessing strategies for various attack vectors. We have proposed and implemented the algorithm that efficiently finds such trees, and use it for providing several applications of this approach, which include the best known attacks on Noekeon, GIFT, and RECTANGLE.

Keywords: cryptanalysis, S-box, key-guessing, affine decision trees

1 Introduction

Literally *all* sensitive data needs to be encrypted, and it is vital to have trustworthy symmetric primitives. The only way to build confidence in these primitives is through a continuous effort to evaluate their security and constantly update their security margin: this is the role of cryptanalysis.

Several different attack families against symmetric ciphers exist. The most important are differential and linear cryptanalysis ([3], [16, 17]) and their variants. While the boundary is often blurry (see e.g. [11]), many attacks can usually be separated into two parts: a distinguisher and a key-recovery part.

A distinguisher highlights some non-random behaviour in a part of a cipher, like linear correlation between several states or an output difference occurring unusually often when a specific input difference is introduced.

The key-recovery part usually involves the rounds before and after the distinguisher, and makes use of this non-random behaviour to (partially) recover the secret key. Fundamentally, the attacker guesses some key information from this outer part, and checks if the non-random behaviour occurs with the distinguisher. If the data behaves as expected, the key guess is likely correct. Our work focuses on the key-recovery step.

Various commonly-used ideas to improve the efficiency of this part have been proposed, such as reducing the data complexity by using plaintext structures (see e.g. [15] for applications to ARX), improved statistical tools (e.g. [4]), and the

use of the Fast Fourier Transform (FFT) in linear cryptanalysis ([9] and the improved [13]).

For SPN ciphers, the key-guessing is often done in a word-oriented fashion in which key-words are guessed in alignment with the S-box layer. The S-box is treated like a black box, and a full key-word is guessed when the attacker needs some information about its output. There are examples of partial improvements to the key-guessing in some specific attacks, albeit never in a generic manner. Some decompose the S-box to either filter wrong pairs (e.g. [12]), avoid unnecessary key guesses ([6]), or improve filtering in meet-in-the-middle attacks ([8]).

A comprehensive and focused study of S-box properties with respect to optimal key-guessing strategies is nevertheless missing.

Our Contribution

In this paper we provide this overdue analysis by introducing a unified and generic framework to optimize the key-recovery part of various attacks. Inspired particularly by the techniques used in [6], we aim to reduce the number of key bits guessing to the strict minimum for which the output information is still determined, avoiding unnecessary guesses of full-key words. To this end, we first transform an S-box (or one of its component functions) into a binary decision tree. We then show that all the important optimizations naturally arise as properties of this tree. We find that one of the most important properties is the number of leaves. Consequently, finding tree representations with a minimal number of leaves directly optimizes the attacks.

While their application to cryptanalysis is new, (parity) decision trees for boolean functions themselves are not. For an overview of the theory see [18]. The (asymptotic) size (which we call numLeaves) and approximation of parity decision trees (i.e. decision trees with arbitrary instead of unit vector labels) is subject to research (e.g. [20]). Here, a link to linear structures in the case of vectorial boolean functions is examined.

We first describe this tree representation and discuss some basic properties in Section 2. In particular, we show that optimizing the number of leaves automatically considers linear structures, a simple and well-known property. Moreover, we show that equivalence conditions for functions to lead to isomorphic trees, which allows us to classify functions with respect to their optimal trees. In addition, this provides new criteria for choosing good S-boxes with better resistance against attacks which exploit our representation explicitly or implicitly. We also provide a simple yet efficient algorithm which computes an optimal tree for reasonable S-box sizes ($n < 8$), which has been necessary for the applications. An implementation is provided as supplementary material online [3].

Before giving several specific application examples, we explain how using trees can improve various generic attack families in a broader sense in Section 3.

Concrete applications are detailed in the following sections. In Section 4 we explain how to optimize linear attacks by giving the current best attack on

[3] https://github.com/rub-hgi/ConditionsLib

NOEKEON [10]. We then focus on differential attacks. We improve the best attack on GIFT that was known at the time of writing ([14], see Section 5), a related-key rectangle attack, and decrease its time complexity by a factor of more than 2^{20} and its data complexity by a factor of 2. However, we are confident that an improvement to the new best attack on GIFT [21] is also possible thanks to our techniques. We also attack the cipher RECTANGLE (see Section 6) and improve the time complexity of the best attack by a factor larger than 2^{14}. Finally, we explain how meet-in-the-middle or more precisely sieve-in-the-middle attacks can also benefit from our improvements on the example of PRESENT [5] in the extended version of the paper [7]. Our attack provides just a small improvement factor, but shows how our technique can be applied. Our on PRESENT concrete findings are summarized in Table 1.

We expect that follow-up work will use our results for building even better attacks, including attacks on more rounds. Our main aim was to provide applications that underline the usefulness of the framework. For covering more rounds, one should design a whole new attack using our ideas. In particular, we expect that a 19 round attack on RECTANGLE is within reach, due to the fact that the already large margin for the key-guessing complexity can be further improved if one aims at optimization (and not simplicity, as we do in the present work). Furthermore, note that there is nothing fundamental that prevents the framework from being applied to larger S-boxes.

Cipher (Block,Key)	Rnds	Type	(Time, Data)-Previous	This paper	Best
NOEKEON-128-128	12	linear	$(2^{124}, 2^{124})$ [10]	$(2^{122.14}, 2^{119})$	yes
GIFT-64-128	25	RK rectangle	$(2^{120.92}, 2^{63.78})$ [14]	$(2^{99.18}, 2^{62.73})$	no
RECTANGLE -64-80	18	differential	$(2^{78.88}, 2^{64})$[22]	$(2^{64}, 2^{64})$	yes
PRESENT-64-80	8	sieve-in-the-middle	$(2^{73.42}, 2^{6})$ [8]	$(2^{72.91}, 2^{6})$	no

Table 1. Overview of the Applications. The improvements on NOEKEON and RECTANGLE provide the new best known attacks on these ciphers. [14] was the best attack on GIFT at the time of writing. An attack on 26-round GIFT was presented in [21].

2 Representing Functions as Affine Decision Trees and Applications in Cryptanalysis

In this section, we develop our new, condition-centered representation of S-boxes that is motivated by trying to compute (parts of) the output given only partial information on the inputs.

We denote by $\mathbb{F}_2$ the field with two elements, i.e. a bit, and by $\mathbb{F}_2^n$ the n-dimensional vector space over it. For x, y in $\mathbb{F}_2^n$ we denote the canonical inner product $\sum_i x_i y_i$ by $\langle x, y \rangle$.

An S-box, or more generally, a part of a cipher, is a function

$$S : \mathbb{F}_2^n \to \mathbb{F}_2^m.$$

Such functions are either represented by a simple look-up table or its algebraic normal form.

However, for our purpose of improving the key-recovery part of several attacks, the representation as a look-up table or as a polynomial is not very suitable, as they hide possible short-cuts and finding an optimal solution with them often requires exhaustively trying all the possible restrictions. The basic property we are going to use in all the attacks is that we can deduce information about (parts of) the output even when only partial information on the input is available.

As a first example we consider the NOEKEON S-box S

x	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
$S(x)$	7	a	2	c	4	8	f	0	5	9	1	e	3	d	b	6

In particular, consider the function $f(x)$ that outputs the most significant bit of $S(x)$, which is given as the following look-up table.

x_0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
x_1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
x_2	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
x_3	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
$f(x)$	0	1	0	1	0	1	1	0	0	1	0	1	0	1	1	0

A closer look at the table above reveals that the output of f actually does not depend on x_3 at all. This corresponds to the well-known property of a linear structure of a Boolean function and in this example is given by the fact that

$$f(x) + f(x + (0, 0, 0, 1)) = 0 \quad \forall x.$$

This is a first, trivial but very helpful example of the property we are looking for. However, more can be said. To cite another example, in the case of $x_1 = 0$ and $x_0 = 0$ we get $f(x) = 0$ independent of the value of x_2. If $x_1 = 0$ and $x_0 = 1$ we get $f(x) = 1$. In case $x_1 = 1$ knowing a single bit in addition will not be sufficient, however one additional bit of information actually is. Namely if $x_1 = 1$ and $x_0 + x_2 = 0$ we get $f(x) = 0$. Finally, if $x_1 = 1$ and $x_0 + x_2 = 1$, we get $f(x) = 1$.

Now, instead of collecting these conditions in terms of equations, a better way is to present them in terms of graphs which we will define formally below. The example given here translates into the graph shown in Figure 1. Starting with the root, each node is labeled with a vector corresponding to a linear combination of the inputs. Depending on the value of this linear combination of the inputs, the left or right edge is taken until one ends up in a leaf. Leaves are labeled with the value the function takes on all the inputs fulfilling the conditions corresponding to the path leading there.

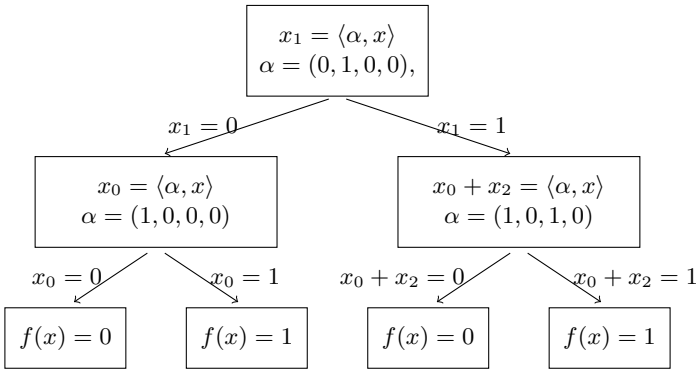


Fig. 1. Graph representation of the conditions for f

Thus, the graph is a representation of the function f that actually captures exactly the conditions we wanted. It can also be thought of as a way of implementing the function. Clearly given a function, the representation of the graph is not unique. Indeed, the graph in Figure 2 is a graph for the same function, but intuitively (and also formally as we will see later) less helpful. Indeed, except x_3, which we know is not relevant, in order to compute the output, every input bit has to be known in this representation.

Considering $x + k$ as input to f, in order to evaluate the function for some fixed x, we have to obtain (usually by guessing all the possible values) enough bits of k to calculate $\langle \alpha, x + k \rangle$ for inner nodes on the path which is taken during the evaluation of $f(x + k)$.

In the end, for each fixed x, we find that we must consider a different guess of k for each possible evaluation path through the graph, or that is, one guess for each leaf of the tree. Since the number of leaves is at most equal to 2^n (which is the "naïve" number of guesses), we can often reduce the time complexity of the attack, as we will explain below when optimizing the tree towards a minimal number of leaves.

In the remaining part of this section, we explain how to find good graphs automatically, how those related to the linear structures, and how equivalent functions lead to equivalent graphs.

All the guessing strategies we consider can be thought of as guessing one bit at a time and depending on the result of the guess continue on the left branch (in case we guess zero) or in the right branch (in case we guess one).

We assume that, along one path from the source to a leaf, all node labels (the α-values) are independent. Then, at each stage, the linear combination of inputs splits the space into equal parts, a subspace and its complement. Both

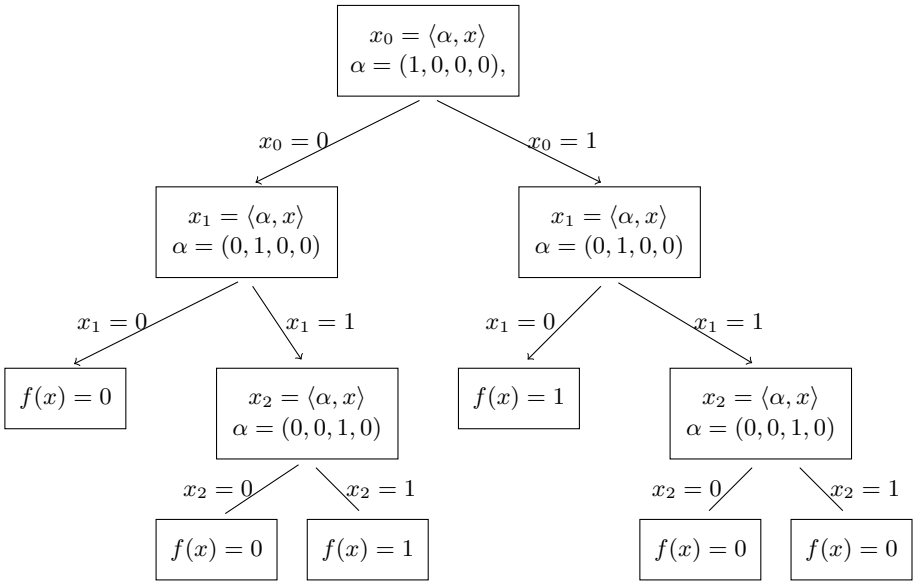


Fig. 2. Alternative Graph representation of the conditions for f

can be identified with a space again, and this is the space in the next level. The advantage is that this view is very simple, general and recursive.

2.1 Formalization

Instead of starting with a function and building a tree in the above manner, it is more convenient to directly start with a tree and discuss the function it corresponds to afterwards. The trees we consider are binary trees, where each node either has two or no children. More formally, we consider trees defined as follows.

Definition 1 (Affine Decision Tree). *An* (n, m)-affine decision tree *is a regular and binary tree where each node* v *has a label* $v.label$*. A node without children is called a* leaf *and has labels in* $\mathbb{F}_2^m$*. A node* v *which is not a leaf is called an* inner node *and has labels in* $\mathbb{F}_2^n$*. Its children are denoted by* $v.left$ *and* $v.right$*.*

We identify a tree and its root whenever it simplifies the formulation. For a tree r*, we also write* $v \in r$ *when* v *is a node of* r*.*

In Figure 1 the labels for the inner nodes are denoted with α and the nodes for the leafs correspond to the value the function takes on the corresponding inputs.

These trees correspond to maps (generalizing the example above) as follows. We will take the liberty to use the same notation for both the tree and the corresponding map.

Definition 2. *Given an (n, m)-affine decision tree r we can construct an associated map $r : \mathbb{F}_2^n \to \mathbb{F}_2^m$. For $x \in \mathbb{F}_2^n$ we define $r(x)$, the value calculated by an affine decision tree r, recursively:*

1. *If r is a leaf, $r(x) = r.label$.*
2. *If r is an inner node and $\langle r.label, x\rangle = 0$, $r(x) = r.left(x)$.*
3. *If r is an inner node and $\langle r.label, x\rangle = 1$, $r(x) = r.right(x)$.*

Given $f\colon \mathbb{F}_2^n \to \mathbb{F}_2^m$, if $r(x) = f(x)$ for all x, we say that r is a tree for f.

It is clear that for any given function, there can be many possible trees, again see Figures 1 and 2. However, for the applications considered later, we are interested in trees which lead to a small overhead on attack complexity. In our applications this is mostly achieved for trees with a low number of leaves, which we denote by

$$\text{numLeaves}(r) = \text{ Number of leaves of } r$$

for a tree r.

To give an example, Figure 1 corresponds to a tree r_1 with $\text{numLeaves}(r_1) = 4$, while Figure 2, implementing the same function, corresponds to a tree r_2 with $\text{numLeaves}(r_2) = 6$.

Especially for linear cryptanalysis, besides the number of leaves, the union of all inner labels $r.label$ that have to be evaluated as $\langle r.label, x\rangle$ when evaluating the function r on all possible inputs is of interest. This is what we call the *actual linear domain* and is formally defined in the next definition.

Definition 3 (domsize**, Actual Linear Domain).** *The* actual linear domain *of r is the space spanned by all inner node labels:*

$$\text{Dom}(r) = \text{span}\{n.label.\ n \text{ is an inner node of } r\}.$$

We call its dimension $\text{domsize}(r)$.

In the graph r_1 from Figure 1 the actual linear domain is given by

$$\text{Dom}(r_1) = \text{span}\left\{(1,0,0,0), (0,1,0,0), (1,0,1,0)\right\},$$

which corresponds to all vectors x of the form $x = (*, *, *, 0)$. The actual linear domain of the graph r_2 in Figure 2 is exactly the same, even though the concrete labels are different. In both cases we get

$$\text{domsize}(r_1) = \text{domsize}(r_2) = 3.$$

For a fixed function, we are interested in the optimal tree with respect to the number of leaves and with respect to the actual linear domain. For a given function $f\colon \mathbb{F}_2^n \to \mathbb{F}_2^m$ we denote the minimal number of leaves of all trees for f by $\text{minLeaves}(f)$. That is

$$\text{minLeaves}(f) = \min_{r:\forall x.r(x)=f(x)} \text{numLeaves}(r).$$

A tree r for f taking on this minimum is called numLeaves-*minimal*.

Similarly, we call the minimal actual linear domain of all trees for f the *optimal actual linear domain size* of f, denoted by $\mathrm{domopt}(f)$. More formally,

$$\mathrm{domopt}(f) = \min_{r:\forall x.r(x)=f(x)} \mathrm{domsize}(r).$$

Again, a tree r for f taking on this minimum is called domsize-*minimal*.

We are interested in finding trees which optimise either parameter depending on the application. Luckily, any tree that is optimal with respect to the number of leaves is also optimal with respect to the actual linear domain, as we show now.

Connection between Linear Structures and Dom(r) In order to see this, it is helpful to have a closer look at all the values $x \in \mathbb{F}_2^n$ that end up at the same leaf N in the tree in the evaluation of $r(x)$. As the evaluation of r for a given input x consists of computing a sequence of inner products $\langle r.label, x\rangle$ along the path from the root to a leaf, the exact values x that end up in the same leaf are characterised by the values of those inner products.

For a node N of a tree r, let us denote by $D(N)$ the set of all labels on the path from the root to that node, excluding N itself.

$$D(N) \coloneqq \mathrm{span}\{v.label \colon v \textit{ is on the path leading from } r \textit{ to } N, v \neq N\}$$

Furthermore, again for a given node N we denote by $N.space$ the set of all inputs $x \in \mathbb{F}_2^n$ such that the evaluation path of $r(x)$ contains N.

If N is a node, then $D(N)$ corresponds to the set of inner products which have been evaluated to reach it, while $N.space$ corresponds to the set of inputs which end up in that leaf during the evaluation of $r(x)$. So $N.space$ is an affine subspace of the form

$$N.space = V(N) + a(N),$$

where $V(N) \subseteq \mathbb{F}_2^n$ is a vector subspace and $a(N) \in \mathbb{F}_2^n$ is a translation. In fact, $V(N)$ consists of all vectors v such that $\langle r.label, v\rangle = 0$, for $r.label \in D(N)$. Thus, $V(N)$ is the dual space of the span of $D(N)$:

$$V(N) = D(N)^{\perp}.$$

For example, the left-most leaf N of the tree in Figure 1 is reached for all elements of $N.space = D(N)^{\perp} + a(N) = \mathrm{span}\{(0,1,0,0),(1,0,0,0)\}^{\perp} + 0 = \mathrm{span}\{(0,0,1,0),(0,0,0,1)\}$. The underlying subspace $V(N)$ of the two leaves of the right subtree is $\mathrm{span}\{(1,0,1,0),(0,0,0,1)\}$.

From these considerations we arrive at another interpretation of the actual linear domain:

Lemma 1.

$$\mathrm{Dom}(r)^{\perp} = \bigcap_{N \in r} V(N) = \bigcap_{\substack{N \in r \\ N \textit{ is a leaf}}} V(N)$$

Proof. We rewrite $\mathrm{Dom}(r)$ in term of the grouped labels $D(N)$:

$$\mathrm{Dom}(r) = \mathrm{span}\left(\bigcup_{N\in r} D(N)\right) = \sum_{N\in r} \mathrm{span}(D(N)),$$

and considering the dual spaces we get

$$\mathrm{Dom}(r)^\perp = \bigcap_{N\in r} \mathrm{span}(D(N))^\perp = \bigcap_{N\in r} D(N)^\perp = \bigcap_{N\in r} V(N).$$

Since $D(N_2) \subseteq D(N_1)$ whenever N_2 is a descendant of N_1, we deduce that $V(L) \subseteq V(N)$ if L is a leaf and N is one of its ancestors. This means that we can restrict the intersection to just the leaves of the tree. □

The domain of the tree in Figure 1 has already been given, its orthogonal complement is $\mathrm{span}\{(0,0,0,1)\}$. The intersection of the two $V(N)$ occurring in this tree is the same, $\mathrm{span}\{(0,0,1,0),(0,0,0,1)\} \cap \mathrm{span}\{(1,0,1,0),(0,0,0,1)\}$.

Since $\mathrm{Dom}(r)$ consists of all the inner products which may need to be evaluated, inner products v that are not contained in $\mathrm{Dom}(r)$ are never used when computing $r(x)$, and the value of $\langle v, x\rangle$ does not influence the image $r(x)$ for any x. This is a property which resembles the notion of linear structures. Linear structures, see e.g. [12][4], can be thought of as truncated differentials with probability one. More formally, they are defined as follows:

Definition 4 (Linear Structures). *The set of* 0*-linear structures of a function* $f\colon \mathbb{F}_2^n \to \mathbb{F}_2^m$ *is defined as*

$$\mathrm{LS}_0 = \{\alpha : \forall x \in \mathbb{F}_2^n,\ f(x) + f(x+\alpha) = 0\}.$$

It can be easily shown that LS_0 is in fact a vector subspace of $\mathbb{F}_2^n$.

To understand the connection with $\mathrm{Dom}(r)$, consider $\alpha \in \mathrm{Dom}(r)^\perp$ and two inputs $x, y \in \mathbb{F}_2^n$ which differ by α, that is, $x + y = \alpha$. Taking Lemma 1 into account, this implies that $x + y \in \bigcap_{N\in r} V(N)$ and thus x and y follow the same evaluation path and map to the same image, $r(x) = r(x+\alpha)$. Thus α is a 0-linear structure of r, and we conclude that

Lemma 2. *For any affine decision tree* r *we have*

$$\mathrm{Dom}(r)^\perp \subseteq \mathrm{LS}_0$$

We note that given a funtion f, the space LS_0 is independent of the tree r we choose, and it can be computed directly from f. This allows to efficiently bound the optimal actual linear domain size $\mathrm{domopt}(f)$ of any function simply by computing the dimension of its 0-linear structures.

[4] We use a slightly different definition of linear structures for vectorial Boolean functions which suits our purpose better than the original.

Theorem 1. *Let $f : \mathbb{F}_2^n \to \mathbb{F}_2^m$ be a given map and r be an $(n,m)-$affine decision tree for f which is optimal with respect to* numLeaves*. It holds that*

$$\mathrm{LS}_0(f) = \mathrm{Dom}(r)^\perp$$

and consequently

$$\mathrm{domopt}(f) = n - \dim(\mathrm{LS}_0(f)) = \mathrm{domsize}(r).$$

For our purposes, this has two important consequences: (i) any tree that is optimal with respect to the number of leaves is actually optimal with respect to the actual domain size, too, and (ii) computing the 0-linear structures of a target function first allows to compute the optimal tree on the function modulo its 0-linear structures, which provides a computationally less costly reduced input space.

The intuition for proving Theorem 1 is that for any tree evaluations of inner products $\langle \alpha, x \rangle$ can be removed when they correspond to 0-linear structures. We give a formal proof of Theorem 1 in the extended version of the paper [7].

Invariance under Transformations of the Input and Output The most important cryptographic criteria, e.g. the algebraic degree, the maximal probability for differential transitions, or the maximal absolute linear correlations, are invariant under affine equivalence. That is to say that, given a function $f : \mathbb{F}_2^n \to \mathbb{F}_2^m$ and two affine permutations $A : \mathbb{F}_2^n \to F_2^n$ and $B : \mathbb{F}_2^m \to \mathbb{F}_2^m$ the function $B \circ f \circ A$ has the same values for these criteria. This is of importance as it in particular (i) allows to classify S-boxes with respect to these criteria and (ii) gives larger freedom to the designer of a new primitive.

We next argue that the optimal number of leaves and the optimal actual linear domain size are invariant under an even larger notion of equivalence.

For this, let $f \colon \mathbb{F}_2^n \to \mathbb{F}_2^m$ be a function and let r be a tree for f. Consider an arbitrary, not necessarily affine, permutation $\pi : \mathbb{F}_2^m \to \mathbb{F}_2^m$. Replacing the labels of the leafs of r by their images under π, we automatically get a tree for $\pi \circ f$ directly. Moreover, the structure of the tree, and thus the number of leaves, is not affected by this modification. This implies that $\mathrm{numLeaves}(B \circ f) = \mathrm{numLeaves}(f)$ for any function f and any permutation B.

Next, consider an affine permutation $A : \mathbb{F}_2^n \to \mathbb{F}_2^n$. In order to change r, the tree for f, into a tree for $f \circ A$ two changes are necessary. First, the constant part of A will (potentially) swap the children of a node. Second, the linear part will be taken care of by changing the labels of all inner nodes of the tree (replacing a label α by $A^t\alpha$ in case A is linear). These observations, which are made more precise in the extended version of the paper [7], are summarized in the following.

Theorem 2. *Let r be a tree for a function $f \colon \mathbb{F}_2^n \to \mathbb{F}_2^m$. Let $A : \mathbb{F}_2^n \to \mathbb{F}_2^n$ be an affine permutation, and $\pi : \mathbb{F}_2^m \to \mathbb{F}_2^m$ be a permutation. It holds that*

$$\mathrm{minLeaves}(f) = \mathrm{minLeaves}(\pi \circ f \circ A)$$

and

$$\mathrm{domopt}(f) = \mathrm{domopt}(\pi \circ f \circ A)$$

We give a formal proof in the extended version of the paper [7].

Remark 1. Note that besides domopt and minLeaves any other criteria computed from the trees that is invariant under graph-isomorphism, behaves as described in Theorem 2. Examples that might be of interest include but are not limited to the number of bits used averaged over all inputs, maximal depth of the tree, and the number of leaves of a certain depth.

2.2 Computing Trees

In this part, we discuss the algorithmic aspects of computing (optimal) trees for a given function $f\colon \mathbb{F}_2^n \to \mathbb{F}_2^m$. Conceptually, it is easy to compute all possible trees recursively by choosing a root label $r.label = \alpha$ and then applying the algorithm recursively to $f|_{\langle \alpha, x\rangle = 0}$ and $f|_{\langle \alpha, x\rangle = 1}$ until these functions become constant. As we are mainly interested in optimal trees, and in order to (significantly) decrease the run time of the algorithm, several improvements are helpful. Those improvements basically avoid to search for, in a sense, "equivalent" trees and use early abort strategies when searching for a tree with a minimal number of leaves.

Algorithm 1 ListTrees(f, V, W)

Require: $f : \mathbb{F}_2^n \to \mathbb{F}_2^m$
Affine subpaces $V, W \subset \mathbb{F}_2^n$, $V = U + c \subseteq W = Z + c$, where U, Z are subspaces and c is a translation in $\mathbb{F}_2^n$
For the initial call we set $V = W = \mathbb{F}_2^n$
Ensure: A list of trees for $f|_V$.
Initialize an empty list L of all trees (generated) for $f|_V$.
if f is constant on V **then**
    Add leaf r with label $f(c)$ to L.
    return L.
end if
Calculate P such that $P \oplus U^\perp = Z$.
for all $\alpha \in P \setminus \{0\}$ **do**
    $U_0 \coloneqq \{x \in U\colon \langle \alpha, x\rangle = 0\}$.
    Choose $c' \in U$ such that $\langle c', \alpha\rangle = 1$. ▷ Exists due to the choice of P.
    $b = \langle c, \alpha\rangle$. ▷ Translating U_0 into V can change the value of $\langle \alpha, \cdot\rangle$.
    $V_b \coloneqq U_0 + c$, $V_{1-b} \coloneqq U_0 + c' + c$.
    Initialize a tree with root r and $r.label = \alpha$.
    for all $(r.left, r.right) \in$ ListTrees$(f, V_0, V) \times$ ListTrees(f, V_1, V) **do**
        Add a copy of r to L.
    end for
end for
return L

Improvements We already stated that, when building a subtree we can omit root labels which are linear combinations of the labels on the path leading to this subtree. That is, in all trees we consider, the labels along a path are linearly independent. Moreover, each label can be chosen up to the space spanned by the labels aready used, i.e. a label for a node N can basically be chosen in $\mathbb{F}_2^n/D(N)$. Algorithmically, this is done by running through a fixed complement space of $D(N)$. A pseudocode for the algorithm including this optimization is given in *Algorithm* 1.

A numLeaves-minimal tree can have at most 2^{n-d} leaves in a subtree of depth d as otherwise it would involve redundant bits of information on a path. This can be used to cut some recursive calls and reduce the run time of the algorithm.

For functions with linear structures in the sense of Definition 4 we can also ignore choices of sister nodes which only differ by a linear structure due to Theorem 1. This is equivalent to finding trees for the function $g\colon \mathbb{F}_2^n/\mathrm{LS}_0 \to \mathbb{F}_2^m$ with $g(x + \mathrm{LS}_0) = f(x)$. This can be done not only for the initial function but also for each sub-tree recursively.

Using these kinds of optimizations we could analyze individual functions up to dimension 7 in a reasonable amount of time. For our experiments we used a standard PC with a 2.3-GHz CPU. For dimension 4 it is usually possible to enumerate all trees using Algorithm 1 without optimizing the costs and filter afterwards. For k-bit Boolean functions chosen uniformly at random computing the optimal tree on a single core takes on average roughly 4 milliseconds for $k = 4$, 190 milliseconds for $k = 5$ and 21 seconds for $k = 6$. For $k = 7$ we could not test enough to get a reliable run-time estimate, but the program usually takes somewhere around 1.7 hours. For $k = 8$ we estimate an average running time of less than three weeks on the above machine.

Analyzing Balanced Boolean Functions in Dimension up to 5 When considering single components of S-boxes, only balanced Boolean functions are of interest. Using Theorem 2 together with Algorithm 1 allows us to classify all possible values for the optimal number of leaves at least for all balanced Boolean functions in small dimensions.

3 Application to generic attack families

The purpose of this section is to illustrate the time complexity improvements which can be obtained by applying the tree descriptions of boolean functions to some of the most widely-used attack families on SPN block ciphers.

The most natural case directly depends on minLeaves(S) which will become the cost of performing the guess, compared to 2^n. This natural case directly applies to linear attacks with no FFT acceleration and to differential attacks with more than one round covered by the key-guessing part, when some values coming from non-active S-boxes are needed after the first round in order to compute the differential transitions of the next rounds, automatically reducing the key-guessing complexity of the latter.

3.1 The case of linear cryptanalysis with FFT acceleration

Although our generalised approach can often reduce the time complexity of most key-recovery attack families, sometimes other accelerations may provide better results, and a method must be picked. This is the case of linear cryptanalysis when combined with the fairly common FFT acceleration of [9].

Consider a linear attack using a single approximation. The "naïve" implementation consists of counting for how many of the N plaintexts the approximation is zero for each of the 2^k guesses of the key (where k is the number of bits) by processing each combination individually. The time complexity is $O\left(N2^k\right)$.

We now construct a tree for the S-box layer with minLeaves $\leq 2^k$ (this automatically considers things like inactive S-boxes). From each plaintext, we can extract all the information from minLeaves key guesses. However, each leaf is associated to different key guesses depending on the value of the same bits in the plaintext. We thus have to keep a separate set of minLeaves key guess counters for each of these plaintext groupings. When all the data has been processed, we can filter promising partial key guesses (those which exhibit high correlation for part of the plaintexts) and separate them into full guesses until the complete guess with the highest counter can be located. This means we can reduce the time complexity of this kind of attack to $O\left(N \cdot \text{minLeaves}\right)$.

When the data complexity is large, we first distill the data into a table according to the bits which interact with the key (time complexity $O(N)$) and then guess all possible values of the key for each entry (time complexity $O\left(2^{2k}\right)$), as was first shown in [16]. If we apply guessing trees on the S-boxes, we find that for each of the minLeaves guesses of the key, we still have to look up 2^k entries of the table. The distillation table must work for every key guess, so its size can only be reduced to 2^{domopt}. The best time complexity reduction we can achieve on this attack algorithm is thus $O(N) + O\left(\text{minLeaves} \cdot 2^{\text{domopt}}\right)$.

Another common improvement to linear cryptanalysis makes use of the Fast Fourier Transform, and was introduced in [9]. By using the FFT in order to process the distilled data more quickly, the time complexity of the analysis phase can be reduced to $O\left(k2^k\right)$. Since the size of the distilled table cannot be reduced by using decision trees, we can only reduce this complexity to $O\left(\text{domopt}2^{\text{domopt}}\right)$.

The best approach here is to compute minLeaves and domopt for each S-box and find an optimal trade-off between these approaches (as we can use different techniques in each S-box), as we show with an example in the extended version of the paper [7].

3.2 Applications to differential cryptanalysis.

Intuitively, differential cryptanalysis improvements seem naturally more complex than linear ones, as in addition to possibly determining some values we need to determine some differences, and depending on the cases, several trees should be studied. This also implies that the gain can be quite significant.

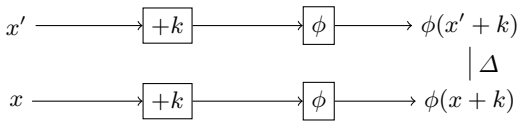


Fig. 3. Finding good pairs over one round of an iterated cipher.

Besides the case presented earlier covered by the natural case, there are other (usually coexistent) cases that often appear [5]: 1) given one plaintext x, determine another one x' that generates a certain difference Δ after the S-box ϕ along with an associated partial guessed key; 2) given pairs of plaintexts (x, x'), determine the ones that might generate a wanted difference Δ after ϕ; 3) given pairs of plaintexts (x, x'), determine the optimal partial key guess that ensures Δ after ϕ; 4) when at least two consecutive rounds are considered in the keyguessing, in any of the above cases we might need to know, in addition, the value of certain bits to verify the differential transition of further rounds; 5) when at least two consecutive rounds are considered, a key guess of a later round can be absorbed by a needed output defined by a linear equation.

We will next show how to use the S-box properties defined in the previous section to propose improvements in each of the 5 cases, while considering the example from Figure 3 for the three first cases.

Case 1: Input difference not determined. We are interested in determining x' such that $x' = \phi^{-1}(\phi(x+k)+\Delta)+k$. If we let $y = x+k$, the attacker can try to find inexpensive trees for

$$f_\Delta(y) = \phi^{-1}(\phi(y)+\Delta) = x'+k. \tag{1}$$

These trees allow the attacker to cheaply deduce bits of $x'+k$ by guessing a small amount of bits of $x+k$. Since the value of x is considered known, this is equivalent to guessing bits of k (the tree is the same for all values of x but different paths are taken for each value). Using this approach we only get information about $x'+k$ and about some bits of x', which correspond to the bits of k which were guessed.

An important limitation of using f_Δ is that some "evidently useful" relations might be missed, like for example if there is a differential $\delta \to \Delta$ through ϕ with probability 1, then $f_\Delta(y) = y+\delta$: by simply looking at the relations of f_Δ, it would seem that we need to guess all the bits of the key, but no key-guessing is necessary here since $x' = x+\delta$. In other words, decisions based on expressions of the form $\langle \gamma, (x'+k)+(x+k)\rangle$ are "key-free" and this can be incorporated into the search. A way to get trees with cost 0 is to apply the tree search algorithm to $F_\Delta(y) = f_\Delta(y)+y = x'+x$. The resulting trees provide "direct" information

[5] For the sake of simplicity, we will consider in this section that key-guessing rounds are done in the beginning, but everything can be applied similarly in the last rounds.

about x' (as x is known) and only require guessing the bits of key directly involved in the decision trees, as well as detecting completely free key guesses like the one described above.

Furthermore, when the key addition is shorter than the S-box size (like for GIFT), decisions on the same path involving only the same bits of the key but also some bits of the unaltered plaintext have no additional cost in the application, as the involved key bits cancel out. To deal with particular cases we simply use our algorithm to generate a list of optimal trees filtered in accordance to the individual requirements of the attack at hand, sometimes considering restricted functions. An application is described in Section 5 and an example can be found in the extended version of the paper [7].

Case 2: Preliminary sieving. Filtering wrong pairs is important as it often allows to reduce the time-complexity (and the noise) in attacks. We know that (x, x') is a wrong pair if $x + x'$ is not in the image of $F_\Delta(x) = f_\Delta(x) + x$. Note that the image of F_Δ is the same as the image of $y \mapsto \phi^{-1}(y) + \phi^{-1}(y + \Delta)$ and thus exactly corresponds to the possible input differences for the given output difference Δ. This idea has already been used in differential cryptanalysis already in the beginning [3] and also more recently like for instance [19], but many recent attacks do not use this despite the ample margin of improvement, as we show for instance in our GIFT applications in Section 5, where using this for filtering in the output already allows to reduce the complexity of the best known attacks.

Case 3: Fixed input difference. Suppose that we know the value of x (which is the case for external rounds of keyguessing) and that $x' = x \oplus \delta$ for a fixed δ (this is often the case in applications, since the difference of the pair is not key dependent).

Clearly, the possible input differences δ are given by the image of $(F_\Delta)^{-1}$. However, we can say more: a pair $(x \oplus k, x \oplus k \oplus \delta)$ satisfies

$$S(x \oplus k) \oplus S(x \oplus \delta \oplus k) = \Delta \tag{2}$$

with $x = x' \oplus \delta$ if and only if $x \oplus k \in (F_\Delta)^{-1}(\delta)$. Notice that $|(F_\Delta)^{-1}(\delta)|$ is in fact the DDT with input difference δ and output difference Δ.

Let us define the function $g^\delta_\Delta : \mathbb{F}_2^n \to \mathbb{F}_2$ such that $g^\delta_\Delta(x) = 0$ if and only if $x \in (F_\Delta)^{-1}(\delta)$. Our problem has now become equivalent to computing the value of $g^\delta_\Delta(x \oplus k)$ with as little information on k as possible: indeed, the best key-guessing strategy to determine whether a pair is a good pair is the one given by the optimal tree for $g^\delta_\Delta(x \oplus k)$ and the cost of this guess is given by the number of its leaves (minLeaves).

If we use this guessing strategy for each δ, we can drastically decrease the average guessing cost for determining whether a pair is a good pair. As an example, if we wanted to find what are the good pairs for the RECTANGLE S-box and $\Delta = 2$, this technique will allow to do so with an average guessing cost of 3 for each pair, instead of the 16 when using the naïve strategy, where for

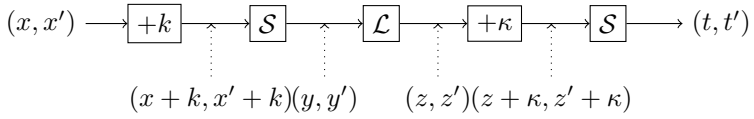
each possible value of k, one would compute Equation (2). A detailed example for the slightly more general transition ???? $\rightarrow$ 00?0 can be found in the extended version of the paper [7].

Case 4: Determining values in addition to good pairs. If we are mounting an attack with two or more consecutive rounds of key-guessing, then in the first round we do not only want to sieve the good pairs, but we also want to determine the values of one or more output bits of the plaintexts that form those pairs. To retrieve these bits in addition to the difference value we might need less bits than a whole key word.

This can be easily done by looking at the optimal tree of the output bits that we are interested in, where we fixed the first nodes based on what key bits have already been guessed to determine the output difference.

Case 5: Absorbing next round guessing. We can clearly apply the same method to determine the good pairs seen in Case 3 for later rounds in a chained manner. However, we have anticipated that it is actually not always necessary to determine this value for middle rounds, contrary to the previous cases, thanks to the following approach, that we call *key absorption*.

More concretely, consider the case of a two consecutive rounds of key-guessing, where we indicate as k the round key of the first round and κ the round key of the second one. Let $(x \oplus k, x' \oplus k)$ be the pair before going through the S-box layer $\mathcal{S}$ (which is a parallel application of S to each nibble) of the first round, and $(z + \kappa, z' + \kappa)$ be the pair before going through the second S-box layer, i.e. that we want to determine whether it is good or not for this second S-box layer.



Suppose for simplicity that, in order to determine the output difference of $S(z + \kappa) + S(z + \kappa)$, following the strategies explained in Cases 1 and 3, we need to determine the first bit

$$z_0 + \kappa_0 = \mathcal{L}_0(y) + \kappa_0 = \langle \alpha, y \rangle + \kappa_0,$$

of $z + \kappa$ only, where α corresponds to the first row of L. Doing a step-by-step guess would require to guess the key-bit κ_0 and compute $\langle \alpha, y \rangle$. Using the trees as explained above, we can make use of the case where $\langle \alpha, y \rangle$ depends linearly on a linear combination $\langle \gamma, k \rangle$ of key-bits of k. Instead of guessing all those key-bits we actually have to guess only their linear combination $\langle \gamma, k \rangle + \kappa_0$, i.e. only a single bit.

A detailed example can be found in the extended version of the paper [7].

3.3 Further extensions

When several rounds are taken into account in the key-guessing parts, the best interactions between the different trees need to be considered and carefully studied, which complicates the optimization of the application a bit. The automatically generated trees with the algorithm are particularly useful in these cases, which can become quite intricate. Some example of such applications can be found in Sections 5 and 6. In addition, the previous properties and techniques can be extended to other types of attacks, like for instance:

Differential-linear attacks. All the improvements of both differential and linear key-guessing parts will be applicable also to these type of attacks. See for example [6].

Rectangle and boomerang attacks. Using the properties of the S-box and of F_Δ for finding good pairs we can reduce the number of key guesses and total complexity. An example can be found in Section 5.

Meet-in-the-middle - sieve-in-the-middle. Though the framework is not the same as the attacks based on distinguishers we presented in the beginning, using the S-box properties that we enounced can allow to determine more known bits in the middle and therefore have a higher sieving probability, improving the complexity. To illustrate the principle of this improvement we provide a small improved attacks on 8-round PRESENT. The time complexity of the 8-round sieve-in-the-middle attack on PRESENT from [8] can be reduced from about $2^{73.42}$ to about $2^{72.91}$ full encryptions. We elaborated the details in the extended version of the paper [7]. In short, you can use the trees to derive more bits around the middle round after guessing the key and this decreases the sieving-probability.

4 Application to NOEKEON

In this section we describe the best known linear attacks on 12-round NOEKEON. NOEKEON is a 16-round block cipher which was presented by Daemen et al. ([10]) to the Nessie competition and has a block and key length of 128 bits. A short description of NOEKEON can be found in the extended version of the paper [7]. We denote the linear transformation (including shifts) by $\widehat{\theta}$. We can consider that the key is added to the state either before or after this linear transformation by considering an equivalent key.

Iterative linear trails of NOEKEON. Our attacks are based on iterative two-round linear trails with correlation 2^{-14}. Since all the transformations in a NOEKEON round except for the constant and key additions are invariant under rotation, we can obtain new trails from known ones by rotation and round swapping. We have identified four families of trails, shown in figure 4.

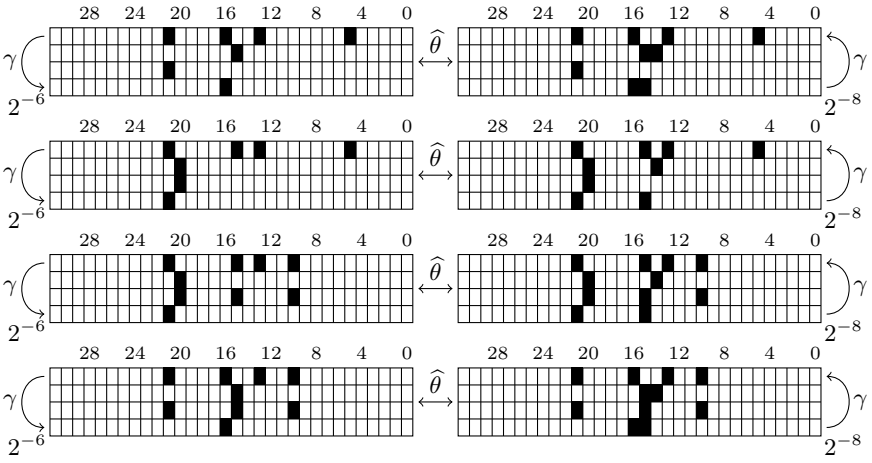
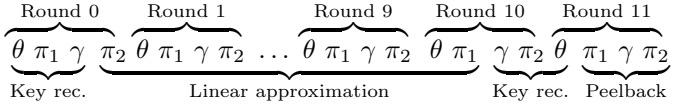


Fig. 4. Four two-round iterative linear trails of NOEKEON.

4.1 Attacks on reduced-round NOEKEON (without relations)

A 12-round linear attack on NOEKEON is sketched by its designers in [10]. An iterative trail is extended to nine rounds with correlation 2^{-62}. The trail is used as a distinguisher between rounds 1 and 9 to mount a 12-round linear attack with the following key recovery structure:



We guess 24 bits of the transformed keys after $\widehat{\theta}$ in round 0 and before $\widehat{\theta}$ in round 11, or 48 in total. The data complexity is around $2^{62 \cdot 2} = 2^{124}$ known plaintexts. If a distillation table is used as in [16], the time complexity is $2^{124} + 2^{48 \cdot 2} = 2^{124}$.

4.2 Attacks on reduced-round NOEKEON (with relations)

We propose a 12-round attack which modifies the nine-round distinguisher (using the first iterative linear trail) which will reduce the data complexity to 2^{119}. This improvement in correlation is achieved by modifying the linear trail in two ways:

- In the first round, we remove S-box 15 from the approximation (so that the input mask is "staggered"), increasing the correlation by a factor of 2^2.
- In the last round we substitute the S-box 15 approximation from $\mathtt{2} \to \mathtt{2}$ to $\mathtt{2} \to \mathtt{b}$, the correlation changes from 2^{-2} to 2^{-1}.
- We also modify the other transitions in the first and last rounds in order to reduce the number of active S-boxes in the key recovery.

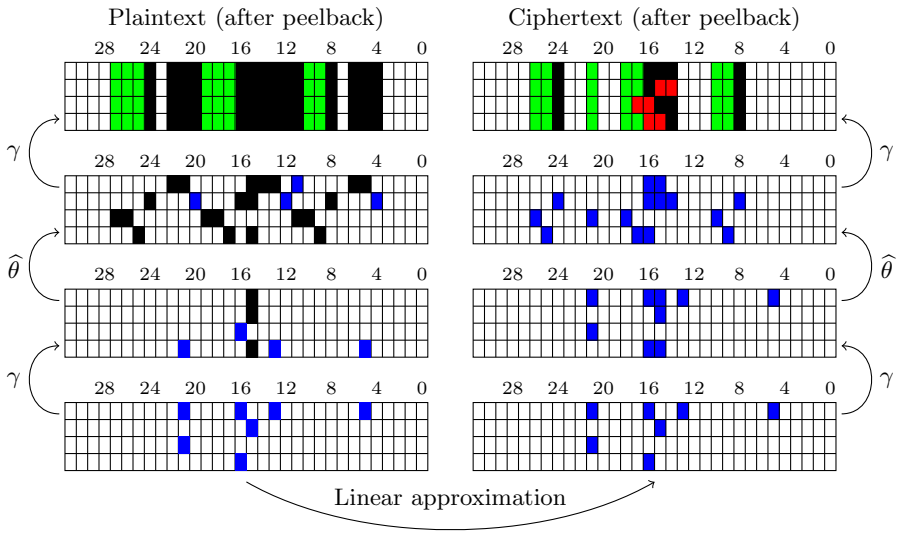


Fig. 5. Attack on 12-round NOEKEON with 2^{119} data and $2^{124.5}$ time complexity.

The correlation of the linear trail increases from 2^{-62} to 2^{-59}. However, in a key recovery attack, we would need to guess 92 key bits in the first round, 4 in the second, and 48 in the last. Even with the FFT techniques of [9] and [13], the time complexity surpasses 2^{144}. We look at the properties of the S-box:

- S-box 15 in the second round: if we only know x_0, x_1 and x_3, y_1 can still be computed with probability 1/2. We can thus ignore input bit x_2, which doubles the data complexity (we'll reject plaintexts for which x_2 would be used) but reduces the active bits and S-boxes in the first round.
- In the first and last rounds, whenever we need y_2 or y_3 at the output of an S-box, which happens for 8 S-boxes in the first round and 7 in the last, we can reduce the key guess by one bit because domopt = 3.

The key guess is now 124 bits. If we apply the FFT algorithm directly, the key recovery cost would be $121 \cdot 2^{124} \simeq 2^{130.9}$ additions. It can be decreased by using Walsh transform pruning as described in [13]. There are three key bits repeated in the first and second rounds, as well as six last round key bits which can be deduced from the first round. The time complexity can thus be reduced to

$$2^3 \cdot \left(2^{121} + (121 - 9)2^{121-9}\right) \simeq 2^{124.29} \text{ additions.}$$

The details of the key recovery are specified in figure 5. Blue bits represent the masks of the linear approximation, while the active bits for the key recovery are black. The S-boxes where domopt = 3 are in green, while the red bits on the last round can be deduced from the first round key guess.

We must also compare the costs of additions and a 12-round NOEKEON encryptions. A conservative estimate[6] is at least 3840 bit operations for an encryption. An addition of 3·128-bit integers takes around 768 bit operations. Therefore its cost is at most one fifth of the cost of an encryption. The full time complexity is thus $2^{119} + 0.2 \cdot 2^{124.29} \simeq 2^{122.14}$ encryptions.

Overall, the new attack has a data complexity of 2^{119} and a time complexity of $2^{122.14}$, which is as far as we know the best on 12-round NOEKEON. The best attacks without relations have 2^{124} data and time complexity.

5 Application to GIFT

In this section we describe an improved version of the attack presented in [14]. This related-key rectangle attack is the known attack which reaches the most rounds of GIFT-64 (25). We apply our improved key-guessing techniques in order to improve its complexity. The section is structured as follows: We provide a brief description of GIFT, next we present the original attack, and we propose two ways of improving its complexity in the two last subsections.

5.1 Description of GIFT-64

GIFT-64 is a block cipher first introduced in [2] of block size 64 and key length 128. The 64-bit state consists of 16 4-bit nibbles which will be denoted by $b_{63} \dots b_0 = x_{15} \| \dots \| x_0$. Each round consists of three steps: the application of a 4-bit S-box, a bit permutation, and the addition of a 32-bit subkey.

The GIFT S-box. The GIFT S-Box is given as a lookup table.

$\boldsymbol{x}$	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
$\boldsymbol{S(x)}$	1	a	4	c	6	f	3	9	2	d	b	7	5	0	8	e

Bit permutation and key addition. As a linear layer, GIFT uses the permutation

$$P_{64}(i) = 4 \left\lfloor \frac{i}{16} \right\rfloor + 16 \left(\left(3 \left\lfloor \frac{i \bmod 16}{4} \right\rfloor + (i \bmod 4) \right) \bmod 4 \right) + (i \bmod 4) \,.$$

GIFT-64 uses 32-bit round subkeys which are XORed to the bit positions of the state of the form $b_{4i}, b_{4i+1},\ i = 0, \dots, 15$ (that is, the two rightmost bits of each S-box before the non-linear layer). We won't detail the keyschedule as it won't be used in the attack.

5.2 The best previous attack on GIFT-64 ([14])

We now describe the attack on 25-round GIFT-64 from [14], which is a related-key rectangle attack. The 20-round boomerang distinguisher can be found in [14]. We just need to know that its probability is $2^{-n}\hat{p}^2\hat{q}^2 = 2^{-64} \cdot 2^{-58.557}$.

Table 2. ([14], Table 5) The related key rectangle attack on 25-round GIFT-64.

Plaintext	???? ???? ???? ???? ???? ???? ???? ???? ???? ???? ???? ???? ???? ???? ???? ???? #0
R1 After S	??0? 1??0 01?? ?0?? 1?0? ?1?0 0??? ?0?? ??0? ???0 0??? ?0?? ??0? ???0 0??? ?0?? #1
After P, K	???? ???? ???? ???? 0000 0000 0000 0000 11?? ???? ???? ???? ???? 11?? ???? ???? #2
R2 After S	0?01 00?0 000? ?000 0000 0000 0000 0000 0100 00?0 000? ?000 ?000 0100 00?0 000? #3
After P, K	???? 0000 ?1?? 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 ?1?? #4
R3 After S	1000 0000 0010 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0010 #5
After P	0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0010 1010 0000 0000 0000 #6
	20-round rectangle distinguisher
R24 Before S	0000 0100 0000 0000 0000 0000 0000 0000 0000 0001 0010 0000 0001 0000 0000 0000 #7
Before P, K	0000 ???1 0000 0000 0000 0000 0000 0000 0000 ???? ???? 0000 ???? 0000 0000 0000 #8
R25 Before S	00?0 0000 00?? 0?00 0001 0000 ?00? 00?0 ?000 0000 ??00 000? 0?00 0000 0??0 ?000 #9
Before P, K	???? 0000 ???? ???? ???? 0000 ???? ???? ???? 0000 ???? ???? ???? 0000 ???? ???? #10
Ciphertext	??0? ??0? ??0? ??0? ???0 ???0 ???0 ???0 0??? 0??? 0??? 0??? ?0?? ?0?? ?0?? ?0?? #11

The key recovery extends the distinguisher by three rounds at the top and two rounds at the bottom and can be found in Table 2.

The authors build a key-recovery attack by applying the model from [23] to the external rounds. We start with the initial difference right before the first key addition, numbered #2 in Table 2. We have $r_b = 44$ (? bits in #2), and $m_b = 30$ (active key bits in the differential transitions of the initial rounds), $r_f = 48$ (? bits in #11), $m_f = 32$ (involved key bits in the differential transitions of the final rounds). Let $s = 2$ be the expected number of good quartets per structure. The attack proceeds as follows:

1. Build $y = \sqrt{s}\frac{2^{n/2-r_b}}{\hat{p}\hat{q}} = 2^{17.79}$ structures of $2^{r_b} = 2^{44}$ plaintexts. Encrypt each plaintext four times, using the four keys $K_1 = K$, $K_2 = K \oplus \Delta$, $K_3 = K \oplus \nabla$ and $K_4 = K \oplus \Delta \oplus \nabla$. For each structure j, we obtain four lists L_1^j, L_2^j, L_3^j and L_4^j, which we sort by the r_b active bits in #2.
2. We guess the m_b bits of the first two round subkeys as K_b. For each guess:
   (a) For each structure, we partially encrypt all the plaintexts of L_1^j until #6 using K_b, we add the difference α from the rectangle path, and partially decrypt back to #0 with $K_b \oplus \Delta$. We find the plaintext in L_2^j which matches it. After doing this for all the structures, we obtain a list S_1 which contains $y \cdot 2^{r_b}$ pairs with the right input difference at the distinguisher. We repeat this with lists L_3^j and L_4^j to obtain S_2. We sort S_1 and S_2 according to the non-active bits of the ciphertexts.
   (b) We go through S_1 and S_2 to find all collisions in the non-active bits of the ciphertexts. We obtain a list S_3 of $y^2 \cdot 2^{2r_b+2r_f-2n}$ candidate quartets.
   (c) For each guess of the m_f bits of key K_f, we examine each candidate in S_3 to see how many satisfy the rectangle distinguisher. As we can guess and filter S-box by S-box (detailed in [23]), the cost is negligible.
   (d) Keep the $h = 22$ values of K_f with the most conforming quartets, and find the correct one with an exhaustive search over the rest of the key.

[6] 128 operations per S-box layer or key addition, 64 operations per linear layer.

The data complexity of the attack is $D = 4 \cdot y \cdot 2^{r_b} = 2^{63.78}$ chosen plaintexts. The time complexity is

$$T = 4 \cdot y \cdot 2^{r_b} + 2^{m_b} \left(3 \cdot y \cdot 2^{r_b} + y^2 \cdot 2^{2r_b + 2r_f - 2n} \cdot \frac{4}{25} \right) + 2^{k-h} \simeq 2^{120.92}$$

encryptions with a success probability of 74%.

5.3 S-box properties in the first rounds for better sieving

We now explain how to gain 6 bits in time complexity and slightly improve the data complexity. The improvement is quite technical, but it can be summarized as modifying the way we build the structures using the S-box properties. The aim of organising the plaintexts in structures is for each one to produce enough rectangle quartets so that we obtain enough in total. By taking all the possible values for the active bits of the plaintext and partially encrypting forwards and backwards, each possible guess of K_b will map one entry of L_1^j to an entry of L_2^j. Each structure thus produces exactly 2^{r_b} pairs which verify the input difference α. As can be seen in the formula of y, this is discounted from the total number of structures. By exploiting the properties of the S-boxes we can reduce the size of the structures as well as the number of key bits m_b, which will allow us to reduce the time complexity, and potentially the data. For computing the new needed number of structures, y', we won't use the same formula as before, as the elements in the lists will have some particularities now, but instead will deduce the new value of y' from the wanted expected number of good quartets, S, and from carefully computing how many potentially good pairs and quartets we keep in each list with the new type of structures.

Finding S-box properties. We applied the tree search algorithm on $F_\Delta = f_\Delta + x$ for all output differences and filtered them according to two criteria. First, we wanted only one of the two key bits to be involved. We also forced at least one subtree on level 2 not to be of full depth to reduce the search space.

Property of $f_{(0010)_2} + x$. The most interesting tree we obtained was in the case of $f_{(0010)_2} + x$, where the following relation appeared:

$$x_0 = x_3 = 0 \implies F_2(x) = 2.$$

It is useful with transitions of the form ???? $\rightarrow$ 00?0, which appear in S-boxes 1, 6 and 14 at round 2. In particular, it implies that guessing the key bit added to x_1 is not *a priori* necessary. The aim is to build smaller structures where these properties are verified, and to guess less key bits, which will in turn reduce the time complexity (the number of quartets to try stays the same, but the number of guesses decreases). From now on, we consider that all the data has bit x_3 of the input to S-boxes 1, 6 and 14 at R2 fixed to zero. Intuitively, though the number of structures available is tight, guessing less key bits implies a relaxation of the conditions, and this in turn implies proportionally more kept pairs and quartets.

Reducing the bits in round 3. We can also show that it is unnecessary to guess the bit k_1 in the three active S-boxes of round 3. In essence, not all the pairs in S_i will necessarily have this bit determined, which will allow us to keep more quartets while guessing less bits.

The output differences of the three S-boxes can take two values, which in turn affect the input differences of the active S-boxes in the third round (0, 13 and 15). We need to carefully compute how many pairs will verify the input difference α when guessing six less key bits than before. The transitions of S-boxes 0 and 13 in round 3 are $?1X? \rightarrow 0010$, where X depends on the transitions from round 2 where the key guess was reduced, and can thus be active or not. In S-box 15, the transition is $??X? \rightarrow 1000$.

Let us examine how we build the differential pairs from the lists L_i^j. After guessing the key bits associated to all the active S-boxes of round 2 but 1,6 and 14, we can compute, for each plaintext, the three bits x_0, x_2, x_3 at the input to S-boxes 0, 13 and 15. Choosing the value of Δx_1 for each S-box determines the other plaintext so that the pair generates α. *A priori* this should produce 2^3 different plaintexts, but we should note:

- When the input bit x_0 (which is known) of the three round 2 S-boxes 1,6,14 is 0, $F_{(0010)}$ is independent of x_1.
- In order to exploit the property efficiently, we will only consider pairs of plaintexts for which $x_0 = 0$ for S-boxes 1, 6 and 14. The property therefore always holds (as we also have $x_3 = 0$) and we can focus on the active S-boxes in the third round.
- Each element of the list L_i^j will have a different number of associated plaintexts in the other list, and each pair will have determined one additional key bit value per treated transition (so three in total). When looking at just one S-box, for the sake of simplicity, this bit will not be the same for each pair: some will exclusively determine the associated bit from round 3, which are the ones involving a difference value in round 2 or a non difference value but a 0 in the round 2 position 0 S-box, and some will determine the xor of the not-guessed key bit of round 3 with the not-guessed bit from round 2 of the related S-box: when the bit at position 0 of the S-box at round 2 takes a value one, both values 1 or 0 are possible in the output at position 1, while only one value is possible when $x_0 = 0$.

Taking this into account, we can now say that the transitions of round 3 of S-box 0 (or 13 that will behave the same way), for all the possible 2^3 values of the 3 known input bits, 3 cases will imply that no difference exists at position 1 (no matter the value of bit at position 1), 3 cases imply that there is always a difference and two cases imply that depending on the value of the bit at position 1 there will or there will not be a difference.

So for one S-box, for each input pair, we have a number of possible pairs from L_2 to be associated to L_1 that is:

$$1/8(3(1/2 \cdot 2 + 1/2 \cdot 2) + 3(1/2 \cdot 2) + 2(1/2 \cdot 2 + 1/2 \cdot 1)) = 1.5.$$

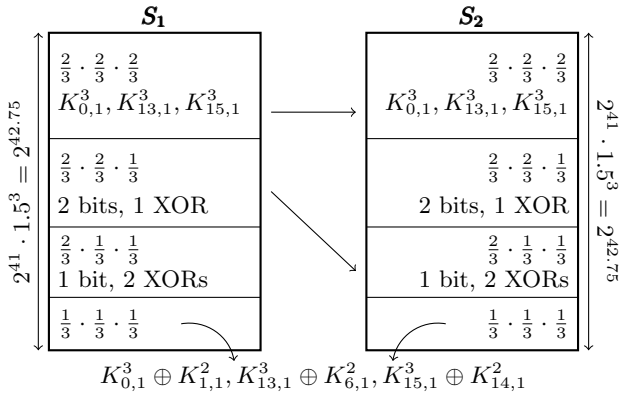


Fig. 6. Representation of the lists S1 and S2 of size $2^{42.75}$ and the distribution of their elements. In each chunk we can see: 1) the proportion of their size (the first for instance has a size of $\frac{2^3}{3^3}2^{42.75}$ as well as 2) the bits that are determined for these pairs from rounds 2 and 3. When two bits are xored, this can be seen as the bits of values $K^3_{0,1}, K^3_{13,1}$ and $K^2_{14,1}$ are three absorbed bits: κ_1, κ_2, and κ_3. In order to build list S_3, we consider the subset of the crossproduct of all the elements of each list that verify the output conditions and additionally that has the same value when some identical key bits of information have been determined, as otherwise it would imply and impossible quartet.

The previous amount includes pairs generated when the bit x_0 of the input of the associated S-box of round 2 is 0 or 1. As we saw in the previous facts, that will change the key bits that become implicitely determined from the formed pairs (bit from round 3, or xor of this with the bits from round 2). Let us separate the previous amount regarding this: $1.5 = 1/8(3+2+3) + 1/8(3+1) = 8/8 + 4/8$, which implies that in 2/3's of the cases the bit from round 3 will be determined, and 1/3 it will be the xor of bit, which have no incompatibilities between them.

Regarding the transitions of round 3 of S-box 15 we have a different distribution of the cases, but it is easy to check that we arrive at the same configuration of 2/3 and 1/3.

The lists S_1 and S_2 that we obtain this way are represented in figure 6. The structures we build in this new attack will have size of $2^{44-3} = 2^{41}$, as the bit at position 3 of the 3 considered S-boxes are fixed to 0. The size of S_1 and S_2 is given by $2^{41} \cdot 1.5 \cdot 1.5 \cdot 1.5 = 2^{42.75}$. We now just have to compute the exact number of compatible pairs that we can obtain from merging both lists before taking into account the output conditions. This number that we will call P will have to verify later (where $2^{y'}$ will be the new number of structures that we need to compute now):

$$y' = y2^{r_b}/\sqrt{P} = 2^{17.78+44}/\sqrt{P}.$$

By looking at the properties of the different chunks in Figure 6 and all their possible crossproducts, that will determine how many common key bit conditions that will filter they have, we can compute P as:

$P = 2^{42.75}[2^3/3^3 \cdot 2^{42.75}(2^3/3^3 \cdot 2^{-3} + 3 \cdot 2^2/3^3 \cdot 2^{-2} + 3 \cdot 2/3^3 \cdot 2^{-1} + 1/3^3) + 3 \cdot 2^2/3^3 \cdot 2^{42.75}(2^3/3^3 \cdot 2^{-2} + 2^2/3^3 \cdot 2^{-3} = 2 \cdot 2^2/3^3 \cdot 2^{-1} + 2/3^3 + 2^2/3^3 \cdot 2^{-2} + 1/3^3 \cdot 2^{-1}) + 3 \cdot 2/3^3 \cdot 2^{42.75}(2^3/3^3 \cdot 2^{-1} + 2^2/3^3 + 2 \cdot 2^2/3^3 \cdot 2^{-2} + 2/3^3 \cdot 2^{-3} + 2 \cdot 2/3^3 \cdot 2^{-1} + 1/3^3 \cdot 2^{-2}) + 1/3^3 \cdot 2^{42.75}(2^3/3^3 + 3 \cdot 2^2/3^3 \cdot 2^{-1} + 3 \cdot 2/3^3 \cdot 2^{-2} + 1/3^3 \cdot 2^{-3})]$

$$\Rightarrow P = 2^{85.5} \cdot 2^{-9.509} \cdot 2^{8.09} = 2^{84.09}$$

And therefore we can compute the needed y':

$$y' = 2^{17.78+44-(84.09/2)} = 2^{19.73}.$$

We have now an improved data complexity of $D = 4 \cdot 2^{19.73} \cdot 2^{41} = 2^{62.73}$, instead of $2^{63.78}$ previously. Please note that the data limit here is $4 \cdot 2^{64}$, we are encrypting each plaintext with 4 different keys, and that the limit of y is 2^{20}.

The time complexity will become:

$$T = 4y' \cdot 2^{41} + 2^{m_b-6}(3y'2^{41} + 2^{2*19.73}2^{84.09}2^{-2(n-r_f)})2^2/25 = 2^{114.92}$$

instead of $2^{120.92}$ with the same success probability.

5.4 Using S-box properties in the final rounds for better sieving

We use our improved key-guessing techniques to improve the complexity of the previous attack. This idea will improve the overall complexity by reducing the size of r_f, which in turn reduces the size of S_3 and therefore of the quartets to try.

If we now have a look at the final rounds, we can see that the rightmost S-box need to verify a transition of ???? to ?000 through S^{-1}. That means that this input difference can be 0 or 8 at the end of round 24. If the difference is 0, we have 4 additional conditions when building up the quartets and we will sieve more of them, if the difference is 8, then, by looking at the image of $F_8(X_3, X_2, X_1, X_0) = S(S^{-1}(X_3, X_2, X_1, X_0) \oplus (1000)) \oplus (X_3, X_2, X_1, X_0)$, we see it can only take four values : $(3, 7, F, B)$. In total, with the zero difference is a total of 5, that leaves an additional factor of $(4+1)/16 = 2^{-1.67}$. We can do the same with the transitions ???? to 0?00 and ???? to 00?0 (that appear each two times) and add a sieving factor of $(5+1)/16 = 2^{-1.41}$ and of $(6+1)/16 = 2^{-1.19}$ respectively per transition. Transition ???? to ?010 has a factor of $(6+6-4)/16 = 2^{-1}$

Step 2(c), that before kept $2^{91.56}$ quartets to try, was the bottleneck when multiplied by the 2^{30} complexity of guessing m_b. We will see now how this amount of quartets can be reduced:

$$2^{91.56}(2^{-1.67})^2(2^{-1.41})^4(2^{-1.19})^4(2^{-1})^2 = 2^{91.56-15.74} = 2^{75.82},$$

where the first factor corresponds to the F_8 relations, and it is squared as it has to be verified by both of the pairs that form a quartet, the second factor correspond to the relations of F_4, that appears twice and should also be squared, which gives a power of 4, and the same goes for the third factor from F_2. The fourth one that comes from the relation from transition ???? to ?010 where the non-zero difference is not an option, and needs to be squared because of the two pairs.

This $2^{75.82}$ will be the new cost of this step (multiplied by 2^{30} gives $2^{105.82}$ instead of $2^{121.56}$), as we can directly check the values from S_2 that have a difference that belongs to the image of their corresponding F_i, which means that we have reduced the complexity by a factor $2^{15.74}$. Thanks to the trees of F_i step 2(d) could become slightly smaller than 2^2, but as the gain would be very small we won't detail it here (but we point out to consider this in other scenarios where it could help).

When taking into account the factor of the computations for the attack compared to an encryption we obtain a final complexity of $2^{105.18}$ instead of $2^{120.92}$.

5.5 Combining both.

As both improvements consider independent parts of the attack, they can both be taken into account, generating a new improved time complexity of $2^{114.92-15.74} = 2^{99.18}$ and data complexity of $2^{62.73}$, improving time by a factor bigger than 2^{21}, and data by a factor of 2.

6 Application to RECTANGLE-80

In the present section, we want to improve the best attack on the updated version of the SPN cipher RECTANGLE-80 [22] which, to the best of our knowledge, is the differential attack presented by the authors of the cipher themselves in the same paper.[7]

A description of the cipher can be found in the extended version of the paper [7].

In this section (following the same framework used in [19]), we will indicate the round key i as K_i, the input of the S-box layer at round i as I_i and the output of the S-box layer of round i as O_i. This means that the output of the ShiftRow operation at round i is I_{i+1}. Similarly, we will call $\Delta I_i, \Delta O_i$ the respective differences of the state of a given a pair. We will sometimes indicate a vector of $\mathbb{F}_2^4$ as an hexadecimal number.

[7] A differential attack that requires less data is claimed by the authors of [1] thanks to a distinguisher that covers the same number of rounds with better probability. However, no description or time complexity of the attack was given and we could not verify it due to the large time complexity of the key-guessing phase. We believe that, with the techniques presented in this paper, it could be possible to make the attack work, but the time and memory complexity would still be much worse than the attack we present here.

Table 3. The differential attack on 18 rounds of RECTANGLE-80 with the distinguisher from [22]. The $\emptyset$ indicates an active bit with difference 0. The ? indicates an unknown difference value.

$\Delta P = \Delta I_0$	???? 0000 0000 ???? ???? ???? 0000 0000 ???? ???? 0000 0000 0000 0000 0000 $\emptyset\emptyset\emptyset\emptyset$
ΔO_0	?000 0000 0000 000? 01?0 ?000 0000 0000 000? 0010 0000 0000 0000 0000 0000 0$\emptyset$00
ΔI_1	0000 0000 0000 ?$\emptyset$?? 0000 0000 0000 0000 ?11? 0000 0000 0000 0000 0000 0000 0000
ΔO_1	0000 0000 0000 0110 0000 0000 0000 0000 0010 0000 0000 0000 0000 0000 0000 0000
ΔI_2	0000 0000 0010 0000 0000 0000 0000 0110 0000 0000 0000 0000 0000 0000 0000 0000
	14-round differential distinguisher
ΔI_{16}	0000 0000 0000 0100 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0010 0000
ΔO_{16}	0000 0000 0000 ??11 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 ???? 0000
ΔI_{17}	0000 ?000 0?10 0001 0000 0000 ?000 0?00 0000 0000 0000 0000 0000 0?00 000? 0000
ΔO_{17}	0000 ???? ???? ???? 0000 0000 ???? ???? 0000 0000 0000 0000 0000 ???? ???? 0000

6.1 Description of the attack

As already mentioned, we want to improve the best attack on the new variant of RECTANGLE-80, which is a differential attack presented by the authors themselves with time complexity of $2^{78.67}$ 18-round encryptions, data complexity of 2^{64} and memory complexity of 2^{72} key counters [22]. The bottleneck in this attack is given by the large amount of key-guessing needed in the two rounds before and after the distinguisher.

We will show here how to cover these rounds of key-guessing using the framework presented in [19] and how to reduce the complexity of the key-guessing phase by a factor of about 2^{28}, thanks to the techniques introduced in Section 3.2.

During the key-guessing phase, we gradually guess the necessary bits of the round key K_i nibble by nibble. We will actually simplify the guessing done to determine good pairs with respect to Case 3 of Section 3.2, since this is anyway going to make the key-guessing phase complexity negligible with respect to that of the data collection phase. In particular, any time we have to guess whether a pair $(x \oplus k, x \oplus \delta \oplus k)$ determines an output difference Δ through S, we will not do a gradual guess depending on the value of x using the tree of g^{δ}_{Δ}, as would be preferable. Instead, we will simply guess, whatever the value of x, the bits of k given by the inner nodes of g^{δ}_{Δ} (or, equivalently, by Dom). The guessing cost, then, will be 2^{domopt} and not numLeaves. An example of this simpler guessing strategy is given in the extended version of the paper [7].

In order for the attack to work, we need to prepare enough pairs of plaintext that can satisfy the desired input difference ΔI_2, so that at least $2^{62.83} = p^{-1}$ such pairs are available to the distinguisher starting at round 2 (see [3] for the heuristic).

To this end, we prepare a number of 2^y of data structures, each containing all possible plaintexts with a fixed value in the non-active bits in ΔI_0. Since

the linear layer of RECTANGLE-80 is a permutation of the bits, it is easy to see from Table 3 that the amount of active bits in the first round is 24, i.e. the number of ?. Thanks to the properties of S, we can see that the real number of active bits is actually 23: in fact, for S-box 6 of I_0 we only need to determine the active output bit y_1 of O_0, in addition to the good pairs, and from the trees of y_1 and F_2 we see that their actual domain is generated by the vectors $\mathtt{1},\mathtt{2}$ and $\mathtt{8}$, implying that the bit at position $\mathtt{4}$ from S-box 6 of a plaintext won't affect at all the key-guessing, i.e. it is not active.

Therefore, from each data structures we can generate 2^{23} plaintexts, by letting the active bits vary through all the possible values (while keeping fixed the non-active ones) and build a maximum of 2^{45} ordered pairs. In order to determine the necessary number of structures, we see that for a fixed key guess, we expect 2^{y+22} pairs to lead to the desired input difference ΔI_2: this means that we want $y + 22 - 62.83 \geq 0$, i.e. $y = 41$.

However, by looking at the possible values that the states ΔP and ΔO_{17} can take, we can sieve the pairs to use in the key-guessing phase and keep, on average, $2^{5.71}$ pairs for each structure (see the extended version of the paper [7] for details).

Step 1 (guess of K_0 to determine the good pairs of round 0 and retrieve linear relations for the active bits of O_0) We gradually guess nibble by nibble the necessary amount of key material to determine whether each plaintext pair is a good pair and retrieve linear relations that describe the active bits of O_0 in terms of K_0 (the latter are necessary for key absorption). Just as an example, in order to guess the relevant key-material for S-box 7, we can compute both the good pairs and the linear relations for the active output bit y_0 (necessary for the key absorption in Step 2) with an average number of key-guesses of $2 \times 1/8 + 7 \times 7/8 = 2^{2.73}$. In fact, in case the input difference is $\delta = \mathtt{0}$ (which we expect to happen for 1/8 of the pairs) we only need to guess one key-bit of K_0 to find a linear relation of y_0 (as suggested by the optimal tree for $\langle S(x), \mathtt{1}\rangle$); if $\delta \neq \mathtt{0}$ (which we expect to happen for 7/8 of the pairs), we need to make 7 guesses to determine which pairs are good (thanks to the tree for g_1^{δ}); indeed, these guesses are always enough to also determine a linear relation on y_0 and we need to guess no further. After that, we can sieve all the pairs such that

$$S(x \oplus K_0) \oplus S(x \oplus \Delta + K_0) \neq \{\mathtt{1}, \mathtt{0}\},$$

which happens with a probability of $2/8 = 2^{-2}$. Notice that the verification of this condition costs $2 \times 1/18 \times 1/16$ 18-round encryptions for each pair. Overall, this process is applied to each nibble, for a total time complexity of this step is $2^{y+4.90}$ 18-round encryptions.

Step 2 (guesses of K_1, K_0 to determine the pairs that satisfy ΔO_1) In this step, we guess the remaining key bits to ensure the right difference after the first two rounds. First, we notice that we can discard any pair which has not an input difference that could lead to ΔO_1, by looking at F_{Δ} for S-box 7 and 12, and find

out that we can keep only 3/4 of the remaining pairs so far. Thanks to the key absorption technique of Section 3.2, we can jointly guess an average of 2 bits of (K_0, K_1). Notice that deciding whether a pair satisfies the transition of S-box 7 is independent of the third input bit (and therefore of the second output bit of nibble 13 in round 17, as was anticipated), thanks to the fact that F_2 has domopt = 3 (i.e. independent of bit 4). This also implies that output bit y_2 of S-box 11 of round 0 does not need to be guessed. The total complexity of this step is then $2^{y+4.52}$ 18-round encryptions.

Step 3 (guess of K_{18} to determine the pairs that satisfy ΔI_{17} and retrieve linear relations for the active input bits of I_{17}) As was done in Step 1, we want to filter the good pairs by gradually guessing the necessary key material for each S-box and retrieve linear relations for the active input bits of I_{17}. In the hypothesis that the values of the active bits of O_{18} are uniformly distributed, we expect an average complexity of this step of $2^{y+8.98}$ 18-round encryptions.

Step 4 (guess of K_{17}, K_{18} to determine the pairs that satisfy ΔI_{16}) As done in Step 2, we first sieve all the pairs whose output difference cannot lead to a good pair, using as before F_Δ, and then do a combined guess of K_{17} and K_{18} with key absorption. As before, we notice that determining good pairs through S-box 12 is independent of the second input bit (and therefore of the second output bit of nibble 13 in round 17) by looking at F_2. The total complexity of this step of $2^{y+7.42}$ 18-round encryptions.

Final complexity The time complexity for the key-guessing is about $2^{y+9.50} = 2^{50.50}$ 18-round encryptions, which means that the bottleneck is no longer the key-guessing, as was in the attack of [22]. Together with the data collection phase, the time complexity of the attack is then 2^{64} 18-round encryptions.

7 Conclusion

Using our description of S-boxes as decision trees allows us to improve the best known attacks against Noekeon, GIFT, and RECTANGLE . These attacks belong to different families, yet our general framework to optimized the key-guessing part has been applied to all of them.

As future work, it might be of interest to attempt to handle larger functions, that is, with more input bits. For now, all the applications shown above require some degree of manual analysis of the trees (e.g. when combining several rounds in the GIFT or RECTANGLE application). A more heuristic search for the trees might produce trees for significantly larger functions, thus analyzing more than one S-box or even more than one round. This would have the potential to automatically include many of the manual improvements.

In addition, understanding the general behaviour of the minimal number of leaves is an interesting problem on its own. A non-trivial upper bound on the minimal number of leaves for an arbitrary (balanced) Boolean function of n bits would be of great interest.

We expect that many other attack scenarios will benefit from our framework for gradually performing the key-guessing using binary trees, improving other attacks complexities, as it is quite generic.

Acknowledgment

This work was partially funded by the DFG, (German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA – 390781972. We would further like to thank Shahram Rasoolzadeh for his valuable support. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement no. 714294 - acronym QUASYModo).

References

1. Ankele, R., Kölbl, S.: Mind the gap - A closer look at the security of block ciphers against differential cryptanalysis. In: Cid, C., Jr., M.J.J. (eds.) SAC 2018. LNCS, vol. 11349, pp. 163–190. Springer (2018)
2. Banik, S., Pandey, S.K., Peyrin, T., Sasaki, Y., Sim, S.M., Todo, Y.: GIFT: A small present - towards reaching the limit of lightweight encryption. In: Fischer, W., Homma, N. (eds.) CHES. LNCS, vol. 10529, pp. 321–345. Springer (2017)
3. Biham, E., Shamir, A.: Differential cryptanalysis of DES-like cryptosystems. In: Menezes, A., Vanstone, S.A. (eds.) CRYPTO '90. LNCS, vol. 537, pp. 2–21. Springer (1990)
4. Blondeau, C., Gérard, B., Nyberg, K.: Multiple differential cryptanalysis using LLR and χ^2 statistics. In: Visconti, I., Prisco, R.D. (eds.) SCN 2012. LNCS, vol. 7485, pp. 343–360. Springer (2012)
5. Bogdanov, A., Knudsen, L.R., Leander, G., Paar, C., Poschmann, A., Robshaw, M.J.B., Seurin, Y., Vikkelsoe, C.: PRESENT: an ultra-lightweight block cipher. In: Paillier, P., Verbauwhede, I. (eds.) CHES 2007. LNCS, vol. 4727, pp. 450–466. Springer (2007)
6. Broll, M., Canale, F., David, N., Florez-Gutierrez, A., Leander, G., Naya-Plasencia, M., Todo, Y.: Further improving differential-linear attacks: Applications to chaskey and serpent. Cryptology ePrint Archive, Report 2021/820 (2021), https://ia.cr/2021/820
7. Broll, M., Canale, F., Leander, G., Gutiérrez, A.F., Naya-Plasencia, M.: Generic framework for key-guessing improvements. Cryptology ePrint Archive, Report 2021/1238 (2021), https://ia.cr/2021/1238
8. Canteaut, A., Naya-Plasencia, M., Vayssière, B.: Sieve-in-the-middle: Improved MITM attacks. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013. LNCS, vol. 8042, pp. 222–240. Springer (2013)
9. Collard, B., Standaert, F., Quisquater, J.: Improving the time complexity of Matsui's linear cryptanalysis. In: Nam, K., Rhee, G. (eds.) ICISC 2007. LNCS, vol. 4817, pp. 77–88. Springer (2007)
10. Daemen, J., Peeters, M., Assche, G., Rijmen, V.: Nessie proposal: Noekeon (2000)
11. Eichlseder, M., Kales, D.: Clustering related-tweak characteristics: Application to MANTIS-6. IACR Trans. Symmetric Cryptol. **2018**(2), 111–132 (2018)

12. Evertse, J.: Linear structures in blockciphers. In: Chaum, D., Price, W.L. (eds.) EUROCRYPT '87. LNCS, vol. 304, pp. 249–266. Springer (1987)
13. Flórez-Gutiérrez, A., Naya-Plasencia, M.: Improving key-recovery in linear attacks: Application to 28-round PRESENT. In: Canteaut, A., Ishai, Y. (eds.) EUROCRYPT 2020. LNCS, vol. 12105, pp. 221–249. Springer (2020)
14. Ji, F., Zhang, W., Zhou, C., Ding, T.: Improved (related-key) differential cryptanalysis on GIFT. IACR Cryptol. ePrint Arch. **2020**, 1242 (2020), https://eprint.iacr.org/2020/1242
15. Leurent, G.: Differential and linear cryptanalysis of ARX with partitioning - application to FEAL and chaskey. IACR Cryptol. ePrint Arch. **2015**, 968 (2015), http://eprint.iacr.org/2015/968
16. Matsui, M.: The first experimental cryptanalysis of the data encryption standard. In: Desmedt, Y. (ed.) CRYPTO '94. LNCS, vol. 839, pp. 1–11. Springer (1994)
17. Matsui, M., Yamagishi, A.: A new method for known plaintext attack of FEAL cipher. In: Rueppel, R.A. (ed.) EUROCRYPT '92. LNCS, vol. 658, pp. 81–91. Springer (1992)
18. O'Donnell, R.: Analysis of Boolean Functions. Cambridge University Press (2014)
19. Shan, J., Hu, L., Song, L., Sun, S., Ma, X.: Related-key differential attack on round reduced RECTANGLE-80. IACR Cryptol. ePrint Arch. **2014**, 986 (2014), http://eprint.iacr.org/2014/986
20. Shpilka, A., Tal, A., lee Volk, B.: On the structure of boolean functions with small spectral norm. Comput. Complex. **26**(1), 229–273 (2017)
21. Sun, L., Wang, W., Wang, M.: Accelerating the search of differential and linear characteristics with the SAT method. IACR Trans. Symmetric Cryptol. **2021**(1), 269–315 (2021)
22. Zhang, W., Bao, Z., Lin, D., Rijmen, V., Yang, B., Verbauwhede, I.: RECTANGLE: a bit-slice lightweight block cipher suitable for multiple platforms. Sci. China Inf. Sci. **58**(12), 1–15 (2015)
23. Zhao, B., Dong, X., Jia, K.: New related-tweakey boomerang and rectangle attacks on Deoxys-BC including BDT effect. IACR Trans. Symmetric Cryptol. **2019**(3), 121–151 (2019)