A generic algorithm for efficient key recovery in differential attacks – and its associated tool

Christina Boura[1], Nicolas David[2], Patrick Derbez[3], Rachelle Heim Boissier[1], and María Naya-Plasencia[2]

[1] Université Paris-Saclay, UVSQ, CNRS, Laboratoire de mathématiques de Versailles, 78000, Versailles, France
{christina.boura,rachelle.heim}@uvsq.fr
[2] Inria, France
{nicolas.david,maria.naya-plasencia}@inria.fr
[3] Univ Rennes, Inria, CNRS, IRISA, France
patrick.derbez@irisa.fr

Abstract. Differential cryptanalysis is an old and powerful attack against block ciphers. While different techniques have been introduced throughout the years to improve the complexity of this attack, the key recovery phase remains a tedious and error-prone procedure. In this work, we propose a new algorithm and its associated tool that permits, given a distinguisher, to output an efficient key guessing strategy. Our tool can be applied to SPN ciphers whose linear layer consists of a bit-permutation and whose key schedule is linear or almost linear. It can be used not only to help cryptanalysts find the best differential attack on a given cipher but also to assist designers in their security analysis. We applied our tool to four targets: RECTANGLE, PRESENT-80, SPEEDY-7-192 and GIFT-64. We extend the previous best attack on RECTANGLE-128 by one round and the previous best differential attack against PRESENT-80 by 2 rounds. We improve a previous key recovery step in an attack against SPEEDY and present more efficient key recovery strategies for RECTANGLE-80 and GIFT. Our tool outputs the results in only a second for most targets.

Keywords: differential cryptanalysis, key recovery, automatic tool, SPEEDY, GIFT, PRESENT, RECTANGLE

1 Introduction

Differential cryptanalysis is an old and powerful technique introduced in 1990 by Biham and Shamir [5]. Soon after its discovery this technique allowed to successfully break some of the most important block ciphers and hash functions of that time, such as FEAL [6], Snefru, Khafre, LOKI [7] and DES [8], to cite just a few. The success of this attack against the cryptosystems of the 70s and 80s, forced the designers of the succeeding ciphers to develop strategies to ensure the resistance of the new designs against this attack. This is how Daemen and Rijmen proposed the *wide-trail strategy* [16] and used it to design the AES or why Vaudenay invented the *decorrelation theory* [34].

Differential attacks are based on the existence of a high-probability differential, that is an input difference that propagates after some rounds to an output difference with a probability much higher than what would be expected for a random permutation. In the case of block ciphers, the existence of one or more such differentials can usually be exploited to recover the secret key through a key recovery procedure. A differential attack against a block cipher can thus be seen as a two-step approach. First, a high-probability differential must be exhibited. Then, this differential is extended to some rounds to permit to recover the secret key or parts of it. The first part of the attack has been, and continues to be, extensively studied and many interesting algorithms and approaches have been proposed. One can cite for example the dynamic programming approach of [21] to find good differential characteristics for `AES-128` in the related-key setting, the MILP-based approach of [32] for bit-oriented block ciphers, the constraint programming (CP)-based method of [30] for all versions of `Rijndael` or the SAT-based method of [31] applicable to many ciphers. It can be seen that this step of differential attacks can be automatized and almost all the approaches that have been proposed lately are based on automatic tools.

All of the above cited algorithms and tools to find good differential distinguishers are exclusively dedicated to this step and there have not been approaches to optimize both steps at the same time. This would be beneficial, as it is not always the best distinguisher that leads to the best attack. There have been efforts in this direction for other families of cryptanalysis, e.g., impossible differential, zero-correlation, integral, meet-in-the-middle, boomerang and rectangle attacks, where tools combining both steps have been proposed [10,38,29,17,22]. However, most of these approaches use heuristics for the key recovery part, by providing for example a rough estimation for the number of involved key bits, and are not guaranteed to lead to the most optimal attack. To build a complete tool for differential cryptanalysis, the key recovery process of these attacks must be well understood. Yet, this step is very technical and error-prone, see for example [33]. Moreover, it is very difficult to come up with an optimal key recovery procedure, as demonstrated by differential attacks published against block ciphers, whose key recovery step was improved by following works. Such examples include attacks against the block ciphers `GIFT` [31] and `RECTANGLE` [37] whose key recovery was later improved in [14] and [13] respectively.

Throughout the years, a series of techniques have been proposed to improve the key recovery step of differential attacks. These techniques and improvements, some of which were introduced for other related attacks but are still applicable to differential cryptanalysis, include the early abort technique [25] to gradually reduce the number of plaintext/ciphertext pairs, the conditional differential cryptanalysis [23], the dynamic key-guessing technique [28,36], other key-guessing strategies [18] or techniques to avoid unnecessary key guesses [2]. Another idea, proposed in [13] for S-box-based designs, was to take advantage of the structure of the S-box in order to reduce the number of necessary key guesses. However, despite the existence of these techniques a global treatment

of the key recovery step is still missing and it remains very difficult to combine the different techniques together to end with a generic efficient procedure.

The lack of a generic and optimal procedure for the key recovery has direct consequences on the design of new ciphers. While most of the newly proposed designs come with claims on the resistance against differential attacks, this is usually done by applying branch-and-bound arguments to determine the highest number of rounds covered by a differential and then, the key recovery added by the designers, if any, is rarely optimal. This can lead then to an erroneous estimation of the security margin and to the choice of a too small number of rounds for the design. This happened recently with the block cipher `SPEEDY-7-192` [24] that provided a wrong estimation of the number of rounds on which key recovery was possible [12].

The main problem one has to solve during the key recovery phase of a differential attack is finding the best key guessing order. This is a difficult combinatorial problem and doing this step by hand is time consuming and there is no guarantee that the followed process is optimal, or even error-free. The existence of a fully automated procedure for this step would be of great help for cryptanalysts but also for designers. Indeed, the existence of such a tool could assist designers in choosing a well suited number of rounds.

Our contributions We propose in this paper the first algorithm for efficiently solving the key recovery problem of a differential attack. Our algorithm captures an efficient key recovery strategy by taking into account many possible optimizations. This algorithm is then transformed into an automated tool that we implemented in `C++`. Our tool takes as input a very simple description of the cipher and a given differential and outputs an efficient order for the key guesses together with the associated time complexity. For each execution of the tool, the user must indicate the number of key recovery rounds to add on both sides. By gradually increasing the number of rounds, the user can get a more precise estimation of the longest valid differential attack that a concrete distinguisher can lead to. This can notably allow designers to choose the best suited number of rounds for their primitive. For this, we focused on SPN ciphers with a bit-permutation as linear layer. We applied our tool to `RECTANGLE`, `PRESENT-80` and `SPEEDY-7-192` in the single-key model and to `GIFT-64` in the related-key model by using differential distinguishers that have been previously given in the literature. Note that we have not verified the validity of the characteristics ourselves as this is out of scope for our work. The correctness of the attacks depends on the validity of these underlying characteristics. Under the assumption that the given characteristic is correct, we improve the best known attack on `RECTANGLE-128` by one round. Further, we slightly improve the best previous key recovery on `SPEEDY`. As for `PRESENT-80`, we extend the best previous differential attack by 2 rounds. Last but not least, we obtain efficient key recovery strategies for `GIFT-64` (in the related-key model) and `RECTANGLE-80`, but do not manage to improve the best previous differential attacks as our tool does not incorporate yet techniques such as the "tree-based graphs" or the "key-absorption" technique used

in [13] or [14]. We however beat all previous key recovery procedures that do not use these techniques, such as the attack of [37] on `RECTANGLE-80` and the attack of [31] on `GIFT-64`. We refer to Table 1 for a summary of our results. Furthermore, the tool is easy to use, as only a very basic description of the cipher is needed. It is also fast as, for most of the attacks, the results were outputted in one second. The code of our tool can be found here:

`https://gitlab.inria.fr/capsule/kyrydi`

The rest of the paper is organized as follows. Section 2 describes the key recovery procedure in differential attacks. Section 3 introduces our modelization of the problem and discusses important optimizations and features we have taken into account, taking as example a toy cipher. Then, Section 4 introduces our new algorithm and describes the related tool. Finally, Section 5 presents our applications to `RECTANGLE`, `PRESENT-80`, `SPEEDY-7-192` and `GIFT-64`.

2 The key recovery problem in differential cryptanalysis

We start this section by introducing differential attacks and the key recovery process. We also describe the type of primitives we will consider in our work, and describe a toy cipher that will be useful to illustrate our algorithm.

2.1 Differential cryptanalysis

As for most cryptanalysis techniques against symmetric primitives, building a differential attack requires two separated, though non-independent, steps. First, an attacker must be able to exhibit a property of the cipher that allows him to distinguish E_K, for any K, from a permutation chosen uniformly at random. In differential cryptanalysis, the distinguisher consists of a pair of differences $(\delta_{in}, \delta_{out}) \in \mathbb{F}_2^n \times \mathbb{F}_2^n$ such that the difference δ_{in} propagates to the difference δ_{out} through a reduced number of rounds with probability strictly higher than 2^{-n}. In a second step, the attacker extends the differential by some rounds, usually to both directions. The appended rounds are called the key recovery rounds. They permit to determine which (partial) keys allow a high number of plaintext pairs and their corresponding ciphertext pairs to follow the differential. An overview of a classical differential attack against a block cipher is depicted in Figure 1. As can

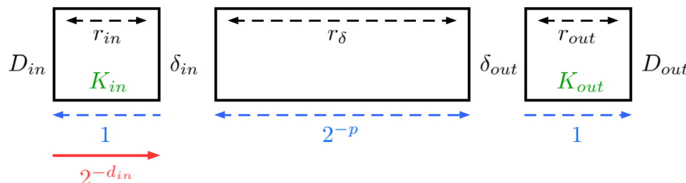

Fig. 1. A differential attack against a block cipher.

be seen in this figure, an attacker first finds a differential $(\delta_{in}, \delta_{out})$ over $r_\delta < \mathtt{r}$ rounds of the block cipher E that has probability $2^{-p} > 2^{-n}$. The difference δ_{in} (resp. δ_{out}) propagates with probability 1 to a difference in a set D_{in}, r_{in} rounds before (resp. D_{out}, r_{out} rounds after). The attack can be symmetrically done in both directions. However, without loss of generality, we focus here on the case where the attacker makes calls to an encryption oracle.

Number of needed plaintext pairs. A classical method to efficiently build the plaintext pairs in differential and other related attacks is to use *structures*. This technique, introduced in [8], permits to reduce the data complexity. A structure is a set of plaintexts that have a fixed value on the inactive bits of D_{in}. Each structure has size $|D_{in}| = 2^{d_{in}}$, and thus allows to build $2^{2d_{in}-1}$ pairs. For the attack to work, the attacker must be able to generate 2^p pairs that have difference δ_{in} after r_{in} rounds. Since each pair in D_{in} satisfies $n - d_{in}$ necessary conditions, we approximate the probability that a pair in D_{in} has difference δ_{in} after r_{in} rounds by $2^{-d_{in}}$. Thus, we need to build $2^{p+d_{in}}$ pairs to get one satisfying the differential with a reasonable probability. This can be done by using $2^s = 2^{p-d_{in}+1}$ structures, corresponding to a data of $\mathcal{D} = 2^{p+1}$.

Key recovery. The goal of this step is to find, for each pair, the possible keys that would partially encrypt the pair to the difference δ_{in} and partially decrypt it to the difference δ_{out}. However, some of the pairs cannot satisfy the differential, independently of the key, e.g., pairs such that their ciphertext difference does not belong to D_{out}. To only work with plaintext pairs whose ciphertext difference belongs to D_{out}, the data of each structure is stored in a hash table indexed by the $n - d_{out}$ inactive bits on the ciphertext. The attacker then looks for collisions on this inactive part. At the end, we can thus build $N = 2^{p+d_{in}-n+d_{out}}$ pairs with a time complexity of $\max(2^{p+1}, 2^{p+d_{in}-n+d_{out}})$ simple operations and a memory complexity of $2^{d_{in}}$ plaintext/ciphertext pairs. For each of these N pairs, we associate the candidate values for the key material involved in the attack and thus generate candidate triplets (P, P', k), where $k \in K_{in} \cup K_{out}$, with K_{in} (resp. K_{out}) being the part of the key to be guessed in the first (resp. last) key recovery rounds. We introduce the parameter C_S to denote the average cost of this step per triplet. At the end of this procedure, if the total number of triplet candidates is smaller than the number of involved key bits, then the attack is considered as successful. Indeed, we can then test the remaining key candidates by completing the missing key bits for a lower cost than that of the exhaustive search. On the other hand, if the number of triplets is higher than the number of involved key bits, we have to consider more data so that the right key will be the one that appears the highest number of times. In this case, one has to take a vector for storing the number of times each candidate key appears, but this makes no difference for the key recovery algorithm we are going to propose. Alternatively, one can also use rounds inside the differential distinguisher to get additional filtering.

As shown in [12], the time ($\mathcal{T}$), data ($\mathcal{D}$), and memory ($\mathcal{M}$) complexities of the attack can be computed in the following way:

$$\mathcal{T} = \left(2^{p+1} + 2^{p+1}\frac{C_S}{C_E} + 2^{p-n+d_{in}+d_{out}}\frac{C_{KR}}{C_E}\right)C_E, \quad \mathcal{D} = 2^{p+1}, \quad \mathcal{M} = 2^{d_{in}},$$

where C_E is the cost of one encryption and C_{KR} is the average cost of the key recovery step per pair. In the time complexity, the first term corresponds to the cost of generating the N pairs, the second term is the complexity of the sieving step and the last term is the complexity of the key recovery step on the remaining $2^{p-n+d_{in}+d_{out}}$ pairs. This last term is the one we want to optimize in this work.

2.2 Efficient key recovery

The key recovery step of a differential attack is traditionally solved by tedious and error-prone procedures. This step is often done by hand and can lead to non-optimal complexities, e.g., [35,31,37]. In this paper, we propose an algorithm that allows to optimize the key recovery step, together with an associated tool. To present our algorithm, we first need to detail further how to perform the key recovery, that is, how to build candidate triplets (P, P', k), with $k \in K_{in} \cup K_{out}$, as described in the previous section. We must also define what it means for an attack to be efficient. We do so in the rest of this section.

To obtain triplets (P, P', k), the attacker must consider each active S-box of the key recovery rounds, and take into account the differential constraints of this S-box. For each pair, the attacker must determine whether this pair *can* respect the differential constraints, and, if yes, under which conditions on the key. Under what we denote by *solving* this S-box, the attacker obtains a list of triplets containing the kept pairs and a partially determined key, with fixed values on the key bits corresponding to this S-box, *i.e.* the key bits added before (resp. after) the S-box on the plaintext (resp. ciphertext) side. The goal of the attack is thus to efficiently reduce as early as possible the number of pairs considered whilst maximizing the number of key bits of $K_{in} \cup K_{out}$ that are fixed.

An attack is optimal when it has the lowest time complexity.[4] What determines the complexity of the key recovery is threefold. First, the order in which each S-box is solved impacts the complexity. For example, if an S-box allows to reduce significantly the number of pairs whilst fixing all key bits, it is often better to solve this S-box early to reduce the overall time complexity. Second, it is possible to *solve* several S-boxes at the same time, which can also help reduce the overall time complexity. Last but not least, S-boxes and sets of S-boxes can be solved in parallel, as described later on. It comes that finding an efficient key recovery procedure consists in choosing an efficient partition of the S-boxes with an associated order on each element of the partition. We will detail these three techniques further in Section 3.

[4]The attack must also have a reasonable memory complexity, but we will detail how we take this into account later.

Note that we can exhibit a natural upper bound for the complexity of the key recovery step. This bound is $\min(2^\kappa, N \cdot 2^{|K_{in} \cup K_{out}|})$, where κ is the bit-size of the secret key. This corresponds to the naïve procedure for which an attacker would guess for each pair (P, P') all the key bits in $K_{in} \cup K_{out}$ to see which pairs and associated key guess lead to the differential. On the other hand, we can also show a lower bound for the key recovery procedure. This lower bound is $N + N \cdot 2^{|K_{in} \cup K_{out}| - d_{in} - d_{out}}$, where $N \cdot 2^{|K_{in} \cup K_{out}| - d_{in} - d_{out}}$ corresponds to the number of expected solutions, *i.e.* the number of pairs and associated partial keys that are obtained at the end of the key recovery procedure. An efficient key recovery algorithm allows us to reach a time complexity for this step as close as possible to this lower bound.

2.3 Considered ciphers

As determining an efficient key recovery procedure is a complex combinatorial problem, it is difficult to solve it once and for all types of symmetric primitives. For this reason, we decided to start with analyzing this problem for the simplest type of SPN constructions, that is, block ciphers such that their round function is composed of an XOR with the round key, a non-linear layer composed of the parallel application of an S-box and a bit-permutation playing the role of the linear layer. This is already a complex case with many applications and will serve as basis for future extensions. Furthermore, we focused on ciphers with linear or almost linear key schedules. Examples of block ciphers that belong to this category are notably `PRESENT` [9], `RECTANGLE` [37] and `GIFT` [1]. Furthermore, we can add to this category ciphers with more complex linear layers as long as this operation is not involved in the key recovery rounds. This is the case of the `SPEEDY` block cipher [24], whose 7-round variant was broken by a differential attack [12] where only 1.5 round was added to the distinguisher and this 1.5 round did not include the complex linear operation, due to the particular construction of the round function.

Toy cipher. We now describe a toy cipher whose design is inspired by `GIFT` [1], and that belongs to the above category. We will use this toy cipher throughout the paper to explain the way our algorithm and its associated tool work. The structure of the cipher together with the description of a differential attack on it can be seen in Figure 2. The block and key length of the toy cipher are 16 and 80 bits respectively. There are 7 rounds, and each round is composed of a bit-wise key addition, an S-box layer, and a permutation layer. The S-box is the one of `GIFT`:

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

and the linear layer P is a bit-permutation given as:

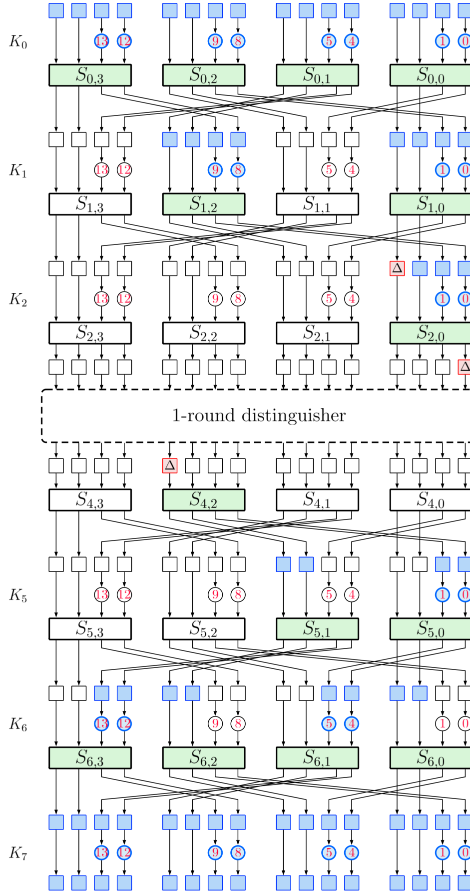

Fig. 2. Our toy cipher and a differential attack against it. S-boxes in green are active. Bits with a difference Δ are active bits, while bits in blue are bits with an unknown difference (0 or 1). Bits in a circle correspond to key bits. Those in blue, are the ones that need to be guessed during the attack.

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

We consider the bit `0`, i.e. the least significant bit, to be on the right for both the state and the round keys. Furthermore, and similarly to `GIFT`, we consider the key addition to be partial and only applied to the two rightmost bits before each S-box. Last, we suppose that there is no key schedule and we treat all round keys as independent. For this toy cipher, we consider the input difference δ_{in} = `0x0001` that propagates after 1 extended round (permutation, addition of K_3, S-box layer, permutation, addition of K_4) to the output difference δ_{out} = `0x0800` as depicted in Figure 2. The difference δ_{in} can be propagated

three rounds backwards with probability one to give $\Delta_{in} = \mathbb{F}_2^{16}$. Similarly, the difference δ_{out} can be propagated three rounds forwards with probability one to give $\Delta_{out} = \mathbb{F}_2^{16}$. The active S-boxes, i.e. the ones which have a non-zero input difference, are colored in green.

Our goal is to determine an efficient algorithm to determine in which order and manner we have to guess these subkey bits, as described in Section 2.2.

3 Modeling the key recovery problem

In this section, we present how we model the key recovery problem. A good modeling plays a crucial role in finding an algorithm to solve the problem in the most efficient way. We also describe three techniques that are used in our tool to obtain an improved key recovery strategy: the S-box sieving technique, the precomputation of partial solutions and the use of parallel computations.

3.1 Our modelization

To model the problem of finding an efficient key recovery for a given differential, we use a directed graph. On this graph, each node represents an active S-box. The graph is then constructed as follows. On the plaintext side, a vertex goes from an S-box S_A at round r to an S-box S_B at round $r+1$ if the input bits of S_B depend on the output bits of S_A. Similarly, for the key recovery rounds on the ciphertext side, a vertex goes from an S-box S_A at round r to an S-box S_B at round $r-1$ if the output bits of S_B depend on the input bits of S_A. An example of such a graph, built to model the toy cipher of Figure 2, is represented in Figure 3. We can see in this graph that the number of vertices going to an internal node is always two, as the four output bits of any S-box affect only two S-boxes in the following round, and as all round keys are considered independent.

Once the graph is built, the key recovery problem corresponds to choosing a specific partition of the nodes together with an associated order. A partition of the nodes divides the original graph into subgraphs and the order indicates in which order the different subgraphs must be treated. A graph representing a key recovery strategy for the toy cipher of Figure 2, can be visualized in Figure 3. Each S-box of the same color belongs to the same subgraph, and the order is represented by a number of the same color. Each partition implies certain subsets of partial solutions and a cost for merging them to obtain the final global solution. The goal is to come up with an algorithm that will output the partition associated with the lowest possible cost.

In the next section, we describe some classical techniques in differential cryptanalysis that impact the modelization of the key recovery problem. Before doing so, we will need some definitions that we introduce below.

Definitions. Let S be an S-box in the key recovery rounds that must satisfy the differential constraint $\nu_{in} \rightarrow \nu_{out}$. We call *solution* to S any tuple

$(x, x', S(x), S(x'))$ such that $x \oplus x' = \nu_{in}$ and $S(x) \oplus S(x') = \nu_{out}$. We denote by *input solutions* (resp. *output solutions*) any pair of values (x, x') (resp. (y, y')) such that $(x, x', S(x), S(x'))$ (resp. $(S^{-1}(y), S^{-1}(y'), y, y')$) is a solution. When it is clear from the context, we sometimes use the term *solutions* to denote input or output solutions.

This definition can be generalized to a subgraph in the following manner. A *solution* to a subgraph is a tuple containing:

- a solution to an S-box in the subgraph that is situated in the first or last key recovery round, i.e. that takes as input a part of the plaintext or the ciphertext.
- a partial solution to an S-box linked to an S-box outside the subgraph. The term partial means that we only consider the bits linked to an external S-box.

For example, a solution to the blue subgraph in Figure 3, is a tuple containing a solution to $S_{0,1}$, a solution to $S_{0,3}$ and a solution to $S_{2,0}$, truncated to its two leftmost bits.

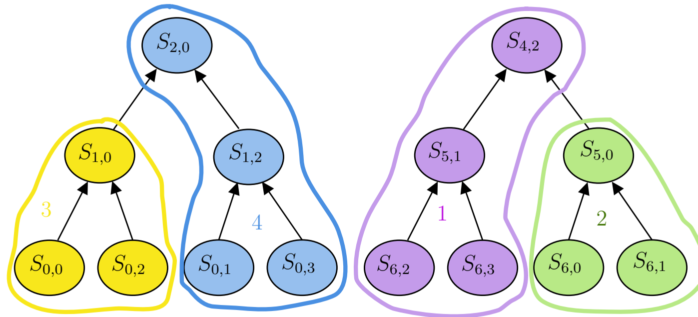

Fig. 3. Graph example and associated partitions of a key recovery strategy for an attack on the toy cipher represented in Figure 2.

3.2 Sieving of the pairs using the differential constraints of the S-boxes

Let N be the number of pairs that the attacker is using for the key recovery. Although N impacts the complexity of the key recovery as this number is multiplied by C_{KR} (i.e. average cost of the key recovery step per pair), the choice of an optimal key recovery strategy does *not* depend on the value of N.[5]

[5] This is assuming N is reasonably big, as we work using average values. We remind the reader that $N = 2^{p+d_{in}-n+d_{out}}$.

To keep C_{KR} as low as possible, it is smart to try to reduce the number of pairs/triplets as quickly as possible. A very classical technique in differential cryptanalysis is to perform a sieving (or filtering) step on the initial N pairs, using the differential constraints of the S-boxes of the first and last round. This sieving step, that we will call *pre-sieving* allows the attacker to further keep only the pairs that *can* satisfy the differential constraints, and to discard those that cannot, as early on as possible. If N' is the number of pairs remaining after the pre-sieving ($N' \leq N$), the attacker performs the rest of the key recovery on those N' pairs.

To perform the pre-sieving, the attacker typically precomputes the *solutions* to each S-box of the external rounds. However, the attacker does not necessarily store all the solutions. Instead, he builds offline a *sieving list* in the following way: on the bits where a key addition is applied, only the difference is stored; on the bits where there is no key addition, the value of each element in the pair is stored. For instance, if the key is XORed to the whole state, only the differences that satisfy the transition are stored. During the online phase, this sieving step consists, for each pair and each S-box, of checking whether the pair is consistent with the sieving list. The filter of the sieving step for that S-box is then computed as the size of the sieving list divided by the total number of possible differences/pairs of values. Note that for this second sieving step we only consider S-boxes that reduce the size of N.

We give an example of this step on the toy cipher of Figure 2. We consider the first S-box of the first round, namely $S_{0,0}$. The differential constraints for this S-box are of the form: $(\texttt{* * * *}) \rightarrow (\texttt{* * 0 0})$ and only the first two input bits to this S-box have key bits XORed to them. Thus, for any solution $(\mathtt{x}_3\mathtt{x}_2\mathtt{x}_1\mathtt{x}_0, \mathtt{x}'_3\mathtt{x}'_2\mathtt{x}'_1\mathtt{x}'_0)$ that satisfies this differential transition, we store the corresponding 6-bit word $(\mathtt{x}_3\mathtt{x}'_3\mathtt{x}_2\mathtt{x}'_2 || \mathtt{x}_1 \oplus \mathtt{x}'_1 \, || \mathtt{x}_0 \oplus \mathtt{x}'_0)$ in a list $\mathtt{L}_{0,0}$. After building $\mathtt{L}_{0,0}$, one can see that for this particular S-box this list has length 36. On the other hand, a 6-bit word can take up to 2^6 values. This provides then a filter of $36/2^6 = 2^{-0.83}$ for this S-box. Using the same reasoning, we obtain the same filter for $S_{0,1}$, as well as a filter $2^{-0.48}$ for $S_{0,3}$ and $S_{0,2}$. On the ciphertext side, the filter for $S_{6,0}$ and $S_{6,2}$ is of $2^{-0.83}$ and it is of $2^{-0.68}$ for the S-boxes $S_{6,1}$ and $S_{6,3}$. In conclusion for this example, the pre-sieving step allows to work with $N' = 2^{-5.63} \cdot N$ pairs instead of N.

Compensation of pre-sieving. One important point is that the pre-sieving described in the above paragraph and performed in the initial steps must be taken into account later on in the key recovery procedure. Take for instance the example of the S-box $S_{0,0}$ of the toy cipher. This S-box has 2^6 solutions, and each solution fixes a value on the two leftmost bits, and a difference on the two rightmost bits (because of the key addition). On the other hand, an online pair can take 2^4 values on the two leftmost bits, and 2^2 differences on the two rightmost bits. An online pair thus has probability $2^6/2^6 = 1$ to match a solution. Since each solution is a pair of values, a match on a solution fixes the key bits. On the other hand, with the pre-sieving, we initially filter out $2^{-0.83}$ of the pairs, but when determining the key bit values we will get $2^{0.83}$ possibilities per pair

on average, thus canceling locally out the gain of the pre-sieving. This example shows that sieving on a single S-box and solving this S-box immediately after is not a good strategy. Nevertheless, there is often a significant advantage when the pre-sieving is applied to all the S-boxes very early on: as the compensation is gradual, the time complexity is overly reduced. Indeed, the attacker will still work with less pairs during the different steps than if no pre-sieving was applied. For example, if we apply the pre-sieving step to all S-boxes but $S_{0,0}$ in the toy cipher, we start by working with $2^{0.83} \cdot N' = 2^{-4.8} \cdot N$ pairs, the cost of solving $S_{0,0}$ is $2^{-4.8} \cdot N$, and we keep only $2^{-4.8} \cdot N$ pairs after solving $S_{0,0}$.

Generalization to any active S-box. Note that it is also possible to sieve later by using S-boxes of other key recovery rounds, as long as we have reached a point in the key recovery procedure where the input of these S-boxes is fixed. This sieving must not be confused with the pre-sieving that only applies to the external key recovery rounds.

3.3 Precomputing partial solutions

Besides the pre-sieving method described above, an attacker can also apply another technique to reduce the time complexity. The idea of this technique is to precompute the partial solutions to some subgraph(s). However, such precomputations can impact the memory complexity of the attack and increase the offline time of the attack. Thus, the optimal key recovery strategy depends on how much memory and offline time are allowed. For this, we introduce a parameter M which corresponds to the maximum memory complexity (aside from storing the pairs) and the maximum allowed precomputation time complexity. The result obtained will directly depend on this parameter. We give an example of such a precomputation phase on the toy cipher of Figure 2.

Precomputing the solutions to $S_{0,0}$, $S_{0,2}$ and $S_{1,0}$. We compute the number of solutions to a subgraph containing as nodes the S-boxes $S_{0,0}$, $S_{0,2}$ and $S_{1,0}$. The differential constraints of $S_{1,0}$ are as follows: `(* * * *)` $\rightarrow$ `(1 * 0 0)`. There are $2^5 (= 2^{2\times4-3})$ output solutions to this S-box, and thus 2^5 input solutions. Looking at the input *before the key addition*, each input solution determines the value of the pair on the two leftmost input bits and the difference on the two rightmost ones, i.e. the ones concerned by the key addition.

We now consider a fixed input solution to $S_{1,0}$, and look at $S_{0,0}$. Two output bits of $S_{0,0}$ are fixed by this solution. However, the two rightmost bits, which have the only constraint of having a zero difference, can take up to 2^2 values. This means that for a solution to $S_{1,0}$, there are 2^2 possibilities at the output of $S_{0,0}$, and thus 2^2 solutions at its input. For each solution, the information at the input of $S_{0,0}$ is fixed in value on the two leftmost bits, and in difference on the two rightmost bits because of the key addition.

Still using a fixed solution to $S_{1,0}$, we investigate $S_{0,2}$. The two rightmost output bits have their difference fixed by the solution to $S_{1,0}$, whilst the two leftmost output bits have a 0-difference by propagation of the differential. There

are therefore 2^4 possibilities for the output pairs for this S-box, and thus 2^4 input solutions. Note that each solution fixes the bits added at the input of $S_{1,0}$. As for the previous S-box, each solution completely fixes the value of the two leftmost input bits (where there is no key addition), and the difference on the other two.

In total we have $2^5 \cdot 2^2 \cdot 2^4 = 2^{11}$ solutions at the input of $S_{0,0}$ and $S_{0,2}$. Each solution is a fixed pair of values after the first key addition. Thus, when considering an online pair, for this pair to be kept whilst solving $S_{0,0}$, $S_{0,2}$ and $S_{1,0}$, it must match a solution in value on the two leftmost bits of each S-box (which can take $2^{2 \cdot 2 \cdot 2} = 2^8$ values), and in difference on the two rightmost bits (which can take $2^{2 \cdot 2} = 2^4$ differences). The average probability of a match is thus $2^{11}/2^{12} = 2^{-1}$. Since a match fixes the solution, and thus the value after key addition, each match fixes the key bits at the input of $S_{0,0}$, $S_{0,2}$ and $S_{1,0}$.

This represents a filter of 2^{-1} on the online pairs. By letting N'' to be the number of solutions before this subgraph, the number of pairs remaining after solving it is $2^{-1} \cdot N''$. If we have already applied a pre-sieving on $S_{0,0}$ and $S_{0,2}$, then this sieving must be compensated. Thus, the number of remaining pairs is $2^{1.3} \cdot 2^{-1} \cdot N'' = 2^{0.3} \cdot N''$. Further, each of the $N'' \cdot 2^{0.3}$ new triplets has a fixed value on the key bits at the input of $S_{0,0}$, $S_{0,2}$ and $S_{1,0}$. *Testing* each of the N'' pairs, that is, checking whether each pair belongs to the pre-computed table of solutions, can be done with a time complexity of $N'' \cdot 2^{0.3}$ using hash tables.[6] In this example, the precomputation is not too costly as it requires 2^{11} applications of the S-box, and the storage of 2^{11} pairs of length $4 \cdot 2 \cdot 2 + 2 \cdot 3 = 22$ bits (the pairs of input values of $S_{0,0}$ and $S_{0,1}$, as well as all the key bits).

We can compare the use of the precomputation technique here to the use of a sequential solving. In this example, we suppose that among the N'' pairs, a pre-sieving on $S_{0,0}$ and $S_{0,2}$ has already been applied. If we want to advance sequentially further in the graph, we need to determine the key bits of $S_{0,0}$ and $S_{0,2}$ before solving $S_{1,0}$. Determining those key bits costs $2^{1.3}N''$, whilst the 2^{-1} filter from $S_{1,0}$ is only applied in a second step. On the other hand, if we use the precomputed list of solutions, we can directly compute the $N'' \cdot 2^{1.3} \cdot 2^{-1} = N'' \cdot 2^{0.3}$ solutions for a cost of $N'' \cdot 2^{0.3} < N'' \cdot 2^{1.3}$.

This example shows how the amount of memory complexity allowed can impact the complexity of the key recovery step. We invite the reader to use our tool on the toy cipher using different memory/offline time complexity constraints and to compare the results.

3.4 Computing in parallel

An attacker can perform certain steps in parallel to improve the complexity of the attack. We give an example using the first step of the key recovery on the toy cipher. We consider the problem of solving in parallel $S_{6,2}$, $S_{6,3}$, $S_{5,1}$ and $S_{4,2}$ on one hand, and $S_{6,0}$, $S_{6,1}$ and $S_{5,0}$ on the other hand. This corresponds

[6] In fact, once divided by the cost of the toy cipher, the time complexity is smaller than $N'' \cdot 2^{0.3}$.

to the purple and green subgraphs in Figure 3 respectively. Further, we consider that we have started our key recovery procedure by applying a pre-sieving on all S-boxes of the first and last round, leaving us with $N' = 2^{-5.63} \cdot N$ pairs.

The purple subgraph. We begin by computing the remaining number of pairs after solving the S-boxes of the purple group. There are 2^6 solutions to $S_{6,2}$. Taking into account the sieving step, the number of triplets remaining after solving this S-box is thus $(2^{6+0.83}/2^6) \cdot N' = 2^{0.83} \cdot N' = 2^{-4.8} \cdot N$, and each triplet has a fixed value on the bits added at the input of $S_{6,2}$. Using the same reasoning, it follows that the number of pairs remaining after solving $S_{6,3}$ and $S_{5,1}$ is $2^{0.83+0.68} \cdot N'' = 2^{-4.12} \cdot N$. There are 2^4 solutions to $S_{4,2}$. After solving $S_{5,1}$, only 2 bits are fixed, and there are 2^4 possibilities for the remaining 2 bits. The filter obtained by solving $S_{4,2}$ is thus $(2^{4+4}/2^8) \cdot N = 1$. The number of solutions after solving the purple is thus $2^{-4.12} \cdot N$. This step can be done at a cost $2^{-4.12} \cdot N$ using precomputations using a memory 2^{16}.

The green subgraph. Using a similar reasoning, the number of pairs remaining after solving $S_{6,0}$ and $S_{6,1}$ is $2^{-4.12}N$. This fixes the 4 input bits to $S_{5,0}$, and thus the number of solutions remaining after solving $S_{5,0}$ is also $2^{-4.12} \cdot N$. Solving these S-boxes can also be done at a cost of $2^{-4.12} \cdot N$ using precomputations with a memory of 2^{12}.

Merging the two subgraphs. We now merge the solutions of the green group with those of the purple group. Solving the green group has fixed two output bits of $S_{5,0}$, whilst solving the purple group has fixed the same two input bits of $S_{4,2}$, but *after* the key addition. Thus, the probability that a solution of the purple group is also a solution to the green group is the probability that the difference on these two bits is equal. This probability is thus 2^{-2}. Further, each newly computed solution fixes the key bits. Thus, after merging the two subgraphs, the number of pairs remaining is $2^{-2+2 \cdot 0.83+2 \cdot 0.68} \cdot N' = 2^{-4.61} \cdot N$. This can be done at a cost of $2 \cdot 2^{-4.12} \cdot N$ (the size of each subgraph), by for example starting by solving the purple subgraph using pre-computed partial solutions to that subgraph and then merging the result with the pre-computed partial solutions of the green subgraph.

Using a sequential attack would have been less efficient here. For example, if one would have first solved $S_{6,0}$, $S_{6,1}$, $S_{6,2}$, $S_{6,3}$, $S_{5,0}$ and $S_{5,1}$ before solving $S_{4,2}$, then the cost of solving $S_{4,2}$ would have been about $2^{2 \cdot 0.83+2 \cdot 0.68} \cdot N'' = 2^{3.02} \cdot N' = 2^{-2.61} \cdot N$ (the number of solutions remaining after solving those S-boxes), which is strictly greater than the cost using parallelization.

4 Algorithm and its associated tool

We provide in this section a description of our tool which captures an efficient key recovery strategy. We begin by presenting the algorithm, on which the tool is based, in a high-level manner. Then, we describe how we incorporate the

techniques of Section 3. Finally, we give the parameters of the tool and discuss the limitations of our algorithm.

4.1 High-level description of our algorithm

Using the modelization described in Section 3.1 it is quite straightforward to describe an algorithm ready to be implemented. Given a distinguisher and a number of key recovery rounds, we start by identifying all active S-boxes involved in the key recovery. As they correspond to the nodes of the graph, we will sometimes denote them by *nodes* in this section. For each S-box, we compute its solutions, and compute the resulting filter as described in Section 3.2.

Then, our algorithm considers what we call *strategies*. Given a subgraph X, a strategy $\mathcal{S}_X$ for the subgraph X is a divide-and-conquer procedure that allows to enumerate all the possible values that the S-boxes of X can take under the differential constraints imposed by the distinguisher. The most important parameters of a strategy $\mathcal{S}_X$ are

- its *number of solutions*, which in fact does not depend on the strategy itself but only on the subgraph X;
- its *online time complexity*, that is, the average time complexity it takes to check whether an online pair is consistent with the differential constraints of the subgraph .

A strategy can be further refined with extra information such as the memory complexity and the offline time complexity needed to attain the online time complexity.

The output of the tool is an efficient *graph strategy*, *i.e.* a strategy for the whole graph. This strategy corresponds to an efficient algorithm associating to each online pair the key values that partially encrypt/decrypt this pair to the distinguisher differences. It is built using *basic strategies*, that is strategies for a single S-box. Given those basic strategies, and by incorporating the techniques of Section 3 as described in Section 4.2, the tool finds the most efficient order in which to combine all basic subgraphs, aiming to minimize the complexity of the resulting strategy. The algorithm is schematically described in Algorithm 1.

Comparing strategies. To look for the best strategy, our tool must be able to compare them. It only makes sense to compare strategies for a common subgraph, at least initially. Considering a subgraph X, and two strategies $\mathcal{S}^1_X$ and $\mathcal{S}^2_X$ for X, these strategies necessarily possess the same number of solutions. To compare them, we thus compare their second parameter, that is, their time complexity. Of course, the best strategy has the lowest complexity. When taking into account the memory complexity, if two strategies have the same online time complexity, then the strategy with the lowest memory complexity is considered to be the best.

Merging strategies. To build a global strategy from the basic ones, we must precisely define what is the complexity of the strategy $\mathcal{S}_{X\cup Y}$ obtained by merging together two strategies $\mathcal{S}_X$ and $\mathcal{S}_Y$. As the number of solutions of a subgraph strategy depends only on the subgraph, the number of solutions of $\mathcal{S}_{X\cup Y}$ only depends on $X \cup Y$. We evaluate this number of solutions by summing (assuming a logarithmic representation) the number of solutions of each node from $X \cup Y$, and by subtracting the number of bit-relations between these nodes, including relations obtained from the key schedule. Note that we restricted ourselves to linear key schedules and thus computing the relations between the round key bits involved in a strategy is as simple as computing the dimension of some vector space. The time and memory complexity are easily computed as follows:

- Time online/offline: $T(\mathcal{S}_{X\cup Y}) \approx \max(T(\mathcal{S}_X), T(\mathcal{S}_Y), Sol(\mathcal{S}_{X\cup Y}))$
- Memory: $M(\mathcal{S}_{X\cup Y}) \approx \max(M(\mathcal{S}_X), M(\mathcal{S}_Y), \min(Sol(\mathcal{S}_X), Sol(\mathcal{S}_Y)))$

This is because the actual procedure behind the combination of two strategies is to first run the one with the smallest number of solutions, store these solutions in a hash table indexed by the bit relations between the two subgraphs, run the second one, and look for matches into the hash table. Note that the *real* complexities of a strategy have to be multiplied by the number of S-boxes it enumerates, and thus the real formula we use is $T(\mathcal{S}_{X\cup Y}) \times |X \cup Y| \approx \max(T(\mathcal{S}_X) \times |X|, T(\mathcal{S}_Y) \times |Y|, Sol(\mathcal{S}_{X\cup Y}) \times |X \cup Y|)$. This formula is quite accurate since most often, optimal strategies are built from disjoint sub-strategies.

It is important at this point to notice that the online time complexity of a strategy resulting from a merge only depends on the time complexities of the two merging strategies. It comes that an optimal strategy can always be obtained by merging two optimal strategies: if a strategy for X (resp. Y) has the lowest online time complexity possible, then necessarily the maximum of these two values is also the lowest possible. Thus, the strategy for $X \cup Y$ obtained by merging these two strategies is necessarily the most efficient. This justifies using a bottom-up approach, merging first the strategies with the smallest time complexity to reach a graph strategy with a minimal time complexity. We can rely on dynamic programming-related techniques to ensure that, for any subgraph X, we only keep one optimal strategy to enumerate it. Furthermore, note that we can extend the cases in which two strategies can be compared. Indeed, if X contains Y and if the number of solutions of X is not higher than the number of solutions of Y, then we can freely replace any strategy for Y by a strategy for X as long as it has a better or similar time complexity.

While this reduces a lot the number of merges to perform, there are still, assuming n nodes, $\sum_{i=2}^{n} \binom{n}{i} \sum_{j=1}^{i-1} \binom{i}{j}$ merge combinations to try for, which is intractable. Thus, to limit the number of allowed merges, our tool only considers merges for which there is at least one bit relation between the two subgraphs. Graphically, this corresponds to the existence of at least one vertex between two nodes of the subgraph, or a common S-box. For instance, looking at Figure 3, our tool does not allow the merge of $S_{1,0}$ together with $S_{0,1}$ since these two S-boxes are not related.

Algorithm 1 Key recovery algorithm

```
L_done ← ∅
L_current ← basic strategy for each node of the graph
while L_current ≠ ∅ do
    Let S be the strategy from L_current with the smallest time complexity
    for every S' in L_done allowed to be merged with S do
        Let S'' be the merge between S and S'
        if no strategy from L_done nor L_current is similar or better than S'' then
            Update S'' for free (by merging with basic strategies)
            Remove from both L_done and L_current all strategies worst than S''
            Add S'' to L_current
        end if
    end for
    Remove S from L_current and add it to L_done
end while
```

4.2 Taking into account the techniques of Section 3

It can be easily seen that the algorithm as described above already outputs strategies taking parallel computations into account. However, other techniques such as pre-sieving and precomputations require some adaptation.

Sieving. To take into account the sieving, the tool uses in practice a modelization that is slightly more complex than the one described so far where each S-box was seen as a single node. In this modelization, each S-box is fully described by *two* nodes. The first node corresponds to the sieving step on this S-box (it contains the sieving list defined in Section 3.2), whilst the second node contains the full S-box (i.e. all the solutions). Thus, a subgraph containing the sieving node but not the second node describes a step in the key recovery in which the attacker has performed a sieving on this S-box but does not want to enumerate its actual values to avoid increasing the number of solutions at this step of the key recovery. Note that to merge the second node to a subgraph, we require the sieving node to belong to this subgraph.

Precomputations. Our algorithm distinguishes between online and offline strategies. Online strategies depend on the data (their online time complexity is expressed as a coefficient that multiplies N), while offline ones do not (in fact, they do not possess an online time complexity). This attribute has one main effect regarding the merge of two strategies and slightly modifies the formula used to compute the inner complexities of the resulting one. Let $\mathcal{S}_X$ and $\mathcal{S}_Y$ be respectively an online and an offline strategy. The complexities of the strategy obtained by merging both of them, i.e. $\mathcal{S}_{X \cup Y}$, are defined as follows:

- Time online: $T_{on}(\mathcal{S}_{X\cup Y}) \approx \max(T_{on}(\mathcal{S}_X), Sol(\mathcal{S}_{X\cup Y}))$
- Time offline: $T_{off}(\mathcal{S}_{X\cup Y}) \approx \max(T_{off}(\mathcal{S}_X), T_{off}(\mathcal{S}_Y))$
- Memory: $M(\mathcal{S}_{X\cup Y}) \approx \max(M(\mathcal{S}_X), M(\mathcal{S}_Y), Sol(\mathcal{S}_Y))$

In other words, in this particular scenario, the offline strategy is selected to be the one stored into the hash table while the solutions of the online strategy will be generated on the fly.

The user can set an upper bound on the time spent offline and this bound can be higher than the bound for the online time (a feature that could be useful depending on the attack scenario the key recovery is a part of).

Restricting merges. To decrease the number of merges to consider, we add an additional restriction on the allowed merges. Our idea is to force that a non-filtering node can be merged with an online strategy if and only if either the node is obtained for free (i.e. it does not increase the current number of solutions of the strategy) or it is computed by partially encrypting/decrypting the data. For instance, if $S_{1,2}$ was not filtering then we would not allow to merge $S_{0,1}$ with it, and only a strategy enumerating both $S_{0,1}$ and $S_{0,3}$ would be allowed to do so.

4.3 Parameters and limitations

To provide the best key recovery strategy, our tool must take as input a description of the cipher and the distinguisher.

The block cipher. To describe the block cipher, the user must provide the specification of the S-box $\mathtt{S}$, the bit-permutation $\mathtt{P}$, the number of rounds n_R and the key schedule. Note that for now, only linear key schedules are allowed. For non-linear key schedules, the user can either replace $\mathtt{S}(k)$ by a new variable (i.e. omitting the relation between k and $\mathtt{S}(k)$) or replace it by $L(k)$ for a random invertible linear function L. These two options give the user an upper and a lower bound respectively on the complexity of the key recovery.

The distinguisher. To specify the distinguisher, the user must provide the input difference δ_{in} and the output difference δ_{out} of the distinguisher, the number of key recovery rounds nr_p to prepend to the differential (plaintext side) and the number of key recovery rounds nr_c to append to the differential (ciphertext side). Note that for our tool, a round corresponds to the XOR of the round key to the state, the S-box layer and then the permutation layer, always taken in this exact order. Last but not least, if the attack is in the related-key model, the attacker must specify the difference on each round key.

An option to control the propagation. The user can optionally specify some differential constraints on the last S-box transition preceding the distinguisher. This can be done by specifying a difference on some or all bits at the input of the last S-box application before entering the distinguisher. Similarly, the user can optionally specify a difference on some or all bits at the output of the first S-box after the distinguisher. This is a way for the attacker to be able to control part of the propagation if they wish to do so.

The above parameters are given as input to the program in the form of a `.txt` file containing a very simple description of the above parameters. We give an example of such a `.txt` file for our toy cipher in Annex A.

Constraints on the precomputation. Finally, the program allows for optional constraints on the memory complexity and on the offline time complexity in a precomputation phase. This is done thanks to options specified during the execution, as described in the `README.md` file that can be found in the git repository of our tool.

An option to accelerate the search. Our tool can be used either to find the most efficient key recovery strategy using the maximum amount of memory and or precomputation specified by the user or simply to find a key recovery strategy that has complexity under a given security parameter, to verify whether there is an attack that exists. While the tool is in general very efficient, in some cases, when several key recovery rounds are appended, the running time could be too high. In this case, using a particular option specified in the above mentioned `README.md` file, permits the user to quickly check whether an attack exists, without asking the tool to find the most efficient key recovery strategy.

Minimizing the memory complexity. We have described up to now only parallel merges, in which the memory depends on the smallest number of solutions between the two strategies involved in the combination. However, let X, Y and Z be 3 subgraphs and assume we have access to a strategy for each of them. If the optimal strategy, in terms of time complexity, to enumerate $X \cup Y \cup Z$ requires to merge $\mathcal{S}_{X \cup Y}$ and $\mathcal{S}_{X \cup Z}$, it might be interesting to construct both of them with *sequential* merges to minimize the memory complexity. In other words, we would like to first enumerate all the possible values for X and then, for each of them, compute the possible values for both Y and Z in parallel and after that look for matches. This might be very efficient to reduce the memory complexity of the final strategy. However, doing so makes the search for the best strategy much harder, especially since the order in which merges were performed inside a strategy does matter for future merges, forbidding to use an approach based on dynamic programming. Thus we do not optimize on the memory complexity and only keep at most one strategy per subgraph. Still, among the strategies constructed during the search procedure, we will always keep the ones minimizing the memory complexity.

Parallel matching algorithms. In several key recovery procedure, like in [14] and in [12], a complex algorithm for efficiently matching partial solutions that were computed in parallel is used: the parallel matching [26,15] permits to find with an efficient complexity the total number of solutions with respect to non-linear relations. Actually, this technique can be seen as a merge of three strategies in one step while we only described how to merge two strategies. We did not take this technique into account for two reasons. First, it would make the search space much bigger since the number of possible combinations increases exponentially.

Second, when the third strategy is composed of two sub-strategies (which is the case for all of them but the basic ones enumerating a single S-box), we experimentally found that merging them 2 by 2 always led to the same overall time complexity (though usually to higher memory needs). Additionally, in the remaining case of a third strategy enumerating a single S-box or its associated filter, the gain over our merging process is marginal, due to both the small number of solutions enumerated and the small filter that basic strategies offer. Still, it would be a nice future work to identify precisely the cases in which we should switch to the parallel matching technique instead of the simple merge procedure.

Accuracy of the complexity. In order to evaluate the complexity of a given strategy, we need to be able to compute the number of solutions outputted by each sub-strategy it is composed of. This is a difficult problem in general as it corresponds to the evaluation of the number of solutions of a complex polynomial system over $\mathbb{F}_2$. Thus we use the common assumption that a system of n linearly independent polynomial equations involving m variables on $\mathbb{F}_2$ has approximately 2^{m-n} solutions. Taking into account that the non-linear part of the polynomial equations all come from the S-boxes involved in the target cipher, that all the S-boxes are permutations and that the number of pairs on which is performed the key recovery is most often much higher than the size of the S-boxes, this assumption should hold in most scenario. Actually the same one was for instance used in [11] in which the authors verified its accuracy on some examples involving the `AES` S-box.

We provide a comparison with previous works in Annex B.

5 Applications

We describe the application of our tool to four block ciphers: `RECTANGLE`, `GIFT-64`, `PRESENT-80` and `SPEEDY-7-192`. The three first ones have a bit-permutation as linear layer, whilst `SPEEDY` has two types of linear operations: a bit-permutation and a more complex one, represented by a matrix multiplication. However, this second operation is not involved in the key recovery rounds of the attack we are going to analyze. Thus, our tool can still be applied to it. `SPEEDY` and `GIFT`, have a linear key schedule, whilst `PRESENT` and `RECTANGLE` do not. Since our tool does not handle non-linear schedules, we apply it to `PRESENT` by replacing the S-box involved in the key schedule by a randomly generated matrix. Then, we verified by hand that the proposed attack still worked with the original key schedule of `PRESENT` as there is only a single S-box application per round. For `RECTANGLE`, as the key schedule is more complex, we consider that all round keys are independent.

The four applications we considered demonstrate the applicability of our tool to primitives with different characteristics. Once a differential distinguisher is provided, it is very easy to determine the highest number of rounds that can

be attacked by trying different configurations for the key recovery rounds. This allowed us to mount a 19-round differential attack on `RECTANGLE-128` with a differential distinguisher provided by the designers [37], while designers as well as cryptanalysts who improved this same attack [13] missed this possible extension and stopped at 18 rounds. While it has been shown in [4] that the underlying differential distinguisher is only valid for half of the keys, our application still demonstrates the facility of our tool to check for the existence of attacks on any number of rounds once a differential distinguisher is given. It will be easily applied if better valid distinguishers are provided in the future for this cipher. For `PRESENT`, differential attacks are not the best existing attacks. On the other hand, the differential attack of Wang [35] on 16-round `PRESENT-80` has been known for more than 15 years now and has not been significantly improved since. We show here that using the same differential distinguisher with an improved efficient key recovery strategy, one can mount an attack on two extra rounds. This result was very easy to find by simply applying our tool for all relevant configurations of the key recovery rounds. The particular case of `GIFT` is very interesting, and a comparison with previous attacks will be described in detail. Finally, for `SPEEDY` we slightly improve the complexity of the key-recovery part of the attack in [12] by launching the tool on this cipher and analyzing the produced key recovery graph. This only took one second to the tool, while the authors of [12] found the key recovery strategy by hand through a very tedious procedure.

For each application, we briefly provide the specification of the cipher as well as the previous best known differential attack on it. In each case, we keep the same differential distinguisher as the one given in the original cryptanalysis papers. Indeed, our goal is not to improve the differential search step, but to ameliorate the key recovery procedure (or at least to find an equivalent one effortlessly). We will present our different applications. The graph produced for each attack can be visualized on the `Git` repository we provide. The complexity of the key recovery step for each analyzed cipher is summarized in Table 1.

5.1 Validity and Experiments

We applied our tool to differential distinguishers that have been described in the literature for the different ciphers to show how much our tool can improve previous key recoveries. We also verified by hand the key recovery of the new attacks given in Table 1. More precisely, we checked that the strategies outputted by our tool are coherent (e.g. no missing nodes, realistic costs, ...) and then we computed by hand the number of solutions at each step of the process as we did in Section 3 for the toy example. For concision reasons, we do not include these descriptions in this work.

We also emphasize that we did not check whether the theoretical probabilities of the distinguishers match the experimental ones as this was out of scope for the current work. Therefore, our attacks (and actually all differential attacks in the literature) can be considered as valid only under the assumption that the underlying distinguishers are valid themselves. In particular, Beyne and Rijmen showed that the differential characteristic used in the attack against

Cipher	# Rounds	N	$N \times C_{KR}$	$\mathcal{T}$	Ref.
RECTANGLE-80	18	$2^{50.83}$	$N \cdot 2^{27.84}$	$2^{78.67}$	[37]
RECTANGLE-80	18	$2^{50.83}$	$N \cdot 2^{13.27}$	2^{64}	[13]
RECTANGLE-80	18	$2^{50.83}$	$N \cdot 2^{19}$	$2^{69.83}$	Sec. 5.2
RECTANGLE-128	19	$2^{78.83}$	$N \cdot 2^{43}$	$2^{121.83}$	Sec. 5.2
PRESENT-80	16	2^{28}	N	2^{65}	[35]
PRESENT-80	18	2^{58}	$N \cdot 2$	2^{59}	Sec. 5.3
SPEEDY-7-192*	7	$2^{190.32}$	$N \cdot 2^{2.83}$	$2^{187.84}$	[12]
SPEEDY-7-192*	7	$2^{190.32}$	N	$2^{187.38}$	Sec. 5.5
GIFT-64†	26	$2^{115.96}$	$N \cdot 2^{7.27}$	$2^{123.23}$	[31]
GIFT-64†	26	$2^{115.96}$	N	$2^{115.96}$	Sec. 5.4

Table 1. Summary of the previous best attacks, and of how our results impact them, on RECTANGLE, PRESENT-80, SPEEDY-7-192 and GIFT-64. C_{KR} corresponds to the cost of the key recovery, while $\mathcal{T}$ represents the total time complexity of the attack. *The complexities in the attack on SPEEDY are given with the cost of one encryption as unit, that is estimated to 2^7. We use the same unit here. †These attacks are in the related-key model.

RECTANGLE-80 only holds for at most half of the keys [4]. The recent work of Peyrin and Tan [27] also suggests that the differential characteristics used in other lightweight ciphers should be carefully checked.

5.2 RECTANGLE

RECTANGLE is a block cipher designed by Zhang, Bao, Lin, Rijmen, Yang and Verbauwhede in 2015 [37]. It is based on an SPN construction and uses a state of 64 bits. The state can be seen as a concatenation of 16 nibbles. The round function consists of an XOR with the round key, the application of a 4-bit S-box in parallel to each nibble of the state and a bit-permutation that plays the role of the linear layer. The S-box of the cipher, its DDT as well as the linear layer are provided in Annex C. The round function is iterated 25 times. The key of RECTANGLE can be 80-bit or 128-bit long. We do not describe the key schedule here as we consider all subkeys to be independent.

The differential attack of [37] The designers of RECTANGLE described a differential attack on 18-round RECTANGLE for both versions of the cipher. This attack is based on a 14-round differential distinguisher of probability $2^{-62.83}$. Two key recovery rounds are added on both sides (see Figure 2 for the propagation of the differences). This attack has a data complexity of 2^{64}, a time complexity of $2^{78.67}$

and a memory complexity of 2^{72} key counters. An improvement of the time complexity of the original attack was however later given in [13]. In this last article, the authors used several advanced techniques, such as the "tree-based graphs" and the key-absorption technique, which allowed them to improve the time complexity to 2^{64} in a non-automated, very technical and hard to verify way. While we would have liked to reach the same complexity, this example shows that our tool can only be beaten by using very sophisticated techniques and permit to researchers to focus on them. Note however, that Beyne and Rijmen showed in [4] that the distinguisher on which both the attacks of [37] and [13] are based is valid for at most half of the keys only.

S-box	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ΔI_{r-3}	****	****	****	****	****	****	****	....	****	****	*11*	0000	****	****	....	****
ΔO_{r-3}	**0*	***.	*...	...*	.***	*1**	*.*.	....	...*	..**	..1.	.0..	0*..	*...	....	.*.0
ΔI_{r-2}	****	....	....	****	****	****	....	....	****	*11*	....	....	....	....	....	0000
ΔO_{r-2}	*...	....	....	...*	.1*.	*...	....	....	...*	..1.	....	....	....	....	....	.0..
ΔI_{r-1}	....	....	....	*0**	....	....	....	....	*11*	....	....	....	....	....	....	....
ΔO_{r-1}	....	....	....	.11.	....	....	....	....	..1.	....	....	....	....	....	....	....
ΔI_r	....	....	..1.	....	....	....	....	.11.	....	....	....	....	....	....	....	....
14-round distinguisher																
ΔI_{r+14}	....	....	....	.1..	....	....	....	....	....	....	....	....	....	....	..1.	....
ΔO_{r+14}	....	....	....	**11	....	....	....	....	....	....	....	....	....	....	****	....
ΔI_{r+15}	....	*...	.*1.	...1	....	....	*...	.*..	....	....	....	....	....	..*.	...*	....
ΔO_{r+15}	....	****	****	**1*	....	....	****	****	....	....	....	....	....	****	****	....
ΔI_{r+16}	*.*.	****	.*1*	...*	..*.	*.**	****	.*.*	....	*...	*.*.	.*..	..*.	..**	...*	....
ΔO_{r+16}	****	****	****	****	****	****	****	****	....	****	****	****	****	****	****	....

Table 2. Differential attack on $(14+x)$-round RECTANGLE based on the 14-round distinguisher from [37], where $x \in 4, 5$. ΔI_r and ΔO_r are respectively the states before and after the S-box layer of round r. The symbol '.' stands for a non-active bit, '1' for an active bit, and '*' stands for a bit with an unknown difference. Finally, '0' corresponds to a bit with 0 difference but whose value needs to be known for the key-recovery.

Application of our tool We applied our tool to RECTANGLE by adding 2 key recovery rounds in both directions as done in the original attack of [37]. As the probability of the distinguisher is $2^{-p} = 2^{-62.83}$, the data complexity of the attack is $\mathcal{D} = 2^{p+1} = 2^{63.83}$. The number of pairs can be computed based on the related spaces D_{in} and D_{out} (see Section 2). For the 18-round attack, $d_{in} = 24$ and $d_{out} = 28$. Thus, by using the encryption oracle, we can form $2^{p+d_{in}} = 2^{86.83}$ pairs, and among them $N = 2^{p+d_{in}-(n-d_{out})} = 2^{50.83}$ should survive the filtering by the ciphertext difference. Our tool outputted a complexity for the key recovery phase equal to $N \cdot 2^{19} = 2^{69.83}$. This is the dominant term in the complexity of our attack, yet, it is is much lower than the complexity of the key recovery in the original attack of [37]. As the execution of the tool for the above instance was very fast, we decided to test whether this distinguisher could be extended to more rounds in each direction. Thus, we tried to prepend one more round at the

beginning or to append one more round at the end, or both. The propagation of the differences is shown in Table 2. In this table, round r is the round on which the differential starts. For the attack of [37] with 2 prepended rounds $r = 2$, while if we prepend 3 rounds then $r = 3$.

Table 3 summarizes the results. The best configuration for a concrete total number of rounds is shown in blue. We see for example, that when adding 2 key recovery rounds at the beginning and 3 at the end, it is possible to obtain a valid attack on `RECTANGLE-128`. Indeed, in this case $d_{in} = 52$, $\mathrm{d}_{out} = 28$, so $N = 2^{p+d_{in}-(n-d_{out})} = 2^{78.83}$. For this configuration, the tool returned a key recovery complexity of 2^{43}. This complexity is optimal in the sense that it corresponds to the number of expected solutions. This gives an attack with time complexity $2^{78.83+43} = 2^{121.83}$, which is smaller than the exhaustive search for the 128-bit key version. This would lead to the the first attack on `RECTANGLE-128` reaching 19 rounds in the single-key setting if there would have been no problem with the differential distinguisher. As can be seen from Table 3 we also launched the tool with three added key recovery rounds in both directions, but the returned complexity of 2^{70} was too high to lead to a valid attack.

Cipher	nr_p	nr_c	#Rounds $(14 + nr_p + nr_c)$	d_{in}	d_{out}	N $(2^{p+d_{in}+d_{out}-n})$	C_{KR} $(\cdot N)$	Valid attack
`RECTANGLE-80`	2	2	18	24	28	$2^{50.83}$	2^{19}	✓
`RECTANGLE-128`	2	3	19	24	56	$2^{78.83}$	2^{46}	✓
`RECTANGLE-128`	3	2	19	52	28	$2^{78.83}$	2^{43}	✓
`RECTANGLE-128`	3	3	20	52	56	$2^{106.83}$	2^{70}	✗

Table 3. Summary of the results on `RECTANGLE`. The column C_{KR} corresponds to the cost of the key recovery given by our tool and should be multiplied by the number of pairs N. If $(N \cdot C_{KR}) < 80$ then we get a valid attack against both versions of `RECTANGLE`. If furthermore $(N \cdot C_{KR}) < 128$, then we obtain a valid attack against `RECTANGLE-128`. Best attacks are highlighted in blue.

5.3 `PRESENT`

The `PRESENT` block cipher was designed by Bogdanov, Knudsen, Leander, Paar, Poschmann, Robshaw, Seurin and Vikkelsoe in 2007 [9]. Similar to `RECTANGLE`, it uses a 64-bit state where the state can be seen as a concatenation of 16 nibbles. Its round function also consists of an XOR with the round key, the application of a 4-bit S-box in parallel to the state and a bit-permutation. The S-box of the cipher, its Difference Distribution Table (DDT) as well as the linear permutation are described in Annex D. The round function is iterated 31 times with a final

whitening subkey. PRESENT supports keys of 80 or 128 bits. The key schedule of PRESENT-80 is also described in Annex D.

The best attacks on PRESENT are linear attacks reaching 28 rounds [20] and 29 rounds [19] on the 80-bit and 128-bit versions respectively. To illustrate the efficiency of our tool, we consider here the best known differential attacks. Indeed, PRESENT is a very interesting example as it shows that our tool can be efficient with not only linear but almost linear key schedules. To analyze PRESENT, we considered an alternative linear key schedule. We replaced the S-box by a matrix multiplication with a randomly generated non-singular matrix in $\mathcal{M}_4(\mathbb{F}_2)$. Once the tool found an efficient key recovery strategy, we adapted the attack to the real key schedule of PRESENT, and verified our result by hand.

The differential attack of [35] In 2007, Wang presented a differential attack against 16-round PRESENT-80. This attack was based on the 14-round differential distinguisher

$$\mathsf{0700\,0000\,0000\,0700} \longrightarrow_{14r} \mathsf{0000\,0009\,0000\,0009},$$

of probability 2^{-62}. Two rounds were appended to this distinguisher on the ciphertext side. This led to an attack with data complexity 2^{64} and time complexity of 2^{65} (measured in number of memory accesses). [7]

We applied our tool to PRESENT-80 with this distinguisher and tried different configurations for the key recovery rounds. More precisely, we tried to append up to 4 rounds to the end and to prepend at most 4 rounds in the beginning. The propagation of the differences is shown in Figure 4. As before, in this table, r is the round number on which the differential starts. For the original attack of Wang with 0 prepended rounds $r = 0$, while if we prepend 1 round (resp. 2 or 3 rounds), $r = 1$ (resp. $r = 2, 3$). The state before the S-boxes application of round r is denoted by ΔI_r, while that after the S-boxes and before the linear layer is denoted by ΔO_r.

For all the relevant configurations we first computed the number of pairs based on the related spaces D_{in} and D_{out} (see Section 2). Then, we launched the tool with this number of pairs to see if a solution was found. As for most of the configurations the execution took a few seconds only, it was easy to automatically test all the attack scenarios. For the configuration with $nr_p = 4$ and $nr_c = 0$ the execution was slow so we ran the tool with the option `- time x`, for `x` the smallest value such that $\log_2(N) + x \geq 80$. This option allows to search only for attacks with time complexity $\leq N \cdot 2^x$ and greatly accelerates the research. This permitted us to check that there was no valid attack in this setting. We also checked all configurations that would permit to reach an attack on 19 rounds of PRESENT-80 for this distinguisher and confirmed that it is not possible to obtain a valid attack on this number of rounds.

[7] Tezcan claimed later in [33] that there were errors in the key recovery procedure, providing a corrected attack for the same number of rounds with the same distinguisher.

S-box	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ΔI_{r-4}	****	****	****	****	****	****	****	****	****	****	****	****	****	****	****	****
ΔO_{r-4}	****	****	****	****	****	****	****	****	****	****	****	****	****	****	****	****
ΔI_{r-3}	****	****	****	****	****	****	****	****	****	****	****	****	****	****	****	****
ΔO_{r-3}	.***	.***	.***	***.	.***	.***	.***	.***	.***	.***	.***	***.	.***	.***	.***	.***
ΔI_{r-2}	....	....	....	....	****	****	****	****	****	****	****	****	****	****	****	****
ΔO_{r-2}	....	....	....	....	.*..	.*..	.*..	.*..	.*..	.*..	.*..	.*..	.*..	.*..	.*..	.*..
ΔI_{r-1}	....	....	....	....	....	****	****	****	....	....	....	....	....	....	....	....
ΔO_{r-1}	....	....	....	....	....	1..1	1..1	1..1	....	....	....	....	....	....	....	....
ΔI_{r}	....	.111	....	....	....	....	....	....	....	....	....	....	....	.111	....	....
14-round distinguisher																
ΔI_{r+14}	....	....	....	....	....	....	....	1..1	....	....	....	....	....	....	....	1..1
ΔO_{r+14}	....	....	....	....	....	....	....	***0	....	....	....	....	....	....	....	***0
ΔI_{r+15}	....	...*	....	...*	....	...*	....	...*	....	...*	....	...*	....	...0	....	...0
ΔO_{r+15}	....	****	....	****	....	****	....	****	....	****	....	****	....	0000	....	0000
ΔI_{r+16}	.*.*	.*.*	.*.*	.0.0	.*.*	.*.*	.*.*	.0.0	.*.*	.*.*	.*.*	.0.0	.*.*	.*.*	.*.*	.0.0
ΔO_{r+16}	****	****	****	0000	****	****	****	0000	****	****	****	0000	****	****	****	0000
ΔI_{r+17}	***0	***0	***0	***0	***0	***0	***0	***0	***0	***0	***0	***0	***0	***0	***0	***0
ΔO_{r+17}	****	****	****	****	****	****	****	****	****	****	****	****	****	****	****	****
ΔI_{r+18}	****	****	****	****	****	****	****	****	****	****	****	****	****	****	****	****
ΔO_{r+18}	****	****	****	****	****	****	****	****	****	****	****	****	****	****	****	****

Table 4. Differential attack on $(14 + x)$-round PRESENT based on the 14-round distinguisher from [35]. ΔI_r and ΔO_r are respectively the states before the S-box layer of round r. The symbol '.' stands for a non-active bit, '1' for an active bit, and '*' stands for a bit with an unknown difference (active of inactive). Finally, '0' corresponds to a bit with 0 difference but whose value needs to be known for the key recovery.

Parameters and complexities of the attacks. For all the attacks, as the probability of the distinguisher is $2^{-p} = 2^{-62}$, the data complexity is $2^{p+1} = 2^{63}$ according to the formulas of Section 2. For each configuration (number of nr_p prepended rounds and nr_c appended rounds), we determine the number of initial pairs N the attack should start with and provide the complexity given by the tool. For the notation of the parameters, we refer to Section 2.

Our results against PRESENT-80 are summarized in Table 5.

As can be seen from Table 5 we managed to extend the previous best differential attack on PRESENT-80 by 2 rounds. Notably for 18 rounds, the complexity of the key recovery stage given by our tool is $C_{KR} = 2^{59}$. In this case, as only 1 round is appended in the beginning, and as the 3 active input S-boxes are continuous, the attack applies directly to the real key schedule, so to PRESENT-80 with 18 rounds. By guessing one extra key bit, that then gets canceled out before the bottleneck of the process, we manage to recover all the key bits with the non-linear key schedule.

5.4 GIFT-64

GIFT-64 is a member of the GIFT family of lightweight block ciphers designed by Banik, Pandey, Peyrin, Sasaki, Sim and Todo [1]. It is a 64-bit block cipher with a 128-bit key and is composed of 28 rounds. It is a classical SPN cipher

Cipher	nr_p	nr_c	#Rounds $(14 + nr_p + nr_c)$	d_{in}	d_{out}	N $(2^{p+d_{in}+d_{out}-n})$	C_{KR} $(\cdot N)$	Valid attack
PRESENT-80	0	2	16	6	24	2^{28}	1	✓
PRESENT-80	0	3	17	6	48	2^{52}	1	✓
PRESENT-80	1	2	17	12	24	2^{34}	2^4	✓
PRESENT-80	2	1	17	48	6	2^{52}	2^8	✓
PRESENT-80	3	0	17	64	4	2^{62}	$2^{10.62}$	✓
PRESENT-80	0	4	18	6	64	2^{68}	$> 2^{12}$	✗
PRESENT-80	1	3	18	12	48	2^{58}	2^1	✓
PRESENT-80	2	2	18	48	24	2^{70}	1	✓
PRESENT-80	3	1	18	64	6	2^{68}	2^9	✓
PRESENT-80	4	0	18	64	4	2^{66}	$> 2^{14}$	✗

Table 5. Summary of the results on PRESENT-80. The column C_{KR} corresponds to the cost of the key recovery given by our tool and should be multiplied by the number of pairs N. An attack against PRESENT-80 is valid if the complexity of the key recovery is lower than 2^{80}. The best attacks are highlighted in blue.

whose state can be divided into 16 nibbles. First, the round key is XORed to the state. A particularity of GIFT is that the key is only added to half of the state, and more precisely to all bits at position b such that $b = 0 \pmod 4$ or $b = 1 \pmod 4$. Then, a 4-bit S-box is applied in parallel to all nibbles of the state and this application is followed by a bit-permutation. The S-box, together with its DDT and the bit-permutation of GIFT-64 can be found in Supplementary Material F. The key schedule consists in a bit-permutation of the master key and is described in Supplementary Material F.

The related-key differential attack of [31] Sun et al. provided a differential attack on 26 rounds of GIFT-64 in the related-key setting. This attack was based on a 18-round related-key differential distinguisher of probability $2^{-p} = 2^{-58}$:

$$\mathtt{0000\ 6000\ 0000\ 0600} \longrightarrow_{18r} \mathtt{0000\ 0014\ 0000\ 0041}.$$

The difference on the 128-bit master key is taken to be as follows:[8]

$$\mathtt{0000\ 1400\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000}.$$

The authors prepended 3 key recovery rounds in the beginning and appended 5 key recovery rounds at the end to mount an attack with data, time and memory complexities equal to $\mathcal{D} = 2^{60.96}, \mathcal{T} = 2^{123.23}, \mathcal{M} = 2^{102.86}$. The differential propagation on the key recovery rounds can be visualized in Table 6.

[8]Note that in [31] the least significant bit (LSB) was taken on the left, contrary to the original cipher's description [1]. Here, we stick to the original notation and place the LSB on the right.

S-box	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ΔI_0	****	****	****	****	****	****	****	****	****	****	****	****	****	****	****	****
ΔO_0	****	****	****	****	1***	11**	*1**	****	*1**	****	1***	11**	****	****	****	****
ΔI_1	****	****	11..	****	****	****	11..	****	****	11.	****	****	****	11.	****	****
ΔO_1	...*	1...	.1..	..*.	...*	*...	.1..	..*.	1...	.1..	..*.	...*	*...	.1..	..*.	...*
ΔI_2	....	....	....	....	....	....	....	....	11**	*1**	....	....	....	....	11**	*1**
ΔO_2	....	....	....	....	....	....	....	....	.1..	..1.	....	....	....	....	.1..	..1.
ΔI_3	....	....	.11.	....	....	....	....	....	....	....	....	.11.	....	....	....	....
18-round related-key differential distinguisher																
ΔI_{21}	...1	.1..	....	....	....	....	....	....	.1..	...1	....	....	....	....	....	....
ΔO_{21}	****	***1	....	....	....	....	....	....	***1	****	....	....	....	....	....	....
ΔI_{22}	.**.	....	.**.	....	..*1	....	..**	....	*..*	....	*..1	....	**..	....	**..	....
ΔO_{22}	****	....	****	....	****	....	****	...	****	....	****	....	****	....	****	....
ΔI_{23}	.*.*	.*.*	.*.*	.*.*	.*.*	.*.*	.*.*	.*.*	.*.*	.*.*	.*.*	.*.*	.*.*	.*.*	.*.*	.*.*
ΔO_{23}	****	****	****	****	****	****	****	****	****	****	****	****	****	****	****	****
ΔI_{24}	****	****	****	****	****	****	****	****	****	****	****	****	****	****	****	****
ΔO_{24}	****	****	****	****	****	****	****	****	****	****	****	****	****	****	****	****
ΔI_{25}	****	****	****	****	****	****	****	****	****	****	****	****	****	****	****	****
ΔO_{25}	****	****	****	****	****	****	****	****	****	****	****	****	****	****	****	****

Table 6. Differential attack on 26-round `GIFT-24` based on the 18-round related-key distinguisher from [31].

Application of our tool and comparison with the attack of [31]. Using the same distinguisher and attack parameters as in [31], our tool outputted a key recovery strategy of complexity $N = 2^{115.96}$, that is also the global complexity of the attack. This strategy improved thus the attack of [31] by a factor of $2^{7.27}$. Based on the same setup, the work of [14] improved the complexity of the attack of [31] with a very tedious procedure. It is important to note that while this attack uses refined techniques such as tree-based key recovery techniques and key absorption from [13], our tool was able to derive a more efficient procedure.

5.5 Application to `SPEEDY-7-192`

`SPEEDY-7-192` is a member of the `SPEEDY` family of low-latency block ciphers introduced by Leander, Moos, Moradi and Rasoolzadeh at CHES 2021 [24]. We provide its specification in Annex E. Last year, a differential attack on the full version of `SPEEDY-7-192` was published [12]. This attack exploited a 5.5-round differential of probability $2^{-p} = 2^{-183.59}$ that was extended to one round backwards and half a round forwards. The key recovery in [12] was done by hand, requiring a particularly tedious procedure. We decided to launch our tool on this cipher, keeping the same parameters as in [12], in order to show the applicability of our tool on a different cipher and to see if the key recovery complexity could be improved. Note however that in a very recent note [3], subsequent to ours, the authors claim that the distinguisher used is not valid due to the existence of quasidifferentials cancelling the probability. Our tool outputted a complexity for the key recovery phase equal to N improving thus by a factor of $2^{2.83}$ the key recovery complexity of [12]. As this term was not the bottleneck of the attack, the improvement in the overall time complexity is small: $2^{0.5}$. This application

shows however that our tool can complete in a few seconds a procedure that would be extremely long by hand. Our tool will be automatically applicable to any new and valid differential distinguisher that gets presented for this cipher in the future. More details can be found in Annex E.

6 Conclusion and open problems

In this paper, we have proposed the first algorithm (and an automatic tool that implements it) to find efficient key recovery strategies in differential attacks. This permitted us to find efficient key recovery strategies for the attacks on many ciphers. We believe that our tool, which will be publicly available, will be of great help to cryptanalysts, but also to designers, as it will assist them in mounting attacks and in choosing optimal parameters for their construction.

We believe that the proposed tool is the first step towards a fully automated treatment of differential attacks. Many extensions and improvements can be considered. Indeed, our tool can for the moment only handle block ciphers with a bit-permutation linear layer and a linear or almost-linear key schedule. A natural extension is to adapt the tool such that it applies to ciphers with more complex linear layers, based for example on an MDS multiplication. Another improvement would be to adapt the tool to ciphers with non-linear key schedules as currently, our tool needs the user to either linearise non-linear key schedules or replace non-linear equations with new variables. Another interesting direction is to adapt the tool to take into account tree-based key recovery techniques by exploiting the structure of the involved S-boxes, as those proposed in [13].

Finally, the ultimate goal would be to combine this tool with algorithms that search for differential distinguishers to propose a complete tool for differential cryptanalysis that would produce attacks based on differential distinguishers that are the best adapted for the key recovery. This is a challenging but particularly important task, as it is known that the best distinguisher does not always lead to the best attack.

References

1. 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 2017. Lecture Notes in Computer Science, vol. 10529, pp. 321–345. Springer (2017)
2. Beierle, C., Broll, M., Canale, F., David, N., Flórez-Gutiérrez, A., Leander, G., Naya-Plasencia, M., Todo, Y.: Improved differential-linear attacks with applications to ARX ciphers. J. Cryptol. **35**(4), 29 (2022)
3. Beyne, T., Neyt, A.: Note on the cryptanalysis of speedy. Cryptology ePrint Archive, Paper 2024/262 (2024), `https://eprint.iacr.org/2024/262`
4. Beyne, T., Rijmen, V.: Differential cryptanalysis in the fixed-key model. In: Dodis, Y., Shrimpton, T. (eds.) CRYPTO 2022, Part III. Lecture Notes in Computer Science, vol. 13509, pp. 687–716. Springer (2022)

5. Biham, E., Shamir, A.: Differential cryptanalysis of DES-like cryptosystems. In: Menezes, A., Vanstone, S.A. (eds.) CRYPTO '90. Lecture Notes in Computer Science, vol. 537, pp. 2–21. Springer (1990)
6. Biham, E., Shamir, A.: Differential cryptanalysis of Feal and N-Hash. In: Davies, D.W. (ed.) EUROCRYPT '91. Lecture Notes in Computer Science, vol. 547, pp. 1–16. Springer (1991)
7. Biham, E., Shamir, A.: Differential cryptanalysis of Snefru, Khafre, REDOC-II, LOKI and Lucifer. In: Feigenbaum, J. (ed.) CRYPTO '91. Lecture Notes in Computer Science, vol. 576, pp. 156–171. Springer (1991)
8. Biham, E., Shamir, A.: Differential cryptanalysis of the full 16-round DES. In: Brickell, E.F. (ed.) CRYPTO '92. Lecture Notes in Computer Science, vol. 740, pp. 487–496. Springer (1992)
9. 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. Lecture Notes in Computer Science, vol. 4727, pp. 450–466. Springer (2007)
10. Bouillaguet, C., Derbez, P., Fouque, P.: Automatic search of attacks on round-reduced AES and applications. In: Rogaway, P. (ed.) CRYPTO 2011. Lecture Notes in Computer Science, vol. 6841, pp. 169–187. Springer (2011)
11. Bouillaguet, C., Derbez, P., Fouque, P.: Automatic search of attacks on round-reduced AES and applications. In: Rogaway, P. (ed.) CRYPTO 2011. Lecture Notes in Computer Science, vol. 6841, pp. 169–187. Springer (2011)
12. Boura, C., David, N., Boissier, R.H., Naya-Plasencia, M.: Better steady than speedy: Full break of SPEEDY-7-192. In: Hazay, C., Stam, M. (eds.) EUROCRYPT 2023, Proceedings, Part IV. Lecture Notes in Computer Science, vol. 14007, pp. 36–66. Springer (2023)
13. Broll, M., Canale, F., Flórez-Gutiérrez, A., Leander, G., Naya-Plasencia, M.: Generic framework for key-guessing improvements. In: Tibouchi, M., Wang, H. (eds.) ASIACRYPT 2021, Part I. Lecture Notes in Computer Science, vol. 13090, pp. 453–483. Springer (2021)
14. Canale, F., Naya-Plasencia, M.: Guessing less and better: Improved attacks on GIFT-64. IACR Cryptol. ePrint Arch. p. 354 (2023), https://eprint.iacr.org/2023/354
15. Canteaut, A., Naya-Plasencia, M., Vayssière, B.: Sieve-in-the-middle: Improved MITM attacks. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013, Part I. Lecture Notes in Computer Science, vol. 8042, pp. 222–240. Springer (2013)
16. Daemen, J., Rijmen, V.: The wide trail design strategy. In: Honary, B. (ed.) IMACC 2001. Lecture Notes in Computer Science, vol. 2260, pp. 222–238. Springer (2001)
17. Derbez, P., Euler, M., Fouque, P., Nguyen, P.H.: Revisiting related-key boomerang attacks on AES using computer-aided tool. In: Agrawal, S., Lin, D. (eds.) ASIACRYPT 2022, Part III. Lecture Notes in Computer Science, vol. 13793, pp. 68–88. Springer (2022)
18. Dong, X., Qin, L., Sun, S., Wang, X.: Key guessing strategies for linear key-schedule algorithms in rectangle attacks. In: Dunkelman, O., Dziembowski, S. (eds.) EUROCRYPT 2022, Part III. Lecture Notes in Computer Science, vol. 13277, pp. 3–33. Springer (2022)
19. Flórez-Gutiérrez, A.: Optimising linear key recovery attacks with affine Walsh transform pruning. In: Agrawal, S., Lin, D. (eds.) ASIACRYPT 2022, Part IV. Lecture Notes in Computer Science, vol. 13794, pp. 447–476. Springer (2022)

20. 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, Part I. Lecture Notes in Computer Science, vol. 12105, pp. 221–249. Springer (2020)
21. Fouque, P., Jean, J., Peyrin, T.: Structural evaluation of AES and chosen-key distinguisher of 9-round AES-128. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013, Part I. Lecture Notes in Computer Science, vol. 8042, pp. 183–203. Springer (2013)
22. Hadipour, H., Sadeghi, S., Eichlseder, M.: Finding the impossible: Automated search for full impossible-differential, zero-correlation, and integral attacks. In: Hazay, C., Stam, M. (eds.) EUROCRYPT 2023, Part IV. Lecture Notes in Computer Science, vol. 14007, pp. 128–157. Springer (2023)
23. Knellwolf, S., Meier, W., Naya-Plasencia, M.: Conditional differential cryptanalysis of NLFSR-based cryptosystems. In: Abe, M. (ed.) ASIACRYPT 2010. Lecture Notes in Computer Science, vol. 6477, pp. 130–145. Springer (2010)
24. Leander, G., Moos, T., Moradi, A., Rasoolzadeh, S.: The SPEEDY family of block ciphers engineering an ultra low-latency cipher from gate level for secure processor architectures. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2021**(4), 510–545 (2021)
25. Lu, J., Kim, J., Keller, N., Dunkelman, O.: Improving the efficiency of impossible differential cryptanalysis of reduced Camellia and MISTY1. In: Malkin, T. (ed.) CT-RSA 2008. Lecture Notes in Computer Science, vol. 4964, pp. 370–386. Springer (2008)
26. Naya-Plasencia, M.: How to improve rebound attacks. In: Rogaway, P. (ed.) CRYPTO 2011. Lecture Notes in Computer Science, vol. 6841, pp. 188–205. Springer (2011)
27. Peyrin, T., Tan, Q.Q.: Mind your path: On (key) dependencies in differential characteristics. IACR Trans. Symmetric Cryptol. **2022**(4), 179–207 (2022)
28. Qiao, K., Hu, L., Sun, S.: Differential security evaluation of Simeck with dynamic key-guessing techniques. In: Camp, O., Furnell, S., Mori, P. (eds.) ICISSP 2016. pp. 74–84. SciTePress (2016)
29. Qin, L., Dong, X., Wang, X., Jia, K., Liu, Y.: Automated search oriented to key recovery on ciphers with linear key schedule applications to boomerangs in SKINNY and ForkSkinny. IACR Trans. Symmetric Cryptol. **2021**(2), 249–291 (2021)
30. Rouquette, L., Gérault, D., Minier, M., Solnon, C.: And Rijndael?: Automatic related-key differential analysis of Rijndael. In: Batina, L., Daemen, J. (eds.) AFRICACRYPT 2022. pp. 150–175. Lecture Notes in Computer Science, Springer Nature Switzerland (2022)
31. 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)
32. Sun, S., Hu, L., Wang, P., Qiao, K., Ma, X., Song, L.: Automatic security evaluation and (related-key) differential characteristic search: Application to SIMON, PRESENT, LBlock, DES(L) and other bit-oriented block ciphers. In: Sarkar, P., Iwata, T. (eds.) ASIACRYPT 2014, Part I. Lecture Notes in Computer Science, vol. 8873, pp. 158–178. Springer (2014)
33. Tezcan, C.: Differential factors revisited: Corrected attacks on PRESENT and SERPENT. In: Güneysu, T., Leander, G., Moradi, A. (eds.) LightSec 2015. Lecture Notes in Computer Science, vol. 9542, pp. 21–33. Springer (2015)
34. Vaudenay, S.: Decorrelation: A theory for block cipher security. J. Cryptol. **16**(4), 249–286 (2003)

35. Wang, M.: Differential cryptanalysis of reduced-round PRESENT. In: Vaudenay, S. (ed.) AFRICACRYPT 2008. Lecture Notes in Computer Science, vol. 5023, pp. 40–49. Springer (2008)
36. Wang, N., Wang, X., Jia, K., Zhao, J.: Differential attacks on reduced SIMON versions with dynamic key-guessing techniques. Sci. China Inf. Sci. **61**(9), 098103:1–098103:3 (2018)
37. 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)
38. 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)

A Toy cipher description for the tool

We provide here the file describing the toy cipher, formatted in the way our tool can read and interpret it. For saving space, we removed the last relations for the round keys, but the complete file can be in our `Git` depository. Note, that in this example, the only round key relations that have to be written are those indicating that there is no key addition at state positions `2 mod 4` and `3 mod 4`. This is done by artificially considering one key bit for each such position and interpreting is as zero.

```
------------------------- toy_cipher.txt ------------------------
SB 1 10 4 12 6 15 3 9 2 13 11 7 5 0 8 14
PERM 4 5 2 3 12 13 10 11 0 1 6 7 8 9 14 15
nrSBp 3
nrSBc 3
DINa 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

DINb 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3

DOUTb 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0

DOUTa 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3

DKEY 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

```
nrR 9

KS
k0[2]
k0[3]
k0[6]
k0[7]
k0[10]
k0[11]
k0[14]
k0[15]
-----------------------------------------------------------------
```

B Comparison with previous work

The tool we propose in this work has certain similarities with a previous tool proposed in [11] by Bouillaguet et al. to optimize a different type of attacks. We discuss in this section the similarities that method has with our tool and explain the advantages of our approach. The aim of the automatic tool presented in [11] was to search for meet-in-the-middle attacks on a large variety of block ciphers. More precisely, the authors described in this article a generic framework to solve a system of AES-like equations, i.e. equations of the form:

$$\sum \alpha_i x_i + \sum \beta_i S(x_i) = \gamma,$$

where S is an S-box, and α_i, β_i and γ are constant values in $\mathbb{F}_{2^n}$. Since such equations are *separable*, i.e. $f(X \cup Y) = 0$ can always be rewritten as $\{g(X) = h(Y), g_X(X) = 0, h_Y(Y) = 0\}$ for any set of variables X and any set of variables Y, their main idea was to solve a system of such equations by using a divide-and-conquer approach in which, given an algorithm $\mathcal{A}_X$ solving the equation $g_X(X) = 0$ as well as an algorithm $\mathcal{A}_Y$ solving the equation $h_Y(Y) = 0$, we can combine them into a new algorithm $\mathcal{A}_{X \cup Y}$ solving $f(X \cup Y) = 0$. Furthermore, the time complexity of this new algorithm can be easily computed as:

$$T(\mathcal{A}_{X \cup Y}) = T(\mathcal{A}_X) + T(\mathcal{A}_Y) + \text{number of solutions of } f(X \cup Y) = 0.$$

We refer the reader to [11] for more details on the topic.

While this approach gave very good results, especially regarding low data complexity attacks on the `AES`, it has many limitations. First the properties of the S-box are not taken into account since the S-box is seen as a black box. Second, they only consider the case in which the system is word-oriented, meaning that the S-box depends on only one variable. Third, the only freedom for the user is to declare some variables of the system as *known* variables (for instance the plaintext and the ciphertext can be set as known) but it is not possible to emulate the situation of the key-recovery process of a differential attack. Finally,

the merge between the various sub-algorithms occurring in the problem one wants to solve are performed randomly, making it difficult to solve big instances, especially when we want to ensure the optimality of the solution.

Our work was inspired by this tool but our goal was to provide a solution in the specific case of differential attacks for SPN ciphers with a bit-permutation as linear layer, by overcoming the weaknesses of [11]. Our contributions include a natural modelization of the problem as a graph to limit the number of merges to be performed through the execution of our tool as well as taking into account filters at the input of each S-box leading to a much better granularity of the number of solutions at each step of the process and thus to a more accurate overall complexity of the key recovery attack.

C Specifications and differential properties of `RECTANGLE`

We provide in this section the specifications of the S-box (Table 7) and of the bit-permutation (Table 8) used in `RECTANGLE`. For completeness, we also provide the Difference Distribution Table (DDT) of the S-box in Table 9.

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

Table 7. The S-box of `RECTANGLE`

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
P(i)	0	5	50	55	4	9	54	59	8	13	58	63	12	17	62	3
i	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
P(i)	16	21	2	7	20	25	6	11	24	29	10	15	28	33	14	19
i	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
P(i)	32	37	18	23	36	41	22	27	40	45	26	31	44	49	30	35
i	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
P(i)	48	53	34	39	52	57	38	43	56	61	42	47	60	1	46	51

Table 8. The bit-permutation of `RECTANGLE`

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	0	0	4	2	0	0	0	2	0	0	4	2
2	0	0	0	0	0	0	2	2	2	0	2	0	2	4	0	2
3	0	0	0	2	0	0	2	0	2	4	2	2	2	0	0	0
4	0	0	0	4	0	0	0	4	0	0	0	4	0	0	0	4
5	0	2	0	0	4	2	0	0	4	2	0	0	0	2	0	0
6	0	2	4	0	2	0	0	0	0	0	0	2	2	2	0	2
7	0	0	4	0	2	2	0	0	0	2	0	2	2	0	0	2
8	0	2	0	2	0	2	0	2	0	2	0	2	0	2	0	2
9	0	2	0	0	0	2	4	0	0	2	0	0	0	2	4	0
a	0	0	0	0	0	4	2	2	2	0	2	0	2	0	0	2
b	0	4	0	2	0	0	2	0	2	0	2	2	2	0	0	0
c	0	0	0	0	4	0	0	0	4	0	4	0	0	0	4	0
d	0	2	0	0	0	2	0	0	0	2	4	0	0	2	4	0
e	0	0	4	2	2	2	0	2	0	2	0	0	2	0	0	0
f	0	2	4	2	2	0	0	2	0	0	0	0	2	2	0	0

Table 9. The Difference Distribution Table (DDT) of the RECTANGLE S-box

D Specifications and differential properties of PRESENT

We provide here the specifications of the S-box (Table 10) and of the bit-permutation layer (Table 11) used in PRESENT. For completeness, we also provide the Difference Distribution Table (DDT) of the S-box in Table 12.

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

Table 10. The S-box of PRESENT

Key schedule for PRESENT-80. The master key is stored in a register K and is represented as $\mathtt{k}_{79}\mathtt{k}_{78}\ldots\mathtt{k}_0$. At round i, the round key K_i consists of the 64 leftmost bits of the current content of the register K:

$$K_i = \mathtt{k}_{79}\mathtt{k}_{78}\ldots\mathtt{k}_{16}.$$

Once the round key has been extracted, the register K is updated as follows:

$$[\mathtt{k}_{79}\mathtt{k}_{78}\ldots\mathtt{k}_1\mathtt{k}_0] = [\mathtt{k}_{18}\mathtt{k}_{17}\ldots\mathtt{k}_{20}\mathtt{k}_{19}]$$
$$[\mathtt{k}_{79}\mathtt{k}_{78}\mathtt{k}_{77}\mathtt{k}_{76}] = S[\mathtt{k}_{79}\mathtt{k}_{78}\mathtt{k}_{77}\mathtt{k}_{76}]$$
$$[\mathtt{k}_{19}\mathtt{k}_{18}\mathtt{k}_{17}\mathtt{k}_{16}\mathtt{k}_{15}] = [\mathtt{k}_{19}\mathtt{k}_{18}\mathtt{k}_{17}\mathtt{k}_{16}\mathtt{k}_{15}] \oplus \mathtt{round_counter}.$$

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$P(i)$	0	16	32	48	1	17	33	49	2	18	34	50	3	19	35	51
i	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
$P(i)$	4	20	36	52	5	21	37	53	6	22	38	54	7	23	39	55
i	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
$P(i)$	8	24	40	56	9	25	41	57	10	26	42	58	11	27	43	59
i	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
$P(i)$	12	28	44	60	13	29	45	61	14	30	46	62	15	31	47	63

Table 11. The bit-permutation of PRESENT

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	4	0	0	0	4	0	4	0	0	0	4	0	0
2	0	0	0	2	0	4	2	0	0	0	2	0	2	2	2	0
3	0	2	0	2	2	0	4	2	0	0	2	2	0	0	0	0
4	0	0	0	0	0	4	2	2	0	2	2	0	2	0	2	0
5	0	2	0	0	2	0	0	0	0	2	2	2	4	2	0	0
6	0	0	2	0	0	0	2	0	2	0	0	4	2	0	0	4
7	0	4	2	0	0	0	2	0	2	0	0	0	2	0	0	4
8	0	0	0	2	0	0	0	2	0	2	0	4	0	2	0	4
9	0	0	2	0	4	0	2	0	2	0	0	0	2	0	4	0
a	0	0	2	2	0	4	0	0	2	0	2	0	0	2	2	0
b	0	2	0	0	2	0	0	0	4	2	2	2	0	2	0	0
c	0	0	2	0	0	4	0	2	2	2	2	0	0	0	2	0
d	0	2	4	2	2	0	0	2	0	0	2	2	0	0	0	0
e	0	0	2	2	0	0	2	2	2	2	0	0	2	2	0	0
f	0	4	0	0	4	0	0	0	0	0	0	0	0	0	4	4

Table 12. The Difference Distribution Table (DDT) of the PRESENT S-box

We refer the reader to [9] for the description of the key schedule of the 128-bit version.

E Application to SPEEDY-7-192

SPEEDY-7-192 is a 192-bit cipher, whose state is seen as a (32×6)-bit array. The round function consists of the following operations: An AddRoundKey(A_{k_r}) operation that XORs a subkey k_r to the state. Then, SubBox(SB) applies a 6-bit S-box to the rows in parallel. Follows the ShiftColumns(SC) operation that rotates each column of the state by a different offset. The SB and the SC operation are repeated twice inside a round. Then, the MixColumns(MC) operation multiplies each column of the state by a binary matrix. We omit here the constant

addition operation (AC) that is of no interest to us. For SPEEDY-7-192 the round function is iterated 7 times, but in the last round the SC, MC and AC operations are omitted. The round function of this cipher is depicted in Figure 4.

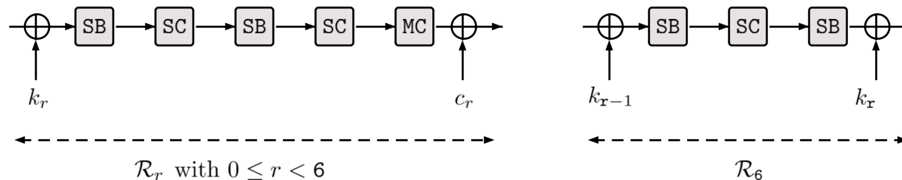

Fig. 4. The round function of SPEEDY-7-192 for the first 6 rounds (left) and the last round (right).

Finally, the key schedule is a simple bit-permutation of the 192-bit master key. This permutation can be found in [24].

The differential attack of [12] Recently, an attack on the full version of SPEEDY-7-192 using a differential attack was published [12], even if the validity of the distinguisher used was recently questioned in [3] due to the existence of quasidifferentials cancelling the probability. This attack exploited a 5.5-round differential of probability $2^{-p} = 2^{-183.59}$ that was extended to one round backwards and half a round forwards. The data, time and memory complexities of this attack were $\mathcal{D} = 2^{187.28}$, $\mathcal{T} = 2^{187.84}$ and 2^{42} respectively. In particular the complexity of the key recovery was $N \cdot 2^{2.83}$. This attack can be depicted in Figure 5.

The key recovery in [12] was done by hand, requiring a particularly tedious procedure. We decided to launch our tool on this cipher in order to see if the key recovery complexity could be improved. This application required some subtleties regarding its implementation. Indeed, in a first approach the round function could seem to be too sophisticated for our framework since it features two SubBox(SB) layers as well as a MixColumns(MC) operation. However we can observe that the MC operation does not appear in the key recovery depicted in Figure 5, hence we do not need to consider it. Handling the double SB application, is also easy. For this, we just need to split the round function into two parts: $R_1 \circ R_0$ where $R_0 = \mathtt{SC} \circ \mathtt{SB} \circ \mathtt{A}_{\mathtt{k_r}}$ and $R_1 = \mathtt{SC} \circ \mathtt{SB} \circ \mathtt{A}_0$ where $\mathtt{A}_0$ corresponds to an artificial XOR with an all-0 round key.

Application of our tool to the differential of [12] We applied our tool to SPEEDY-7-192 by keeping the exact attack parameters as in [12]. The key recovery rounds depicted in Figure 5 feature patterns (represented in pink) that allowed a complexity tradeoff in the first steps of the attack. However, this pink part could not be used in the original attack, as it would have implied to propose a key recovery for each different pattern. This would have meant a

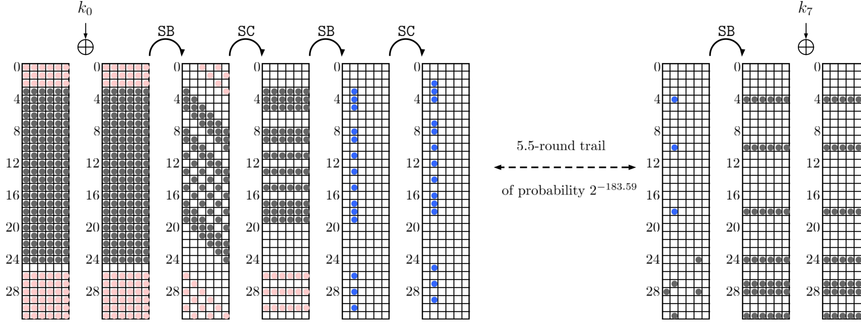

Fig. 5. The differential attack of [12] on full SPEEDY-7-192. Blue cells correspond to active bits and grey cells to bits with an unknown difference. Finally the pink part of the state corresponds to rows among which only 3 will be active in an attack scenario.

huge amount of work given the tedious procedure, and the authors preferred to propose a method for recovering first the key of the other rounds, and at the end the one associated to the pink part in a cheap step that did not need to be detailed. In order to fairly compare our results, we have not considered the pink part either. Hence we implemented our tool with the pink part as inactive and not allowing to filter. Our tool outputted a complexity for the key recovery phase equal to N improving thus by a factor of $2^{2.83}$ the complexity of the key recovery attack given in [12]. As this was not the bottleneck of the attack, the factor of improvement is small $2^{0.5}$, but this application shows how our tool can solve complex procedures automatically and even provides better results that the previous procedures.

F Specifications and differential properties of GIFT-64

We give the specifications of the S-box (Table 13) and of the bit-permutation (Table 14) used in GIFT-64. We also provide the Difference Distribution Table (DDT) of the S-box in Table 15. Finally, we describe the GIFT-64 key schedule.

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

Table 13. The S-box of GIFT

The key schedule of GIFT-64. The 128-bit master key can be seen as eight 16-bit words stored in a register $\mathtt{K} = \mathtt{k}_7 || \mathtt{k}_6 \cdots || \mathtt{k}_1 || \mathtt{k}_0$. At each iteration, first the 32-bit round key is extracted:

$$\mathtt{k_r} = \mathtt{k}_1 || \mathtt{k}_0,$$

then the register K is updated:

$$\mathtt{k}_7 || \mathtt{k}_6 \cdots || \mathtt{k}_1 || \mathtt{k}_0 \leftarrow \mathtt{k}_1 \ggg 2 || \mathtt{k}_0 \ggg 12 \cdots || \mathtt{k}_3 || \mathtt{k}_2.$$

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
P(i)	0	17	34	51	48	1	18	35	32	49	2	19	16	33	50	3
i	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
P(i)	4	21	38	55	52	5	22	39	36	53	6	23	20	37	54	7
i	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
P(i)	8	25	42	59	56	9	26	43	40	57	10	27	24	41	58	11
i	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
P(i)	12	29	46	63	60	13	30	47	44	61	14	31	28	45	62	15

Table 14. The bit-permutation of GIFT-64

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	2	2	0	2	2	2	2	2	0	0	2
2	0	0	0	0	0	4	4	0	0	2	2	0	0	2	2	0
3	0	0	0	0	0	2	2	0	2	0	0	2	2	2	2	2
4	0	0	0	2	0	4	0	6	0	2	0	0	0	2	0	0
5	0	0	2	0	0	2	0	0	2	0	0	0	2	2	2	4
6	0	0	4	6	0	0	0	2	0	0	2	0	0	0	2	0
7	0	0	2	0	0	2	0	0	2	2	2	4	2	0	0	0
8	0	0	0	4	0	0	0	4	0	0	0	4	0	0	0	4
9	0	2	0	2	0	0	2	2	2	0	2	0	2	2	0	0
a	0	4	0	0	0	0	4	0	0	2	2	0	0	2	2	0
b	0	2	0	2	0	0	2	2	2	2	0	0	2	0	2	0
c	0	0	4	0	4	0	0	0	2	0	2	0	2	0	2	0
d	0	2	2	0	4	0	0	0	0	0	2	2	0	2	0	2
e	0	4	0	0	4	0	0	0	2	2	0	0	2	2	0	0
f	0	2	2	0	4	0	0	0	0	2	0	2	0	0	2	2

Table 15. The Difference Distribution Table (DDT) of the GIFT S-box