Related-Key Differential Analysis of the AES

Christina Boura, Patrick Derbez, Margot Funk

IACR Transactions on Symmetric Cryptology
ISSN 2519-173X, Vol. 2023, No. 4, pp. 215–243. DOI:10.46586/tosc.v2023.i4.215-243

Related-Key Differential Analysis of the AES

Christina Boura[1], Patrick Derbez[2] and Margot Funk[1]

[1] Université Paris-Saclay, Université de Versailles, Centre National de la Recherche Scientifique (CNRS), Laboratoire de mathématiques de Versailles, 78000, Versailles, France
christina.boura@uvsq.fr, margot.funk@uvsq.fr
[2] Univ Rennes, Inria, Centre National de la Recherche Scientifique (CNRS), Institut de Recherche en Informatique et Systèmes Aléatoires (IRISA), Rennes, France
patrick.derbez@irisa.fr

Abstract. The Advanced Encryption Standard (AES) is considered to be the most important and widely deployed symmetric primitive. While the cipher was designed to be immune against differential and other classical attacks, this immunity does not hold in the related-key setting, and various related-key attacks have appeared over time. This work presents tools and algorithms to search for related-key distinguishers and attacks of differential nature against the AES. First, we propose two entirely different approaches to find optimal truncated differential characteristics and bounds on the minimum number of active S-boxes for all variants of the AES. In the first approach, we propose a simple MILP model that handles better linear inconsistencies with respect to the AES system of equations and that compares particularly well to previous tool-based approaches to solve this problem. The main advantage of this tool is that it can easily be used as the core algorithm to search for any attack on AES exploiting related-key differentials. Then, we design a fast and low-memory algorithm based on dynamic programming that has a very simple to understand complexity analysis and does not depend on any generic solver. This second algorithm provides us useful insight on the related-key differential search problem for AES and shows that the search space is not as big as one would expect. Finally, we build on the top of our MILP model a fully automated tool to search for the best differential MITM attacks against the AES. We apply our tool on AES-256 and find an attack on 13 rounds with only two related keys. This attack can be seen as the best known cryptanalysis against this variant if only 2 related keys are permitted.

Keywords: AES · differential related-key security · dynamic programming · MILP · differential MITM attack

1 Introduction

The block cipher Rijndael was designed by Joan Daemen and Vincent Rijmen in 1998. In 2000, a variant of Rijndael is selected by the National Institute of Standards and Technology (NIST) through the AES selection process and becomes the Advanced Encryption Standard (AES) in the following year [FIP01]. Since then, the AES has been widely deployed and is undeniably today the most popular and the most used symmetric primitive.

Since the very beginning, many different researchers and teams analyzed the security of the AES and proposed various attacks against reduced-versions of it. Against all these attacks, the three variants of the AES have proven to be extremely robust in the single-key setting. Indeed, the best attacks against AES-128 manage to break at most 7 out

All authors were partially supported by the French Agence Nationale de la Recherche through the OREO project under Contract ANR-22-CE39-0015.

Received: 2023-09-01 Accepted: 2023-11-01 Published: 2023-12-08

of the 10 rounds [DFJ13, BNS14, BNS19, LP21], while 9 (out of 12 and 14) rounds are reached by attacks against AES-192 and AES-256, respectively [LJW14, BDK+18]. However, the two larger variants of the AES were shown to be much weaker in the related-key setting and got fully broken in 2009 by Biryukov *et al.* with related-key attacks [BK09, BKN09]. Other attacks on the full-round AES-192 or AES-256 were presented later, notably boomerang attacks [GSW22, DEFN22] or attacks exploiting other properties [GLMS20] but necessitating a huge number of related-keys.

Understanding the security of the AES in the related-key setting is crucial as these attacks remain not well-understood and have not been generically analyzed. Among the most important attacks to be taken into account in a security analysis are differential attacks [BS90] and their variants. The AES was designed with resistance against differential attacks in mind [DR01] and all its variants are known to be resistant to differential cryptanalysis in the single-key setting. Analyzing the resistance of the AES against differential-based attacks in the related-key setting is a much more challenging task. A natural direction for this is to search for optimal differential characteristics and to establish bounds on the minimum number of S-boxes a differential characteristic can activate. This knowledge provides directly an upper bound on the probability of any differential characteristic and is naturally related to the success of attacks of differential nature. In this direction, Biryukov and Nikolic presented in [BN10] a branch-and-bound algorithm highly inspired from the original algorithm of Matsui [Mat94], to search for related-key differential characteristics for SPN ciphers. This permitted them to find optimal related-key characteristics for the two smaller variants of the AES, but doing so was very time consuming. Indeed, their algorithm needed several days of computation for AES-128 and several weeks for AES-192. Then, in 2013, Fouque *et al.* presented an alternative approach to solve this same problem [FJP13]. Their algorithm was based on dynamic programming and designed for AES-128. It was a very elegant solution with an easy to understand complexity but that consumed in return a high amount of memory (60 GB) and could not be extended to the larger variants of the AES. Finally, Gérault *et al.* in [GLMS18, GLMS20] and more recently Rouquette *et al.* in [RGMS22] developed a constraint-programming (CP) approach to search for optimal differential characteristics for AES-192 and AES-256 (and more generally for all variants of Rijndael) and managed to provide such characteristics for any number of rounds together with bounds for the minimum number of active S-boxes. This approach was fast and memory-efficient at the same time.

Our first contribution is a new MILP model for the optimal related-key differential characteristics problem for all three variants of the AES. To elaborate this model, we analyzed different strategies for eliminating invalid truncated characteristics without adding too many constraints and propose the best trade-off we found. Our model uses less variables and constraints than the CP approach of [RGMS22] and is faster for most of the AES instances analyzed. It is also more efficient than the MILP model used to find the boomerang characteristics in the work of Derbez *et al.* [DEFN22]. Its main advantage is however that it can be easily used as the core algorithm in any MILP model searching for related-key attacks of differential nature on AES. This is something we demonstrate in the second part of this paper.

Next we show that it is possible to use a simple and easy to understand algorithm to find differential bounds and characteristics for AES in the related-key setting, without the need of any generic solver. Similarly to [FJP13], we propose an algorithm based on dynamic programming, that uses a compact representation of the state to be memory-efficient. We applied this algorithm to all three versions of the AES and reproduced the results of [FJP13] and [RGMS22]. For AES-128 our method is much faster and requires much less memory than the one of [FJP13] and for the other two variants our computing time is better than in [RGMS22] for most of the instances. Furthermore, the algorithm we propose is simple and its complexity can be very easily and accurately estimated. Our approach invalidates in

particular the claim made in [RGMS22], where the authors write that dedicated approaches do not scale well for the optimal related-key differential characteristic problem and need weeks to solve its hardest instances. Moreover, it demonstrates that dynamic programming can be memory-efficient if cleverly implemented. This has always been considered as a hard problem, and it is the reason why the authors of [BN10] preferred the branch-and-bound approach to the dynamic programming one.

Having algorithms or tools to find the best (truncated) differential characteristics in the related-key setting for the AES is a first step for evaluating the resistance of this cipher against related-key differential-like attacks. The second step is to perform cryptanalysis, by applying well-known or new techniques. The most popular and efficient attacks against the AES in the related-key setting are boomerang attacks [BK09, GSW22, BL23]. These attacks can fully break both AES-192 and AES-256 but need at least 4, and sometimes much more, related keys to work. Very recently, a new type of attack, called the differential meet-in-the-middle cryptanalysis was proposed by Boura *et al.* [BDD+23]. This attack, that can be seen as a hybrid between a differential and a meet-in-the-middle (MITM) cryptanalysis, was applied against both SKINNY and AES-256 and produced a 12-round attack against AES-256 in the related-key setting requiring only 2 related keys. As this cryptanalysis technique is new and as it was demonstrated that it could be applied against the AES, it is essential to evaluate in-depth its threat against this important standard. Hence, our last contribution in this paper is the design of a new tool, based on the Mixed Integer Linear Programming (MILP) approach to automatically search for differential MITM attacks against all variants of the AES. The tool is complete in the sense that it searches for the distinguisher (by integrating our MILP model for the characteristic search) and the attack at the same time and outputs the configurations that optimize the total time complexity. This tool permitted us to find an attack on 13 rounds of AES-256 that uses only 2 related keys, improving thus the attack of [BDD+23] by one round and becoming the best attack against AES-256 in this more practical scenario where only 2 related keys are permitted. Moreover, we managed to scan automatically all differential MITM scenarios on AES-128 and AES-192 and show that there does not exist any related-key attack of this type against these two variants that performs better than other attacks. This permits to consolidate the hypothesis of [BDD+23] that this kind of cryptanalysis performs better when the key is much larger than the state.

The source codes of all our algorithms and models are available at:

https://github.com/pderbez/ToSC2023_3

The rest of the paper is organized as follows. Section 2 introduces some preliminary notions on the optimal truncated differential characteristics search for the AES. Then, Section 3 discusses known tool-based approaches for the optimal related-key characteristics search on the AES and introduces our new MILP model for this search. Section 4 describes our dynamic programming approach to solve this same problem. The running times of our algorithms, their comparison and a discussion are provided in Section 5. Finally, Section 6 presents our automated tool for differential MITM cryptanalysis as well as the attack on AES-256.

2 AES and Differential Analysis

In this section, we first provide the specifications of the AES and then discuss the difficulties regarding its differential analysis.

2.1 Description of the AES Encryption Scheme

The AES is a block cipher that processes data blocks of 128 bits. There are three versions of the AES, which differ in their key size (128, 192 or 256 bits) and their number of rounds

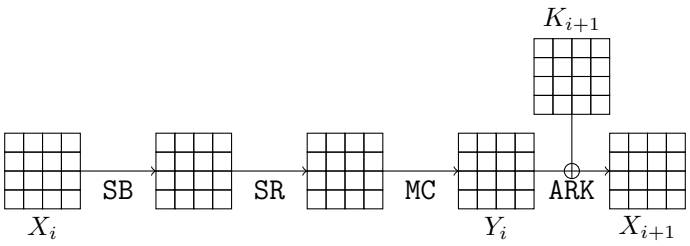


Figure 1: The AES round function. X_0 is obtained by XORing the input block and the subkey K_0. The output block is X_{N_r}. MC is omitted for the last round.

N_r (10 rounds for AES-128, 12 for AES-192 and 14 for AES-256). The block state can be viewed as a 4×4 array of bytes. The numbering of the state bytes we will use is as follows:

	0	1	2	3
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11
3	12	13	14	15

In some cases, when this is more convenient, we will also refer to a concrete byte of the state or the key by its row and column. For example, $X[2][1]$, will refer to the byte of the state X lying on the row 2 and column 1, where the row and column numbering is as indicated in the figure above.

After an initial subkey addition, the state is transformed by iterating a round function. This round function is the same for all versions of the AES. It is composed of four byte-oriented transformations as depicted in Figure 1. SubBytes (SB) applies to each byte of the state a non-linear bijection called S-box. ShiftRows (SR) rotates the second row of the state (resp. third and fourth row) by one byte (resp. two and three bytes) to the left. MixColumns (MC) multiplies each column of the state by an MDS (Maximum Distance Separable) matrix. Finally, AddRoundKey (ARK) XORs the state with the round subkey. For the final round, the MixColumns operation is omitted.

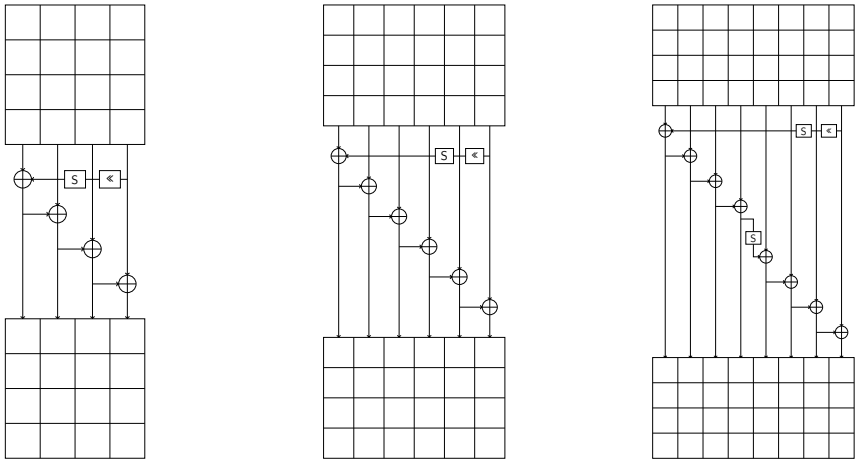
The round subkeys are derived from the master key. The master key is seen as a $4 \times N_k$ array of bytes, where $N_k = \ell/32$ for AES-ℓ. From it, the key expansion algorithm generates a $4 \times 4(N_r + 1)$ array of bytes W from which all the subkeys' columns are extracted. The first N_k columns of W are initialised with the master key's columns. Then, the other columns of W are computed, as depicted in Figure 2. The value of the k-th column w_k of W is computed using those of the columns w_{k-1} and w_{k-N_k}. More precisely, for $k \in [N_r, 4N_r + 3]$,

$$w_k = \begin{cases} w_{k-N_k} \oplus \mathsf{SubWord}\left(\mathsf{RotWord}\left(w_{k-1}\right)\right) \oplus \mathsf{RCon}(k/N_k) \text{ if } k \equiv 0 \mod N_k, \\ w_{k-N_k} \oplus \mathsf{SubWord}(w_{k-1}) \text{ if } N_k = 8 \text{ and } k \equiv 4 \mod 8, \\ w_{k-N_k} \oplus w_{k-1} \text{ otherwise,} \end{cases}$$

where $\mathsf{RCon}(k/N_k)$ is a round-dependent constant, RotWord performs an upward rotation of the column and SubWord applies the AES S-box to each byte of the column.

2.2 AES Differential Characteristics

Finding optimal related-key differential characteristics is a highly combinatorial problem that hardly scales. To limit this explosion, a common solution consists in using a truncated



(a) AES-128 **(b)** AES-192 **(c)** AES-256

Figure 2: Key schedules for the three versions of AES (figure from [FJP13]). The RotWord and SubWord functions are respectively denoted by « and S. The round-dependent constant addition is not represented.

representation [Knu95] of the characteristics' differences. In a truncated characteristic, each cell is abstracted by a Boolean variable that indicates whether this cell has a non-zero difference or not. In this case, the goal is no longer to find the exact input and output differences, but to find the positions of the cells having a non-zero difference. When a non-zero difference is present at the input of an S-box, we say that this S-box is *active*. The number of active S-boxes is an important quantity since, combined with the highest probability of a non-trivial transition through the S-box, it gives an upper bound on the probability of any differential characteristic matching the truncated pattern. The less active S-boxes there are, the higher the probability of a characteristic can be.

The optimal related-key differential characteristic problem is usually solved in two steps [BN10, FJP13, GMS16, GLMS20]. In the first one, we search for all truncated characteristics with a low number of active S-boxes. However, since some constraints at the byte level are relaxed when reasoning with a truncated representation, some truncated representations could be invalid in the sense that there do not exist actual byte values corresponding to the Boolean variables assignment. The second step aims at deciding whether each truncated characteristic is valid, and if it is, at finding the actual cell values that maximize its probability.

Pure truncated differential characteristic. Regarding the AES, we can model the propagation of the truncated differences over MixColumns, SubBytes and ShiftRows, using the fact that MixColumns uses an MDS matrix, that SubBytes is bijective and that ShiftRows is a reorganization of the bytes' positions of the state. For AddRoundKey, the XOR of two active bytes can give an active byte or an inactive byte. Similarly, we can use a simple model for the propagation through the key schedule. A *pure* truncated characteristic is a sequence of truncated differences that respect these propagation rules.

2.3 Invalid Truncated Differential Characteristics

The main issue with truncated differential characteristics is that some of them cannot be instantiated into actual characteristics. Among the latter, some can be directly avoided by exploiting linear equations induced by the round function and the key schedule.

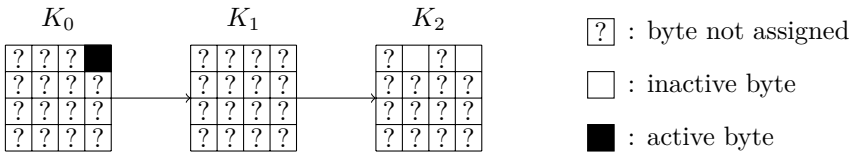


Figure 3: Truncated differential characteristic that does not respect all the linear constraints induced by the key schedule of `AES-128`.

Linear equations from the round function and the key schedule. The round function and the key schedule provide a set of linear equations between the bytes at the output of an S-box and the bytes of the actual differences X_r, K_r, $r \in [0, N_r]$. For instance, in the related-key setting, the equations for the `AES-128` key schedule are as follows (S denotes the S-box permutation):

- $K_{r+1}[i][j] = K_r[i][j] \oplus K_{r+1}[i][j-1], \ \forall r \in [0, N_r - 1], \forall i \in [0, 3], \forall j \in [1, 3],$
- $K_{r+1}[i][0] = K_r[i][0] \oplus S\left(K_r[(i+1) \mod 4][3]\right), \quad \forall r \in [0, N_r - 1], \forall i \in [0, 3].$

Definition 1. We say that a truncated differential characteristic (completely or partially defined) is *not consistent with respect to a set of linear equations* if it is possible to express, from this set of equations, a variable assigned to *active* as a linear combination of variables assigned to *inactive*.

Figure 3 gives an example of a partially defined truncated characteristic that is not consistent with respect to the linear equations induced by the key schedule of `AES-128`. Indeed, if we sum the equations

$$\begin{cases} K_1[0][3] = K_0[0][3] + K_1[0][2] \\ K_2[0][3] = K_1[0][3] + K_2[0][2] \\ K_2[0][2] = K_1[0][2] + K_2[0][1] \end{cases},$$

we obtain that $K_0[0][3] = K_2[0][1] + K_2[0][3]$. This equation does not hold in the example of Figure 3 as $K_0[0][3]$ is active, while both $K_2[0][1]$ and $K_2[0][3]$ are inactive.

Representing the system of equations with a matrix in row echelon form allows one to check the consistency of a partially or completely defined truncated differential characteristic. The matrix can be also used to deduce, from a partial assignment of variables, the values of other Boolean variables. As a toy example, the inconsistency of Figure 3 with respect to the above system of equations is detected with the following matrix:

$$\begin{array}{c} \begin{array}{cccccc} K_0[0][3] & K_1[0][3] & K_1[0][2] & K_2[0][2] & K_2[0][1] & K_2[0][3] \end{array} \\ \left(\begin{array}{cccc:cc} 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 \end{array} \right) \\ \qquad\qquad\qquad\qquad\qquad\qquad \underbrace{\qquad\qquad}_{\text{variables set inactive}} \end{array}$$

A particular case of those inconsistencies that happens often involves some equal columns of the subkeys and the underlying transitions through MixColumns and AddRoundKey. Figure 4 gives an example of such incompatibility.

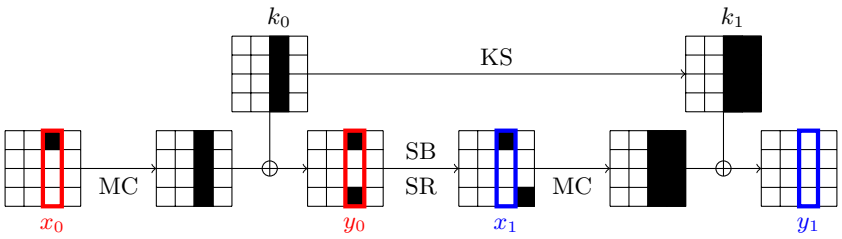


Figure 4: Example of a linear incompatibility. The key schedule imposes the subkeys' active columns k_0 and k_1 to be equal. On the other hand, we have that $\mathrm{MC}(x_0) \oplus k_0 = y_0$ and $\mathrm{MC}(x_1) \oplus k_1 = y_1$. It follows that $\mathrm{MC}(x_0 \oplus x_1) = y_0 \oplus y_1$, which contradicts the MDS property of MixColumns.

These linear incompatibilities related to the XOR operation present in the key schedule, but also in the AddRoundKey step and its interaction with the MixColumns operation are handled differently in the approaches encountered in the literature. Considering a large number of linear equations permits to avoid many invalid differential trails on the one hand but on the other hand heavily impacts the running time. Selecting the most relevant equations to limit as much as possible the number of invalid truncated characteristics is a difficult but also interesting problem that will be discussed in the next section.

3 Tool-based search of related-key differential characteristics for AES

Khoo *et al.* presented in [KLPS17] *human-readable* proofs for the number of minimal active S-boxes of AES-128 in the related-key setting. However, this approach, while very interesting, is tedious, difficulty scales for the largest variants of the AES and needs extra efforts to be applied to other ciphers. For this reason, tool-based approaches were developed to search for optimal related-key characteristics for the AES. We present in this section some of the most important tool-based approaches in this direction. Next, we present a new MILP model for this task and compare it to previous solutions.

3.1 Constraint programming (CP) approach [RGMS22]

Constraint programming (CP) is an algorithmic approach for solving combinatorial optimization problems. A CP model is composed of a set of variables and a set of constraints and the goal is to find an assignment of values to the variables such that all the constraints are satisfied. If the problem to solve is an optimization problem then the best solution, with respect to a concrete objective, is searched. For the modeling part, some specific language has to be used and to solve the problem one has to rely on one of the available solvers such as Gecode [Tea06], OR-Tools [PF], Picat [ZK16] or Choco [PFL16], to cite just a few.

In a recent work [RGMS22], Rouquette et al. proposed a new CP model to find the best optimal related-key differential characteristics for all versions of Rijndael. For this they adapted the strategy presented by Gérault et al. in 2020 [GLMS20] to solve the same problem. The solving procedure is done within two steps. During the first step, the solver searches for a truncated characteristic activating as few S-boxes as possible. The second step consists in trying to instantiate the truncated characteristic with concrete differences.

The main problem that occurs if a very simple model is used is that there will be too many truncated characteristics generated in the first step, that cannot be instantiated. The main reason this can occur is related to the way the XOR operation is modeled.

In a simple CP model, the XOR $a \oplus b \oplus c = 0$ between three variables is represented by the constraint

$$a + b + c \neq 1.$$

This simple way of modeling the XOR can lead to invalid truncated differential characteristics as shown in the following example, provided in [RGMS22]. Suppose we have the two XORs $a \oplus b \oplus c = 0$ and $b \oplus c \oplus d = 0$, modeled by the two constraints $a + b + c \neq 1$ and $b + c + d \neq 1$. Suppose now that a is an inactive word while d is active. It is then easy to see that it is possible to satisfy this model at the truncated level by setting $a = 0, b = 1, c = 1$ and $d = 1$. However, it is not possible to instantiate it with actual differences as $a = 0$ means that $b = c$ and thus we must have $d = 0$.

What the authors of [GLMS20] and [RGMS22] proposed to do to solve this issue is to XOR any two equations of this type and create a new constraint. In the previous example, this would have given the equation $a \oplus d = 0$ and the additional constraint $a + d \neq 1$ and this would have been enough to solve the problem of this example. Therefore, the approach used was to generate all basic constraints for the key schedule equations and then recursively XOR all these equations between them at the moment they contained at most four variables.

A second solution that was used in [GLMS20] and refined in [RGMS22] was to introduce extra Boolean variables, called *diff* variables, to indicate whether two bytes a and b are equal. More precisely, for two bytes a and b, a Boolean variable $\delta_{a,b}$ is set to 0 if the value of the bytes a and b is the same, and is set to 1 otherwise. Such variables were associated to all variables involved in the key schedule, the AddRoundKey and the MixColumns step.

The associated models generated from the basic `AES` equations with the use of the two above tricks were shown to be quite efficient to limit the number of invalid truncated trails but came to the cost of a high number of extra variables and constraints.

3.2 MILP model of [DEFN22]

Instead of adding a quadratic (in the number of S-boxes and round-key bytes) number of Boolean variables indicating whether two differences involved in a characteristic of `AES` are equal or not, the authors of [DEFN22] chose to directly deal with linear algebra. More precisely, they presented a new MILP model to directly search for boomerang attacks on `AES`, integrating a search for truncated related-key differential characteristics as well. For this particular part of their algorithm, they proposed to use a very simple model, containing only the description of the operations composing the round function, without adding any extra constraint to prevent invalid trails. To deal with them, they instead relied on the callback functionality of Gurobi which allows to run some extra pieces of code and to add new constraints during the solving process. Hence, each time a solution is found by the solver, the program checks the presence of any linear inconsistency and, whenever it finds one, a constraint is added to avoid the inconsistency for the upcoming solutions.

This method could permit to converge to an exact solution at the end, but the size of the model and thus the running time exploded.

3.3 Our MILP model

For the reasons stated above, the model of [DEFN22] was very slow and the authors had to restrict themselves to a very small portion of the search space to obtain a result in a reasonable time. We thus propose in this work a new MILP model that we believe is the right trade-off between the approaches of both [DEFN22] and [RGMS22].

Our MILP model starts from the model of [DEFN22] and encodes, as there, all basic operations composing the `AES` into linear inequalities. We then add two types of constraints.

First, it is well-known that, by combining key schedule equations, we can derive new equations involving only 3 subkey bytes, located on non-consecutive rounds. For instance, on `AES-128`, we know that $K_r[i-2] \oplus K_r[i] \oplus K_{r-2}[i] = 0$ holds whenever the index i corresponds to a byte on either the third or the fourth column of a round key. We included in our model the inequalities corresponding to all (linear combinations of) key schedule equations involving at most 3 key bytes. The second type of extra constraints we added to the model is directly inspired from [GLMS20] and [RGMS22]. These constraints target the type of inconsistencies presented in Figure 4 and they are obtained as follows. From a pair of triplets of columns $((x_0, k_0, y_0), (x_1, k_1, y_1))$ satisfying $y_0 = \mathrm{MC}(x_0) \oplus k_0$ and $y_1 = \mathrm{MC}(x_1) \oplus k_1$, we obtain by linearity the equation $y_0 \oplus y_1 \oplus k_0 \oplus k_1 = \mathrm{MC}(x_0 \oplus x_1)$. The MDS property should hold as well on the pair $(u, v) := (x_0 \oplus x_1, y_0 \oplus y_1 \oplus k_0 \oplus k_1)$. While the authors of [GLMS20] and [RGMS22] added such a constraint on all pairs $((x_0, k_0, y_0), (x_1, k_1, y_1))$ and introduced *diff* variables to be as precise as possible, we only add this constraint when the sum $k_0 \oplus k_1$ is equal, by definition of the key schedule, to another subkey column k_2 (see the calls to `addMixColumnsExtraConstr` on lines 12, 13 and 14 of Algorithm 1).

Algorithm 1: MILP model for `AES`

input : Target number of rounds `R` and `version` of `AES`
output : Minimum number of active S-boxes for `R` rounds for `AES-version`

```
1  Initialize a model m
2  addKeyScheduleBasicConstr(m, R, version)
3  addShitfRowsMixColumnsBasicConstr(m, R)
4  addAddRoundKeyBasicConstr(m, R)
5  addKeyScheduleExtraConstr(m, R, version)
6  if version = 128 then w ← 4
7  else if version = 192 then w ← 6
8  else w ← 8
9  for k = 0...4R - 1 do
10     if (version = 256 and k mod 4 ≠ 0 )
11     or (version ≠ 256 and k mod w ≠ 0 ) then
12         addMixColumnsExtraConst(m, k, k-1, k-w)
13         addMixColumnsExtraConst(m, k-1, k-w, k)
14         addMixColumnsExtraConst(m, k-w, k, k-1)
15 ▷ The variable obj contains the number of active S-boxes
16 obj ← getObjectiveFunction(R, version)
17 m.addConstr(obj ≥ 1)
18 Minimize obj.
```

We found this to be the right trade-off between the efficiency of the model and the number of false characteristics removed and confirmed our choice by experiments. Actually, we performed several experiments to understand the contribution of every extra constraint we added to the model. Adding the constraints related to the sparse key schedule equations is highly beneficial for both `AES-128` and `AES-192` and has almost no effect on the 256-bit version. Regarding the XOR of two columns, since the associated constraint is quite costly in terms of both extra variables and inequalities, its effect is fluctuating, depending on the version and the number of rounds. We observed that for `AES-128` it is always better to add the constraints while for `AES-256` it never improves the solving time. As a consequence, we decided to apply it only on consecutive rounds and found this setting to be the most efficient in most of the cases.

Our model is summarized in Algorithm 1. The basic constraints and the additional

Algorithm 2: `addMixColumnsExtraConstr(m, k0, k1, k2)`

▷ The k_2-th column of the subkeys is the XOR of the k_0-th and k_1-th column
for $i = 0 \dots 2$ **do**
  $r_i \leftarrow \lfloor k_i/4 \rfloor$ // round of the k_i-th column
  $c_i \leftarrow k_i \mod 4$ // index of the k_i-th column
if $r_0 < 1$ ***or*** $r_1 < 1$ ***or*** $r_2 < 1$ **then**
  return // a round index is out of range
Let `u[0], u[1], u[2], u[3]` be dummy binary variables
Let `v[0], v[1], v[2], v[3]` be dummy binary variables
for $i = 0 \dots 3$ **do**
  `addXorConstr(m,u[i],` $X_{r_0-1}[SR^{-1}(c_0+4i)]$`,` $X_{r_1-1}[SR^{-1}(c_1+4i)]$`)`
  `addXorConstr(m, v[i],` $K_{r_2}[c_2+4i]$`,` $X_{r_0}[c_0+4i]$`,` $X_{r_1}[c_1+4i]$`)`
$e \leftarrow 0$
for $i = 0 \dots 3$ **do**
  $e \leftarrow e + u[i] + v[i]$
Let `f` be a dummy binary variable
`m.addConstr(e` $\leq$ `8f)`
`m.addConstr(e` $\geq$ `5f)`

constraints for the key schedule are given in Appendix A for `AES-128`. The constraints for the two other versions of `AES` are similar. We also give in Appendix B a comparison of the number of variables and constraints between our model and the model of [GLMS20] for `AES-128`. Note that, as in [DEFN22], we also checked *a posteriori* the consistency of the linear equations that are not taken into account in our model by using the callback functionality of Gurobi.

4 Dynamic Programming for Differential Bounds on AES

We present now a completely different approach to search for differential characteristics and bounds for the `AES`. This approach is based on a dynamic programming procedure and does not use any generic tool. In 2013, Fouque *et al.* [FJP13] used such an algorithm to study this problem for `AES-128`. However, their algorithm, while efficient for `AES-128`, was very memory consuming and their approach could not be extended to the larger variants of the `AES`. To study `AES-192` and `AES-256` with a dynamic programming procedure we need to think of a low-memory approach by choosing a compact representation of the differences. The approach we chose was to only specify the number of actives bytes per column. The second problem to tackle is that the dynamic programming procedure does not propagate linear constraints through several rounds. We integrate these constraints in a second step, while generating the characteristics with a tree traversal approach. During the characteristics' generation, we use the information obtained during the dynamic programming phase to bound from below the number of active S-boxes of the characteristics being formed. In the following, we rephrase the computational principle behind the generic dynamic programming algorithm used in [FJP13] and describe how we applied it to study the related-key differential characteristics of all versions of the `AES`.

4.1 Principle of Dynamic Programming

To describe the algorithm, we denote by f the SPN cipher under consideration. If we choose to work in the related-key scenario, the function f must include the key schedule. In other

words, its domain D must be the Cartesian product of the key space and the internal state space. The function f is seen as the composition of n simple functions: $f = f_n \circ \cdots \circ f_1$. Typically, the f_i's are some sub-functions of the round function or the key schedule. The algorithm allows one to compute for each step $1 \leq i \leq n$ and each difference $d \in D$ a lower bound on the number of active S-boxes of a valid characteristic over $f_i \circ \cdots \circ f_1$ ending with the difference d. The algorithm relies on a compact representation of the differences. The truncated differences are an example of such compact representation. We denote by *comp* the function that associates to an exact difference its compact representation. For each choice of compact representation, a set of propagation rules must be deduced. More precisely, we say that a compact difference $a \in comp(D)$ is compatible over f_i with a compact difference b if there exists an exact and valid differential $(d_a, d_b) \in D^2$ over f_i such that $a = comp(d_a)$ and $b = comp(d_b)$. For each step function f_i and each compact difference $a \in comp(D)$, we denote by $succ_{f_i}(a)$ the set containing all the compact differences that are compatible with a over f_i. We denote by $prec_{f_i}(a)$ the set defined as $prec_{f_i}(a) \coloneqq \{x \in comp(D) : a \in succ_{f_i}(x)\}$. The goal of the dynamic programming is to fill a 2-dimensional array T in such a way that the cell T$[i][a]$ contains the minimum number of active S-boxes of a compact trail over $f_i \circ \ldots \circ f_1$ ending with a. Algorithm 3 starts by initializing the whole sub-table $T[0]$ to 0. Then, for $1 \leq i \leq n$, the sub-table $T[i]$ is computed from the sub-table $T[i-1]$. If we denote by $c_{f_i}(a, b)$ the minimum number of S-boxes that are activated to reach the compact difference b from the compact difference a at step i, we have that

$$T[i][b] = \min_{a \in prec_{f_i}(b)} \left(\mathrm{T}[i-1][a] + c_{f_i}(a, b)\right).$$

The algorithm has a time complexity of about $\sum_{i=1}^{n} \sum_{a \in comp(D)} |succ_{f_i}(a)|$ updates of cells but note that it is sometimes possible to reduce this complexity by replacing the loop of line 6 with a more efficient procedure exploiting some properties of the propagation rules as we will show below. The memory complexity of the algorithm corresponds to the storage of the n sub-tables of size $|comp(D)|$. Note that in fact, since only 2 sub-tables are needed for the computational process, one can choose to only keep some of the sub-tables. We do not include in the memory complexity the storage of the sets $succ_{f_i}(a)$, contrary to what is done in [FJP13]. Indeed, we consider that the function f can be divided into sub-functions simple enough to directly compute the sets $succ_{f_i}(a)$ in the loop of line 7.

Algorithm 3: Bounds on the number of active S-boxes

output : An array T such that for an exact difference d, T$[i][comp(d)]$ is a lower bound on the number of active S-boxes of a valid characteristic over $f_i \circ \ldots \circ f_1$ ending with the difference d.

1 **forall** $a \in comp(\mathcal{D})$ **do**
2 T$[0][a] \leftarrow 0$
3 **forall** $i \in [1, n]$ **do**
4 **forall** $b \in comp(\mathcal{D})$ **do**
5 T$[i][b] \leftarrow \infty$
6 **forall** $a \in comp(\mathcal{D})$ **do**
7 **forall** $b \in succ_{f_i}(a)$ **do**
8 T$[i][b] \leftarrow \min\left(\mathrm{T}[i-1][a] + c_{f_i}(a, b),\ \mathrm{T}[i][b]\right)$
9 **return** T

4.2 Our Variant Dedicated to the AES

Algorithm 3 requires to store the sub-tables $\mathrm{T}[i]$ of size equal to the cardinal of $comp(\mathbb{F}_2^{128} \times \mathbb{F}_2^{\ell})$, where ℓ is the key size. We have two opposite objectives when defining the compression function *comp*: keeping enough information to get a meaningful bound while minimizing the size of the arrays so they can be stored. The first approach we tried was to specify only 3 of the 4 rows of the state and the subkeys, but the bounds obtained were not precise enough. For this reason, we finally decided to indicate only the number of active bytes per column, for both the block state and the subkeys. We call this type of compact differences the *compressed differences* (see Figure 5 for an example). The authors of [FJP13] had already introduced this compact vision of the block state but did not exploit its full potential to decrease the memory and the time complexity. The rationale behind this choice lies in the property of the matrix used to define the MixColumns transformation. As the matrix is MDS (Maximum Distance Separable), we only need to know the number of active bytes per column to propagate the truncated differences through MixColumns. The choice of the compact representation of the subkeys is motivated by the fact that the related-key differential characteristics with few active S-boxes seem to have their subkeys structured by columns, with mostly columns that are fully active or fully inactive.

Figure 5: A truncated state difference and its associated compressed difference.

The propagation rules for the compressed differences are summarized below.

SubBytes, SubWord and RotWord. Since the AES S-box is bijective, SubBytes and SubWord have no effect on the active/inactive bytes. Since RotWord rotates the bytes of a column, it does not affect the number of active bytes per column. Thus, over these three maps, a compressed difference is only compatible with itself.

MixColumns. We use the MDS property of the MixColumns transformation: the sum of the number of active bytes of a column before MC and after MC is equal to 0 or strictly greater than 4.

XOR of two columns for the key schedule and AddRoundKey. The XOR between a column with x active bytes and a column with x' active bytes gives a column with $y \in [\ |x - x'|\ ,\ \min(4, x + x')]$ active bytes.

ShiftRows Let $x = (x_0, x_1, x_2, x_3) \in [0,\, 4]^4$ be a compressed state. The set $succ_{\mathrm{SR}}(x)$ contains all the compressed differences $y = (y_0, y_1, y_2, y_3) \in [0,\, 4]^4$ such that y can be expressed as the sum of 4 vectors of $\{0, 1\}^4$ of respective Hamming weights x_0, x_1, x_2 and x_3. Note that $succ_{\mathrm{SR}}(x) = prec_{\mathrm{SR}}(x)$. For example,

$$succ_{\mathrm{SR}}\left((3,1,0,0)\right) = \left\{ \begin{array}{l} (2,1,1,0), (1,2,1,0), (1,1,2,0), (2,1,0,1), (1,2,0,1), \\ (2,0,1,1), (1,1,1,1), (0,2,1,1), (1,0,2,1), (0,1,2,1), \\ (1,1,0,2), (1,0,1,2), (0,1,1,2) \end{array} \right\}.$$

Working with compact differences relaxes the propagation rules for ShiftRows, RotWord and for the XOR operation. As a consequence, there exist compressed characteristics that do not correspond to any pure truncated characteristic (see Figure 6 for an example).

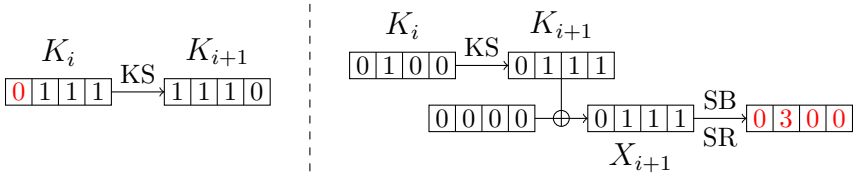


Figure 6: Two examples of inconsistent compressed characteristics. For the left characteristic, if we assume that the first three columns of K_{i+1} are valid, then its last column should have 2 active bytes. For the right characteristic, we know from the key schedule and from the fact that $X_{i+1} = K_{i+1}$, that the active bytes of X_{i+1} are on the same row. Thus, the transition over SR is inconsistent.

Complexity of the dynamic programming phase. To apply Algorithm 3, we decompose the AES with the above operations. For MixColumns, AddRoundKey and the computations of the subkeys, we update one column after the other and store the results after each update. Given that the XOR between two compressed columns gives 2.6 compatible columns on average, computing a new subkey requires in total $4 \times 2.6 \times 5^{4+N_k} \approx 2^{12.65+2.32 \cdot N_k}$ cell updates. For the propagation over ShiftRows, MixColumns and AddRoundKey, we reduce the time complexity thanks to some specific properties of the propagation rules.

The improvement for ShiftRows relies on the fact that the sub-table storing the results can be partitioned into groups of cells that necessarily share the same value. The existence of these groups comes from the following property: for any compressed block state $x' \in [0, 4]^4$ obtained by permuting the columns of a compressed state x, we have that $prec_{\mathrm{SR}}(x') = prec_{\mathrm{SR}}(x)$. For each of these groups, it is thus sufficient to compute the shared value once and copy it in all the cells. While the naive approach needs 16145×5^{N_k} operations in terms of cell updates, this approach requires only 2433×5^{N_k} operations (88.8% of which are an affectation of a shared value).

For MixColumns, the sub-table $\mathrm{T}[i_{\mathrm{MC}}]$ that stores the results, can be filled with less operations than $\sum_a |succ_{MC}(a)|$. For this, we take advantage of relations of the form $prec_{\mathrm{MC}}(b) = prec_{\mathrm{MC}}(b') \cup \{a\}$, where $b, b', a \in [0, 4]^{4+N_k}$. There is indeed, at the compressed column level, an inductive formula to define the sets $prec_{\mathrm{MC}}(c)$, where $c \in [0, 4]$ is a compressed column:

$$prec_{\mathrm{MC}}(c) = \begin{cases} \{0\}, & \text{if } c = 0, \\ \{4\}, & \text{if } c = 1, \\ prec_{\mathrm{MC}}(c-1) \cup \{5-c\}, & \text{if } 2 \leq c \leq 4. \end{cases}$$

This type of relation is interesting since if $\mathrm{T}[i_{\mathrm{MC}}][b']$ has already been computed, the value $\mathrm{T}[i_{\mathrm{MC}}][b]$ can be obtained in 1 cell update:

$$\mathrm{T}[i_{\mathrm{MC}}][b] = \min(\mathrm{T}[i_{\mathrm{MC}}][b'], \mathrm{T}[i_{\mathrm{MC}} - 1][a]).$$

In fact, if we compose the column multiplication of MixColumns with the XOR of a subkey column from AddRoundKey, we still have this type of inductive formula. We thus merge the steps MixColumns and AddRoundKey. In total for these two steps, we have a time complexity of $5.6 \times 5^{4+N_k}$ cell updates while the naive method would require $19.2 \times 5^{4+Nk}$ operations.

The following table summarizes both the time and memory complexities of the function $\texttt{ProgDynCol}(\ell, r)$ that applies Algorithm 3 on r rounds of AES-ℓ. For each round we store the array after the ShiftRows operation. Then, for AES-128, we also store the array after each update of a column for both the key schedule and the ARK $\circ$ MC operations and thus we need to store 9 tables per round. For the two larger versions, in order to keep the

Table 1: Time and memory complexity of `ProgDynCol`(ℓ, r). The time complexity is given in the number of cell updates.

	Time complexity	Memory (Bytes)
`AES-128`	$r \times 2^{22.89}$	$(9r-9) \times 2^{18.58}$
`AES-192`	$r \times 2^{27.53}$	$(3r-3) \times 2^{23.22}$
`AES-256`	$r \times 2^{32.18}$	$(3r-4) \times 2^{27.86}$

memory low enough so that the program can be run on a personal computer, we store the array only after updating the next round-key and after performing the ARK $\circ$ MC operation on the four columns. Hence, for these versions we only store 3 tables per round. Note that some tables can be omitted: there is no need to perform the first and last ShiftRows operations and for `AES-256` the second round-key is part of the master key.

4.3 Reconstructing Truncated Characteristics

First, the procedure to find an optimal truncated differential characteristic calls `ProgDynCol` to compute an array T that stores lower bounds on the number of active S-boxes necessary to reach a given compressed difference. Then, it calls `FindCharacteristic` to search for an r-round truncated differential characteristic with n_s active S-boxes, starting with $n_s = 0$. The variable n_s is incremented until a characteristic is found. The function `FindCharacteristic` makes use of the array T to search for compressed characteristics with n_s active S-boxes. During this search, it integrates linear constraints to detect some invalid compressed characteristics. The compressed characteristics are then turned if possible into valid truncated characteristics, that respect all the linear constraints.

The function `FindCharacteristic` forms the compressed characteristics in the backward direction and with a recursive depth-first search approach. The last sub-array that is computed by `ProgDynCol` – denoted $\mathrm{T}[i_{\max}]$ – gives the starting points of the tree traversal; and the other sub-arrays give data to prune the tree. We begin a characteristic (in the backward direction) with a compressed difference $d_{i_{\max}} \in [0, 4]^{4+N_k}$ such that $\mathrm{T}[i_{\max}][d_{i_{\max}}] \leq n_s$. Before extending a compressed characteristic $d_i \to \cdots \to d_{i_{\max}}$, we always check that the sum between $\mathrm{T}[i][d_i]$ and the number of active S-boxes of the characteristic $d_i \to \cdots \to d_{i_{\max}}$ is below n_s. We also check that no linear inconsistency is detected.

To detect a linear inconsistency, the function `FindCharacteristic` keeps track of information at the truncated byte level. For that, each compressed characteristic is paired with a partially defined truncated characteristic, that locates all the bytes of the compressed characteristic which are known to be active or inactive. This partially defined truncated characteristic is used, along with a system of linear equations, to detect a linear inconsistency and to notice when other bytes must be set active or inactive. When we extend a compressed characteristic, we keep the partially defined truncated characteristic up-to-date. For example, if a compressed difference has a column with k active bytes and if there are already k active bytes in the associated column of the truncated characteristic, the other $4-k$ bytes of the column are set inactive. An inconsistency is detected when the newly deduced bytes of the truncated characteristic are in contradiction with the compressed characteristic or when the truncated characteristic is itself inconsistent. Since the best truncated differential characteristics are very structured, with many columns fully active or inactive, we detect most of the inconsistencies. Still, some of them cannot be detected with this method as it relies on the knowledge of the exact positions of the active and inactive bytes. Thus, in addition to the previous verification, we also detect, at the compressed column level, some frequent transitions over MixColumns and AddRoundKey

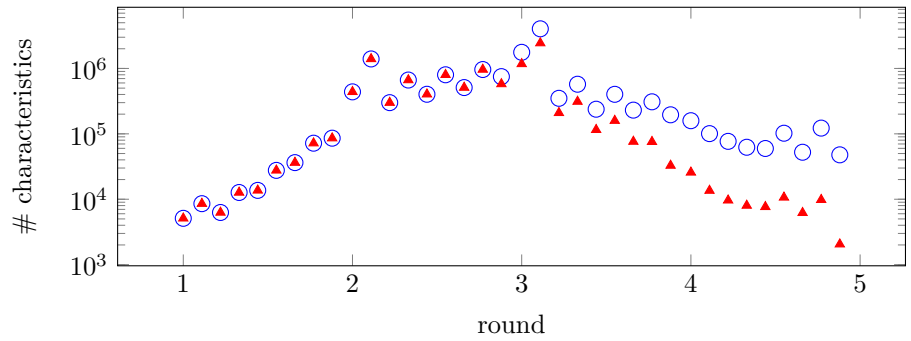


Figure 7: Number of compressed characteristics reaching a given length when calling `FindCharacteristic` with $n_s = 17$, $\ell = 128$ and $r = 5$. Each of the five rounds is divided in 9 steps. We distinguish between the case in which only the verification at the truncated byte level is performed (symbol ○) and the case in which both verifications are used (symbol ▲).

which contradict the MDS property of the MixColumns matrix, exactly as we did in our MILP model (see Section 3.3). For each pair of triplets of columns $((x_0, k_0, y_0), (x_1, k_1, y_1))$ verifying $y_0 = \mathrm{MC}(x_0) \oplus k_0$ and $y_1 = \mathrm{MC}(x_1) \oplus k_1$ and such that the sum $k_0 \oplus k_1$ is equal to another subkey column k_2, we check whether the sum of the number of active bytes of $x_0 \oplus x_1$ and of $y_0 \oplus y_1 \oplus k_2$ can be equal to 0 or strictly greater than 4 in our compressed characteristics. For that, we use the compressed characteristic and its associated partially defined truncated characteristic to bound from above the number of active bytes of $x_0 \oplus x_1$ and of $y_0 \oplus y_1 \oplus k_2$. If the sum of these two bounds is equal to or less than 4, we should have $x_0 \oplus x_1 = 0$ and $y_0 \oplus y_1 \oplus k_2 = 0$. If these equalities cannot be verified, we find an inconsistency. Otherwise, we use them to refine the partially defined truncated characteristic.

For `AES-128`, adding the verification at the compressed column level increases, after a few rounds, the number of invalid compressed characteristics that are detected at each step of the tree traversal. For example, Figure 7 gives the number of characteristics that need to be extended, with and without this verification, to find the optimal 5-round truncated characteristics for `AES-128`. For `AES-256`, the verification at the truncated byte level is sufficient to have a very low number of compressed characteristics to extend (less than 500 and often just a dozen).

Algorithm 4: Computation of an optimal truncated differential characteristic

input : The size ℓ of the key and the number r of rounds
output : An optimal truncated differential characteristic t and its number n_s of active S-boxes

```
T ← ProgDynCol(ℓ, r)
n_s ← 0
t ← null
while t = null do
    t ← FindCharacteristic(n_s, T, ℓ, r)
    if t = null then n_s ← n_s + 1
return t, n_s
```

5 Running Times and Discussion

We provide in this section the running times of our MILP model presented in Section 3.3 and our dynamic programming algorithm given in Section 4 applied to all three versions of the `AES`. The results are summarized in Table 2. The third column of the table shows the minimum number of active S-boxes that both programs computed for different number of rounds. The bounds we obtained are the same as those given by Rouquette *et al.* in [RGMS22]. The fourth column contains the number of optimal truncated differential characteristics found by all approaches. We give in the sixth column the timings we obtained by running our MILP model and in the seventh column the timings of our dynamic programming algorithm. All computations were done on a personal computer equipped with a 8-core Ryzen 3700X processor, running at 3.6 GHz and having 32 GB of RAM. The first measurement shown in the last two columns corresponds to the real time, that is the running time it took to the program to terminate when all 8 cores were used. The second measurement given in the parenthesis corresponds to the user time, that is, the time when a single core is used for the computation. Finally, the fifth column corresponds to the timings given in [RGMS22] to compute the same bounds. Examples of optimal truncated differential characteristics for `AES-192` and `AES-256` are provided in Appendix C.

As it can be seen from this table, our ad-hoc algorithm is very competitive and almost always faster than both the CP-based approach of [RGMS22] and our MILP model. Besides the simplicity of our solution and its comprehensible complexity analysis it offers, it has further the advantage to be easily parallelizable. This is not the case of the CP-approach of [RGMS22] and this is the reason the bounds were computed on a single core. Note that the authors of [RGMS22] might have obtained better results with recent CP solvers which support multi-threading such as the OR-tool. However, our opinion is that the only way for such a solver to benefit from parallelism would be to rely on an *embarrassingly parallel search* and it is unclear whether it would have been easy to balance the workload.

The running times of the three approaches all are practical and the interest of our new ad-hoc algorithm based on dynamic programming does not lie in the few minutes saved to compute the differential characteristics. We believe that its main advantage is to give more insight on the real complexity of the problem of searching for the best related-key differential characteristic on `AES` and to show that the search space is much smaller than one could have expected.

`AES-192` We noticed that our algorithm is much slower than both the CP and MILP approaches for 9 rounds of `AES-192`. The reason comes from the high linearity of the key schedule for this version of `AES`, and more precisely from the existence of sparse relations between round-key bytes separated by many rounds. For instance, for the right values of i and r, the relation $K_r[i] \oplus K_r[i-4]$ computes a byte on a round key located 6 rounds before. As when we reconstruct the characteristics we work locally, handling the round operations one by one, such long key-bridging have a disastrous effect on the efficiency of our algorithm. To overcome this issue, a solution could have been to recompute the bounds while reconstructing the characteristics once enough key bytes have been fixed.

To show the importance of such sparse relations in the efficiency of the different approaches, we removed the corresponding constraints from our MILP model and we observed that, in most cases, solving the model became much slower. For instance, still for 9 rounds of `AES-192`, without the constraints, the solver took 98 minutes to solve this problem instead of only 5 minutes. This also explains why the modelisation of [RGMS22], based on extra variables indicating whether two variables are equal, is particularly efficient on this version of the `AES`.

Table 2: Running times of our MILP model and dynamic programming algorithm on all versions of the AES and comparison with the timings obtained by the CP-based approach of [RGMS22]. The third column contains the bounds on the minimal number of active S-boxes, while the fourth column provides the number of optimal truncated differential characteristics found.

Algorithm	R	Min nb of active S-boxes	# char.	CP [RGMS22] Time	MILP Real Time (User Time)	Dynam. Prog. Real Time (User Time)
AES-128	3	5	2	13s	1s (1s)	1s (1s)
	4	12	1	31s	9s (36s)	1s (1s)
	5	17	81	2h24m	26s (2m22s)	40s (5m6s)
AES-192	3	1	14	1s	1s (1s)	1s (2s)
	4	4	3	6s	2s (3s)	1s (4s)
	5	5	2	8s	1s (3s)	1s (5s)
	6	10	3	17s	10s (34s)	1s (8s)
	7	14	2	46s	1m (4m26s)	1s (9s)
	8	18	4	1m23s	1m38s (8m3s)	1m35s (12m37s)
	9	24	6	30m	5m33s (35m18s)	4d5h (20d4h)
AES-256	3	1	33	1s	1s (1s)	8s (46s)
	4	3	10	3s	1s (1s)	12s (1m10s)
	5	3	4	5s	1s (2s)	16s (1m39s)
	6	5	3	13s	3s (5s)	19s (1m57s)
	7	5	1	18s	3s (5s)	23s (2m21s)
	8	10	2	32s	8s (24s)	29s (3m1s)
	9	15	8	5m46s	23s (1m31s)	32s (3m24s)
	10	16	4	2m39s	2m19s (8m59s)	34s (3m31s)
	11	20	4	5m30s	3m20s (15m35s)	42s (4m30s)
	12	20	4	4m37s	6m31s (37m24s)	42s (4m16s)
	13	24	4	7m	23m16 (160m58s)	52s (5m24s)
	14	24	4	9m17s	32m27s (124m28s)	50s (5m5s)

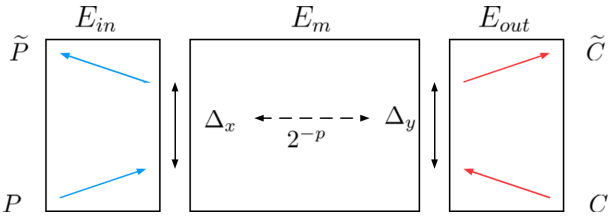


Figure 8: The differential meet-in-the-middle framework

6 Differential Meet-in-the-Middle Cryptanalysis of AES-256

In a recent paper, Boura, David, Derbez, Leander and Naya-Plasencia presented a new cryptanalysis technique, called the differential meet-in-the-middle attack [BDD+23]. This cryptanalysis method can be seen as either a way to extend by more rounds the middle part of a meet-in-the-middle attack, or as an alternative way to do the key recovery in a differential attack. In this paper, the authors provided two applications of their technique. First, a single-key attack against SKINNY-128-384 and then a related-key attack against 12-round AES-256. The interest of the latter attack is that it needs only 2 related keys, while the boomerang attack of Biryukov and Khovratovich that can break the full AES-256 in the related-key setting [BK09], necessitates 4 related keys.

One of the interests of this newly introduced technique is that it permits to exploit differential distinguishers in a different way than what is typically done in classical differential attacks. However, the authors did not provide an automated algorithm to search for these attacks and such a tool would be very helpful to evaluate the resistance of a known or newly developed cipher against this cryptanalysis technique.

To fill this gap, we describe in this section a MILP-based tool to automatically search for differential meet-in-the-middle attacks on AES. This tool permitted us to find a new 13-round attack against AES-256 involving only two related keys.

We first start by briefly describing the attack framework as given in [BDD+23]. Then in Section 6.2 we provide our tool for differential meet-in-the-middle attacks and describe our attack on AES-256 in Section 6.3.

6.1 Differential Meet-in-the-Middle Cryptanalysis Framework

Let E be a block cipher divided into three parts E_{in}, E_m and E_{out} such that $E = E_{out} \circ E_m \circ E_{in}$. Suppose now there exists an r-round differential $\Delta_x \to \Delta_y$ of probability 2^{-p} for the middle part E_m, as depicted in Figure 8. The attacker starts from a random plaintext P and its associated ciphertext C and tries to generate a second plaintext-ciphertext pair $(\widetilde{P}, \widetilde{C})$ such that together they satisfy the differential $\Delta_x \to \Delta_y$ in the middle part E_m:

$$E_{in}(P) \oplus E_{in}(\widetilde{P}) = \Delta_x \quad \text{and} \quad E_{out}^{-1}(C) \oplus E_{out}^{-1}(\widetilde{C}) = \Delta_y.$$

The idea now of the framework is to generate $\widetilde{P}, \widetilde{C}$ using a MITM approach. For this, the attacker treats the parts E_{in} and E_{out} independently and searches, in a second step, a match between those parts, by checking whether $E(\widetilde{P}) = \widetilde{C}$. A successful match indicates a potentially correct guess for the secret key. The detailed procedure is as follows:

Part E_{in}. Given P, the attacker guesses the minimal amount of key information k_{in}, that permits her to compute an associated plaintext $\widetilde{P}$ such that $E_{in}(P) \oplus E_{in}(\widetilde{P}) = \Delta_x$ if the guess of k_{in} is correct. For each guess i for k_{in}, the attacker obtains a different plaintext candidate $\widetilde{P}^i$ and for any of these $2^{|k_{in}|}$ plaintext candidates, she asks the encryption

oracle to generate the corresponding ciphertext $\widehat{C}^i = E(\widetilde{P}^i)$. She stores $\widehat{C}^i$ together with i in a hash table.

Part E_{out}. In a similar way, given C, the attacker will guess some key material k_{out} that will permit her to generate a new ciphertext $\widetilde{C}$ that satisfies $E_{out}^{-1}(C) \oplus E_{out}^{-1}(\widetilde{C}) = \Delta_y$. More precisely for each guess j for k_{out}, the attacker computes a candidate ciphertext $\widetilde{C}^j$.

Match and complexities. If the differential transition $\Delta_x \to \Delta_y$ happened with probability 1, then the attacker would have to search among the $\widetilde{C}^j$ ciphertext candidates one that would be present in the hash table. This would indicate that the guess (i, j) for the key material (k_{in}, k_{out}) is correct. However, as the transition has in general a much lower probability 2^{-p}, the attack has to be repeated 2^p times with different random plaintext-ciphertext pairs (P, C) in order to hope for a match.

If k is the size of the key, the time complexity in the expected case where $|k_{in} \cup k_{out}| \geq n - p$ is computed as

$$\mathcal{T} = 2^p \times (2^{|k_{in}|} + 2^{|k_{out}|}) + 2^{|k_{in} \cup k_{out}| - n + p} + 2^{k-n+p}.$$

The data complexity of this attack can be estimated as

$$\mathcal{D} = \min\left(2^n, 2^{p+\min(|k_{in}|,|k_{out}|)}\right).$$

Finally, the memory complexity is given by $\mathcal{M} = 2^{\min(|k_{in}|,|k_{out}|)}$. However, this complexity can be improved to $2^{\min(|k_{in}|-|k_{in}\cap k_{out}|,|k_{out}|-|k_{in}\cap k_{out}|)}$ by first guessing the common key material before running the attack.

6.2 MILP-based Tool to Search for Differential MITM Attacks

Searching for the best differential meet-in-the-middle attacks against `AES` requires to search for the right trade-off between the length of the distinguisher, its probability, the number of rounds that can be appended around the distinguisher and the entropy of the key material involved in the key-recovery process. As such, and since many recent cryptanalysis tools are based on, we decided to extend our MILP model from Section 3.3 into a new MILP model dedicated to differential meet-in-the-middle cryptanalysis.

Outer parts. Extending our model to find attacks and not only distinguishers only requires to handle the outer parts within the model by adding extra constraints as well as modifying the objective since we now want to minimize the overall complexity of the attack. In those parts, the differences are propagated with probability 1 to the plaintext and to the ciphertext respectively. Then we count the key bytes required to partially encrypt or decrypt the active state bytes. The main advantage of the differential MITM technique over classical differential attacks lies in the simplicity of approximating the complexity. To accurately compute the cost of guessing the required key material in a MILP model, one has to take into account sophisticated key bridging techniques [LWZ16], making the model very complicated. But for differential MITM attacks, the relations between the round-key bytes of both the upper and lower parts only affect the complexity of the matching part. Hence, and since we assumed that the distinguisher will not be extended by more than 2 rounds at the beginning and by more than 2 rounds at the end, we only had to add few and simple extra constraints to compute the dimension of the key material involved in the attack. More precisely we only added to the model the relations involving bytes of the two first round keys and of the two last. Finally, because the key schedule is non-linear, there might be some differences unknown to the attacker that she has to guess to perform the attack. As it was complicated to give a precise estimation of the number of extra guesses

one has to do, we chose to omit this part. This has as a consequence that the complexity given by the tool is slightly underestimated but still remains close enough to its real value.

Distinguisher part. In order to avoid weak-key attacks, we also have to ensure that from any key there exists a corresponding related key satisfying the differential characteristic associated to the attack. In [DEFN22], the authors used an extra information for each key variable involved in order to to know whether this variable is *controlled* (i.e. its value can be freely chosen) or not. In particular they added to the model that k and $S(k)$ should not be both controlled as otherwise this would fix the value of k and thus lead to a weak-key attack, only working for a portion of the key space. To keep our model simpler, we did not follow this strategy. Instead, we checked through the callback functionality of Gurobi whether the current solution is a weak-key attack and, when it is, we added an extra constraint to remove it. Checking a solution is done by extracting the differences on the key and then by trying to echelonize the system of equations describing the key schedule.

6.3 Attack against AES-256

We describe now our attack against 13 rounds of AES-256. It is an attack in the related-key setting exploiting two related keys only. The attack procedure can be visualized in Figure 9. The differential distinguisher corresponds to the blue and red part of the depicted characteristic. The red bytes correspond to active S-box transitions and their number determines the probability of the distinguisher. As there are 18 active S-boxes in the distinguisher and the worst differential transitions for the S-box have a probability of 2^{-7}, the probability of the differential distinguisher can be estimated as $2^{-18\times 7} = 2^{-126}$.

Both the distinguisher and the attack were found together in a completely automated way by the tool described in Section 6.2. The upper part of the characteristic is identical to the one used to attack 12 rounds of AES-256 in [BDD+23] and for this reason the first part of the attack is very similar to the one described there.

Data. First, as also done in [BDD+23], one can remark that the difference on the plaintext P belongs to a vector space of dimension 11, as the bytes $3, 10, 14$ and 15 are inactive and $\Delta P[4] = \Delta P[5]$ as they are both equal on $\Delta k_2[4]$. Therefore, the attack starts by requesting the encryption of a structure of $2^{11\times 8} = 2^{88}$ plaintexts first under the secret key K and then under the related key K'.

Related-keys generation process. The two keys involved in the attack are related as described in Figure 9 and we explain here the algorithm which, given a key K, generates the second related one. First we choose a value for the difference a. As the difference on $k_{10}[4]$ is null, the adversary obtains the value of the difference d on $k_{12}[4]$. She then forces a difference 3d on $k_{12}[12]$ and thus obtains the difference c on $k_{11}[3]$. Finally, from both c and 2a, the adversary computes the difference b which fully determines K'. Note that we only know the value of a, the remaining ones depending on the key K are unknown at the beginning of the attack. In particular, the probability that the differential characteristic holds is around 2^{-4} (transitions $b \to a$, $c \oplus a \to d$, $\Delta x_2[0] \to a$ and $\Delta x_1[0] \to \Delta z_1[0]$ have to hold) and when it does its probability is $2^{-18\times 7} = 2^{-126}$.

Lower part. This step of the attack after the data generation consists in picking a plaintext P and its corresponding ciphertext C, guessing some key material and generate the possible ciphertexts $\widetilde{C}$ by the procedure described in Section 6.1. The attacker starts by guessing both $k_{13}[10]$ and $k_{13}[11]$. She deduces $k_{11}[11]$ and then the value of d from the relation $\Delta k_{10}[4] \oplus \Delta S(k_{11}[11]) \oplus \Delta k_{12}[4] = 0$. Similarly, she guesses both $k_{13}[2]$ and $k_{13}[3]$ and deduces $k_{11}[3]$ and c. She then guesses the bytes $0, 1, 8, 9$ of k_{13}, $3, 7, 11, 15$ of k_{12}, $7, 15$ of k_{11} as well as $k_{10}[3]$. This allows her to compute the differences on k_{10}, k_{12}

and k_{13} and thus to obtain the possible ciphertexts satisfying the output of the differential characteristic. For each of them, the probability that the corresponding plaintext belongs to the structure of 2^{88} plaintexts is $2^{88-128} = 2^{-40}$ and when it does, the attacker stores it in a hash table with the corresponding key guesses.

Upper part. In this step the attacker wants to generate the possible plaintexts $\widetilde{P}$. For this, she needs to guess 9 bytes of k_0, namely the bytes $\mathtt{0}, \mathtt{1}, \mathtt{2}, \mathtt{6}, \mathtt{7}, \mathtt{8}$, $\mathtt{11}, \mathtt{12}, \mathtt{13}$ and the two bytes $\mathtt{5}$ and $\mathtt{10}$ of k_1. Knowing these bytes will permit her to compute the bytes $z_0[\mathtt{0}]$, $z_1[\mathtt{4}]$ and $z_1[\mathtt{8}]$. Then, the attacker guesses the difference on the first column of k_2 as well as the difference $\mathtt{b}$ and propagates backwards the differences to compute $\widetilde{P}$. Note that there are only $2^{7\times 5} = 2^{35}$ possible differences for those 5 bytes since they all need to be valid transition through the S-Box for the fixed value $\mathtt{a}$. As a consequence, the attacker generates $2^{(9+2)\times 8} \times 2^{35} = 2^{123}$ tuples and checks for matches in the hash table.

Matching part. We expect $2^{123+120-128-8} = 2^{107}$ matches since the meet-in-the-middle is performed on both the plaintext and the value $\mathtt{b}$. For each of them, we first obtain the possible values for the last column of k_3 by using both the knowledge of the difference in the first column of k_2 and in the first column of k_4. At this point, we have around $2^{107+4} = 2^{111}$ tuples which is immediately reduced to 2^{103} by using the equation $k_3[\mathtt{3}] = k_5[\mathtt{2}] \oplus k_5[\mathtt{3}]$ (see Appendix D). The best algorithm to reconstruct the master key from those key bytes requires to make 3 more guesses and thus would lead to a complexity of 2^{127}.

Complexity. Since the procedure has to be performed 2^{126} times to get one right pair, the overall complexity would be: 2^{126} plaintext-ciphertext pairs for each key in order to get one pair, the time complexity is equivalent to $2^{126} \times (2^{120} + 2^{123} + 2^{127}) \approx 2^{253}$ encryptions and the memory complexity around $2^{120-40} \times 16 \times 2^{-4} = 2^{80}$ 128-bit blocks. It is possible to decrease the time complexity by asking for enough pairs to get two right ones so that the master key only has to be reconstructed for matches appearing twice. In this scenario, the data complexity is increased by two, the time complexity becomes $2^{127} \times (2^{120} + 2^{123} + 2^{103} + 2^{24}) \approx 2^{250}$ encryptions and the memory complexity is dominated by the storage of all matches, requiring around $2^{127} \times 2^{103} \times 30 \times 2^{-4} = 2^{231}$ 128-bit blocks.

Note that this attack is not a weak-key attack since for each key, there are values for a such that the differential holds. Furthermore, we can check during the attack process whether the guesses actually lead to a valid differential characteristic and thus finding the right value for a is fully amortized.

The complexities of our attack and a comparison with other related-key attacks against `AES-256` are summarized in Table 3.

6.3.1 Application of our Tool to `AES-128` and `AES-192`.

We used our tool to search for differential MITM attacks in the related-key setting against the other two versions of the `AES`. Our tool exhausted the search space for `AES-128` and did not find any non-trivial attack. This is not very surprising as this variant is known to resist related-key attacks much better than the bigger variants. On `AES-192` the tool outputted a single non-trivial attack scenario for 8 rounds but having a complexity higher than the best single-key attacks on this variant. This confirms in the case of the `AES` the assumption made in [BDD+23] that differential MITM attacks work better against ciphers for which the key is much longer than the state.

The 3 guesses are $k_{13}[6]$, $k_{12}[1]$ and $k_{12}[5]$

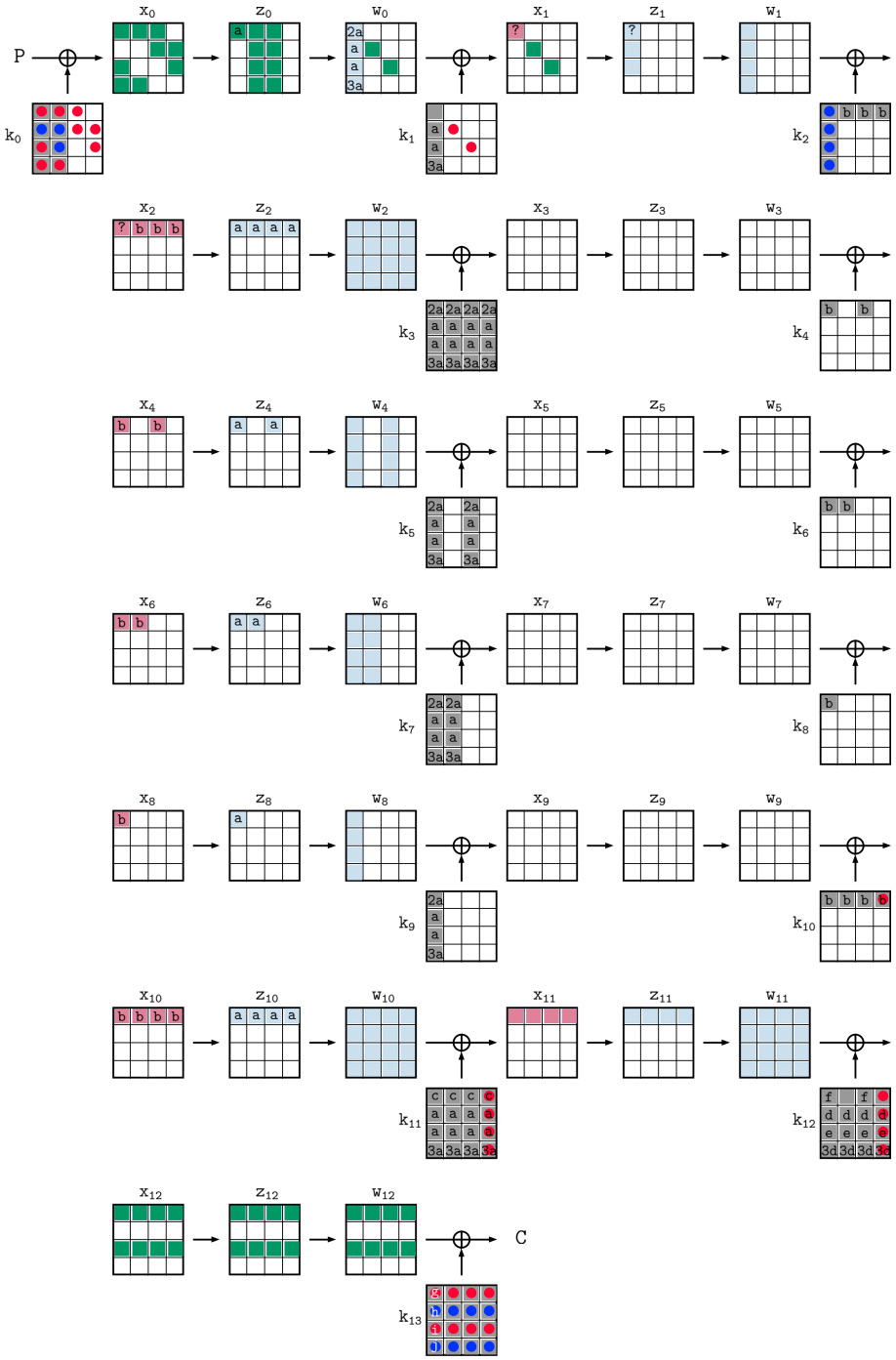


Figure 9: Differential meet-in-the-middle attack against `AES-256`. Blue bytes belong to the distinguisher. Among them, the red bytes correspond to the active S-boxes. Green bytes are part of the key-recovery procedure. White bytes are inactive. Red dots on key bytes correspond to key values that have to be guessed. Finally, blue dots correspond to key differences that have to be guessed.

Table 3: Best known attacks against AES-256 together with the the attack presented in this section. † The parameter s should be such that $0 \leq s \leq 7.5$.

# Rounds	Data	Time	Memory	# Related keys	Type	Ref.
9	2^{120}	2^{203}	2^{203}	0	MITM	[DFJ13]
10	$2^{114.9}$	$2^{171.8}$	2^{64}	256	Rectangle	[BDK05]
10	$2^{113.9}$	$2^{172.8}$	-	64	Rectangle	[KHP07]
12	2^{89}	2^{206}	$2^{71.6}$	2	differential MITM	[BDD+23]
14	$2^{99.5}$	$2^{99.5}$	2^{56}	4	Boomerang	[BK09]
14	2^{91+s}	2^{92+s}	2^{89-s}	2^{19-s}	Boomerang	[GSW22] †
14	$2q$	$q2^{66}$	-	$2q$	q-multicollisions	[GLMS18]
14	2^{125}	2^{125}	2^{65}	2^{32}	basic RK differential	[GLMS18]
13	2^{126}	2^{253}	2^{80}	2	differential MITM	our
13	2^{126}	2^{250}	2^{231}	2	differential MITM	our

7 Conclusion and open problems

In the first part of this work we presented two different approaches for the optimal related-key characteristics problem for the AES. The AES is the most important symmetric-key algorithm in use, and many different cryptanalysis techniques applied against it need the existence of good differential characteristics to work. It is thus important for such an important target, to be able to confirm by other methods the results that were found by generic solvers, especially as bugs in such solvers have already been discovered in the past (see for example [EY20]). Our dynamic approach method permitted to better understand the underlying difficulty of this problem and to propose a pure algorithmic approach that we believe can be adapted to other ciphers. In parallel, it permitted to solve an open problem, as presenting a memory-efficient dynamic programming solution for the larger variants of AES was considered to be hard.

Our work opens also several problems regarding the adaptation of our models and algorithms to ciphers of different nature. For example, it would be interesting to try to adapt our algorithms to ciphers whose matrix is not MDS or to see how the dynamic programming approach could be adapted to ciphers where the key schedule algorithm does not work by columns. Finally, we believe that the MILP tool we developed for differential MITM cryptanalysis can serve as a basis to study the applicability of this newly proposed method to other primitives, by starting by those whose structure is close to the AES.

References

[BDD+23] Christina Boura, Nicolas David, Patrick Derbez, Gregor Leander, and María Naya-Plasencia. Differential meet-in-the-middle cryptanalysis. In Helena Handschuh and Anna Lysyanskaya, editors, *CRYPTO 2023, Part III*, volume 14083 of *Lecture Notes in Computer Science*, pages 240–272. Springer, 2023.

[BDK05] Eli Biham, Orr Dunkelman, and Nathan Keller. Related-key boomerang and rectangle attacks. In Ronald Cramer, editor, *EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 507–525. Springer, 2005.

[BDK+18] Achiya Bar-On, Orr Dunkelman, Nathan Keller, Eyal Ronen, and Adi Shamir. Improved key recovery attacks on reduced-round AES with practical data and

memory complexities. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part II*, volume 10992 of *Lecture Notes in Computer Science*, pages 185–212. Springer, 2018.

[BK09] Alex Biryukov and Dmitry Khovratovich. Related-key cryptanalysis of the full AES-192 and AES-256. In Mitsuru Matsui, editor, *ASIACRYPT 2009*, volume 5912 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2009.

[BKN09] Alex Biryukov, Dmitry Khovratovich, and Ivica Nikolic. Distinguisher and related-key attack on the full AES-256. In Shai Halevi, editor, *CRYPTO 2009*, volume 5677 of *Lecture Notes in Computer Science*, pages 231–249. Springer, 2009.

[BL23] Augustin Bariant and Gaëtan Leurent. Truncated boomerang attacks and application to AES-based ciphers. In Carmit Hazay and Martijn Stam, editors, *EUROCRYPT 2023, Part IV*, volume 14007 of *Lecture Notes in Computer Science*, pages 3–35. Springer, 2023.

[BN10] Alex Biryukov and Ivica Nikolic. Automatic search for related-key differential characteristics in byte-oriented block ciphers: Application to AES, Camellia, Khazad and others. In Henri Gilbert, editor, *EUROCRYPT 2010*, volume 6110 of *Lecture Notes in Computer Science*, pages 322–344. Springer, 2010.

[BNS14] Christina Boura, María Naya-Plasencia, and Valentin Suder. Scrutinizing and improving impossible differential attacks: Applications to CLEFIA, Camellia, LBlock and Simon. In Palash Sarkar and Tetsu Iwata, editors, *ASIACRYPT 2014, Part I*, volume 8873 of *Lecture Notes in Computer Science*, pages 179–199. Springer, 2014.

[BNS19] Xavier Bonnetain, María Naya-Plasencia, and André Schrottenloher. Quantum security analysis of AES. *IACR Trans. Symmetric Cryptol.*, 2019(2):55–93, 2019.

[BS90] Eli Biham and Adi Shamir. Differential cryptanalysis of DES-like cryptosystems. In Alfred Menezes and Scott A. Vanstone, editors, *CRYPTO '90*, volume 537 of *Lecture Notes in Computer Science*, pages 2–21. Springer, 1990.

[DEFN22] Patrick Derbez, Marie Euler, Pierre-Alain Fouque, and Phuong Hoa Nguyen. Revisiting related-key boomerang attacks on AES using computer-aided tool. In Shweta Agrawal and Dongdai Lin, editors, *ASIACRYPT 2022, Part III*, volume 13793 of *Lecture Notes in Computer Science*, pages 68–88. Springer, 2022.

[DFJ13] Patrick Derbez, Pierre-Alain Fouque, and Jérémy Jean. Improved key recovery attacks on reduced-round AES in the single-key setting. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *Lecture Notes in Computer Science*, pages 371–387. Springer, 2013.

[DR01] Joan Daemen and Vincent Rijmen. The wide trail design strategy. In Bahram Honary, editor, *IMA 2001*, volume 2260 of *Lecture Notes in Computer Science*, pages 222–238. Springer, 2001.

[EY20] Muhammad ElSheikh and Amr M. Youssef. A cautionary note on the use of gurobi for cryptanalysis. Cryptology ePrint Archive, Paper 2020/1112, 2020. https://eprint.iacr.org/2020/1112.

[FIP01] FIPS 197. Announcing the Advanced Encryption Standard (AES). National Institute for Standards and Technology, Gaithersburg, MD, USA, November 2001.

[FJP13] Pierre-Alain Fouque, Jérémy Jean, and Thomas Peyrin. Structural evaluation of AES and chosen-key distinguisher of 9-round AES-128. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part I*, volume 8042 of *Lecture Notes in Computer Science*, pages 183–203. Springer, 2013.

[GLMS18] David Gérault, Pascal Lafourcade, Marine Minier, and Christine Solnon. Revisiting AES related-key differential attacks with constraint programming. *Inf. Process. Lett.*, 139:24–29, 2018.

[GLMS20] David Gérault, Pascal Lafourcade, Marine Minier, and Christine Solnon. Computing AES related-key differential characteristics with constraint programming. *Artif. Intell.*, 278, 2020.

[GMS16] David Gérault, Marine Minier, and Christine Solnon. Constraint programming models for chosen key differential cryptanalysis. In Michel Rueher, editor, *CP 2016*, volume 9892 of *Lecture Notes in Computer Science*, pages 584–601. Springer, 2016.

[GSW22] Jian Guo, Ling Song, and Haoyang Wang. Key structures: Improved related-key boomerang attack against the full AES-256. In Khoa Nguyen, Guomin Yang, Fuchun Guo, and Willy Susilo, editors, *ACISP 2022*, volume 13494 of *Lecture Notes in Computer Science*, pages 3–23. Springer, 2022.

[Jea16] Jérémy Jean. TikZ for Cryptographers. https://www.iacr.org/authors/tikz/, 2016.

[KHP07] Jongsung Kim, Seokhie Hong, and Bart Preneel. Related-key rectangle attacks on reduced AES-192 and AES-256. In Alex Biryukov, editor, *FSE 2007*, volume 4593 of *Lecture Notes in Computer Science*, pages 225–241. Springer, 2007.

[KLPS17] Khoongming Khoo, Eugene Lee, Thomas Peyrin, and Siang Meng Sim. Human-readable proof of the related-key security of AES-128. *IACR Trans. Symmetric Cryptol.*, 2017(2):59–83, 2017.

[Knu95] Lars R. Knudsen. Truncated and higher order differentials. In Bart Preneel, editor, *Fast Software Encryption*, pages 196–211, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[LJW14] Leibo Li, Keting Jia, and Xiaoyun Wang. Improved single-key attacks on 9-round AES-192/256. In Carlos Cid and Christian Rechberger, editors, *Fast Software Encryption - 21st International Workshop, FSE 2014, London, UK, March 3-5, 2014. Revised Selected Papers*, volume 8540 of *Lecture Notes in Computer Science*, pages 127–146. Springer, 2014.

[LP21] Gaëtan Leurent and Clara Pernot. New representations of the AES key schedule. In Anne Canteaut and François-Xavier Standaert, editors, *EUROCRYPT 2021, Part I*, volume 12696 of *Lecture Notes in Computer Science*, pages 54–84. Springer, 2021.

[LWZ16] Li Lin, Wenling Wu, and Yafei Zheng. Automatic search for key-bridging technique: Applications to lblock and TWINE. In Thomas Peyrin, editor, *FSE 2016*, volume 9783 of *Lecture Notes in Computer Science*, pages 247–267. Springer, 2016.

[Mat94] Mitsuru Matsui. On correlation between the order of S-boxes and the strength of DES. In Alfredo De Santis, editor, *EUROCRYPT '94*, volume 950 of *Lecture Notes in Computer Science*, pages 366–375. Springer, 1994.

[PF] Laurent Perron and Vincent Furnon. Or-tools.

[PFL16] Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. *Choco Solver Documentation*. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., 2016.

[RGMS22] Loïc Rouquette, David Gérault, Marine Minier, and Christine Solnon. And Rijndael?: Automatic related-key differential analysis of Rijndael. In Lejla Batina and Joan Daemen, editors, *AFRICACRYPT 2022*, Lecture Notes in Computer Science, pages 150–175. Springer Nature Switzerland, 2022.

[Tea06] Gecode Team. Gecode: Generic constraint development environment. Available from http://www.gecode.org/, 2006.

[ZK16] Neng-Fa Zhou and Håkan Kjellerstrand. The Picat-SAT compiler. In Marco Gavanelli and John H. Reppy, editors, *PADL 2016*, volume 9585 of *Lecture Notes in Computer Science*, pages 48–62. Springer, 2016.

A Basic constraints for the MILP model

We give here the basic constraints that are part of our MILP model. The XOR $a \oplus b \oplus c = 0$ between differential bytes is modeled with three inequalities, which ensure that $a+b+c \neq 1$. The XOR of 4 bytes is modeled similarly. To model the MDS property of the MixColumns operation, some dummy variables are used to indicate whether the columns are active or not before and after MixColumns. The basic constraints are a straightforward encoding of the AES with these rules.

Algorithm 5: `addXorConstr(m, a, b, c)`

```
m.addConstr(1 - a + b + c ≥ 1)
m.addConstr(a + 1 - b + c ≥ 1)
m.addConstr(a + b + 1 - c ≥ 1)
```

Algorithm 6: `addXorConstr(m, a, b, c, d)`

```
m.addConstr(1 - a + b + c + d ≥ 1)
m.addConstr(a + 1 - b + c + d ≥ 1)
m.addConstr(a + b + 1 - c + d ≥ 1)
m.addConstr(a + b + c + 1 - d ≥ 1)
```

Algorithm 7: `addAddRoundKeyBasicConstr(m, R)`

```
for r = 1...R do
    for i = 0...15 do
        // Y_r = MC ∘ SR ∘ SB(X_r)
        addXorConstr(m, X_r[i], K_r[i], Y_{r-1}[i])
```

Algorithm 8: `addKeyScheduleBasicConstr(m, R, version` $= 128$`)`

```
for r = 1... R do
    for i = 0...15 do
        if i mod 4 = 0 then
            ρ(i) ← i + 7 mod 16
            addXorConstr(m, K_r[i], K_{r−1}[i], K_{r−1}[ρ(i)])
        else
            addXorConstr(m, K_r[i], K_{r−1}[i], K_r[i-1])
```

Algorithm 9: `addShiftRowsMixColumnsBasicConstr(m, R)`

```
for r= 0...R−2 do
    for c= 0...3 do
        e ← 0
        for i = 0...3 do
            // Y_r = MC ∘ SR ∘ SB(X_r)
            e ← e + Y_r[c + 4i]
            e ← e + X_r[(c + i) mod 4 + 4i]
        Let f be a dummy binary variable
        m.addConstr(e ≤ 8f)
        m.addConstr(e ≥ 5f)
▷ No MixColumns for the last round
for c= 0...3 do
    for i= 0...3 do
        m.addConstr(Y_{R−1}[c+4i] = X_{R−1}[(c + i) mod 4 + 4i]]
```

Algorithm 10: `getObjectiveFunction(R, version` $= 128$`)`

```
obj ← 0
for r= 0... R - 1 do
    for i = 0...15 do
        obj ← obj + X_r[i]
    for i = 0...3 do
        obj ← obj + K_r[3 + 4i]
return obj
```

Algorithm 11: `addKeyScheduleExtraConstr(m, R, version` $= 128$`)`

```
for r = 2... R do
    for i = 0...15 do
        if i mod 4 > 1 then
            addXorConstr(m, K_r[i], K_{r−2}[i], K_r[i-2])
```

Table 4: Comparison of the number of variables and constraints between our model and the model of [GLMS20] for r rounds of `AES-128`

CP model of [GLMS20]		Our MILP model	
diff var.	$\approx 228(r-1)^2$	var. for MC	$4(r-1)$
other var	$80r+32$	var. for MC extra contr.	$81(r-2)$
		other var.	$80r+32$
basic constr.	$56r-16$	basic constr.	$120r+24$
diff var. constr.	$\approx 308(r-1)^2+167(r-1)^3$	extra KS constr.	$24(r-2)$
# KS eqs with 3 var. (eg. 266 for $r=10$)		extra MC constr.	$270(r-2)$
# KS eqs with 4 var. (eg. 1104 for $r=10$)			

B Number of variables and constraints of our MILP model

C Examples of optimal truncated differential characteristics for `AES-192` and `AES-256`

We provide here examples of optimal truncated differential characteristics for `AES-192` and `AES-256`. All the characteristics were designed by using the library [Jea16].

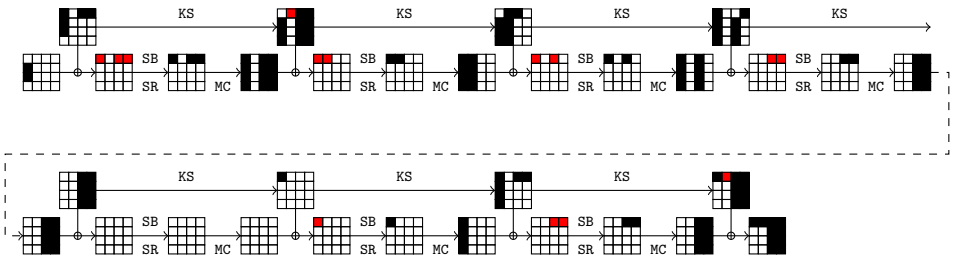


Figure 10: An optimal truncated differential characteristic for `AES-192` over 7 rounds with 14 active Sboxes (in red).

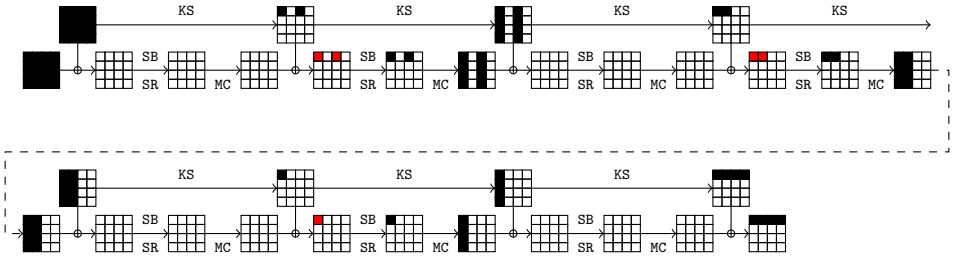


Figure 11: An optimal truncated differential characteristic for `AES-256` over 7 rounds with 5 active Sboxes (in red).

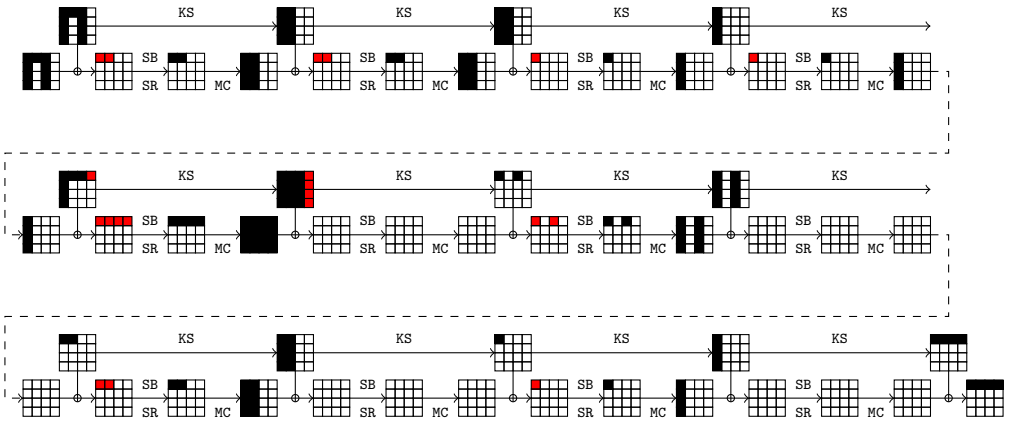


Figure 13: An optimal truncated differential characteristic for AES-256 over 12 rounds with 20 active Sboxes (in red).

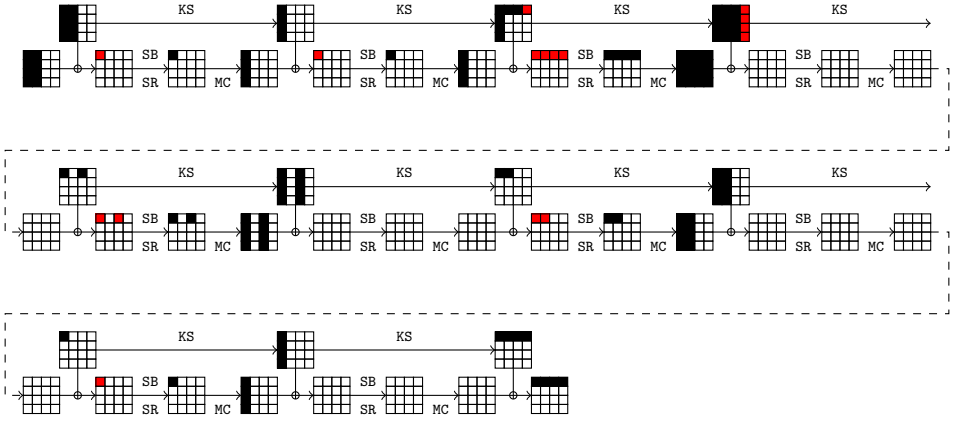


Figure 12: An optimal truncated differential characteristic for AES-256 over 10 rounds with 16 active Sboxes (in red).

D Matching Part for the 13-round Attack against AES-256

A critical part of this attack is the filtering provided by the equation $k_3[3] = k_5[2] \oplus k_5[3]$. While it is quite clear that $k_3[3]$ belongs to k_{in} it is much more complex to see that we are also able to compute $k_5[2] \oplus k_5[3]$ from k_{out}. Actually, this linear combination can be computed from only 2 bytes of k_{out}, exploiting the sparsity of the equations describing the key schedule:

$$\begin{aligned} k_5[2] \oplus k_5[3] &= k_7[1] \oplus k_7[2] \oplus k_7[2] \oplus k_7[3] \\ &= k_7[1] \oplus k_7[3] \\ &= k_9[0] \oplus k_9[1] \oplus k_9[2] \oplus k_9[3] \\ &= S(k_{10}[3]) \oplus k_{11}[0] \oplus k_{11}[0] \oplus k_{11}[1] \oplus k_{11}[1] \oplus k_{11}[2] \oplus k_{11}[2] \oplus k_{11}[3] \\ &= S(k_{10}[3]) \oplus k_{11}[3] \end{aligned}$$