Computing AES related-key differential characteristics with constraint programming

David Gérault, Pascal Lafourcade, Marine Minier, Christine Solnon

Computing AES Related-Key Differential Characteristics with Constraint Programming

David Gerault[a], Pascal Lafourcade[a], Marine Minier[b], Christine Solnon[c]

[a] *Université Clermont Auvergne, CNRS, LIMOS, F-63000 Clermont, France*
[b] *Université de Lorraine, CNRS, Inria, LORIA, F-54000 Nancy, France*
[c] *Univ. Lyon, INSA Lyon, CNRS, LIRIS, F-69621, France*

Abstract

Cryptanalysis aims at testing the properties of encryption processes, and this usually implies solving hard optimization problems. In this paper, we focus on related-key differential attacks for the Advanced Encryption Standard (AES), which is the encryption standard for block ciphers. To mount these attacks, cryptanalysts need to solve the optimal related-key differential characteristic problem. Dedicated approaches do not scale well for this problem, and need weeks to solve its hardest instances.

In this paper, we improve existing Constraint Programming (CP) approaches for computing optimal related-key differential characteristics: we add new constraints that detect inconsistencies sooner, and we introduce a new decomposition of the problem in two steps. These improvements allow us to compute all optimal related-key differential characteristics for AES-128, AES-192 and AES-256 in a few hours.

Keywords:
Constraint Programming, AES, Differential cryptanalysis.

1. Introduction

Since 2001, AES (Advanced Encryption Standard) is the encryption standard for block ciphers [FIP01]. It guarantees communication confidentiality by using a secret key K to cipher an original plaintext X into a ciphertext $AES_K(X)$, in such a way that the ciphertext can further be deciphered into the original one using the same key, *i.e.*, $X = AES_K^{-1}(AES_K(X))$.

Cryptanalysis aims at testing whether an attacker can observe any non-random property in the encryption process: if he cannot, then confidentiality

is guaranteed. In particular, *differential cryptanalysis* [BS91] studies the propagation of the difference $\delta X = X \oplus X'$ between two plaintexts X and X' through the cipher, where $\oplus$ is the exclusive or (XOR) operator. If the distribution of the output difference $\delta C = AES_K(X) \oplus AES_K(X')$ is non uniform, then an adversary can exploit this non-uniformity to guess the key K used to encrypt messages with difference δX faster than by performing exhaustive search: confidentiality is therefore lowered.

Today, differential cryptanalysis is public knowledge, and block ciphers such as AES have proven security bounds against differential attacks. Hence, [Bih93] proposed to consider differences not only between the plaintexts X and X' but also between the keys K and K' to mount *related-key attacks*. In this case, the cryptanalyst is interested in finding *optimal related-key differentials*, *i.e.*, input and output differences that maximize the probability of obtaining the output difference given the input difference. In other words, we search for δX, δK, and δC that maximize the probability that δC is equal to $AES_K(X) \oplus AES_{K \oplus \delta K}(X \oplus \delta X)$ for any plaintext X and any key K.

Finding an optimal related-key differential is a highly combinatorial problem that hardly scales. To simplify this problem, we usually compute *differential characteristics* which additionally specify intermediate differences after each step of the ciphering process. Also, to reduce the search space, Knudsen [Knu95] has introduced *truncated differential characteristics* where sequences of bits are abstracted by single bits that indicate whether sequences contain differences or not. Typically, each byte (8-bit sequence) is abstracted by a single bit (or, equivalently, a Boolean value). In this case, the goal is no longer to find the exact differences, but to find the positions of these differences, *i.e.*, the presence or absence of a difference for every byte. However, some truncated differential characteristics may not be valid (*i.e.*, there do not exist actual byte values corresponding to these difference positions) because some constraints at the byte level are relaxed when reasoning on difference positions. Hence, the optimal related-key differential characteristic problem is usually solved in two steps [BN10, FJP13]. In Step 1, every differential byte is abstracted by a Boolean variable that indicates whether there is a difference or not at this position, and we search for all truncated differential characteristics. Then, for each of these truncated differential characteristics, Step 2 aims at deciding whether it is valid (*i.e.*, whether it is possible to find actual byte values for every Boolean variable) and, if it is valid, at finding the actual byte values that maximize the probability.

Two main approaches have been proposed to solve the optimal related-key

differential characteristic problem on AES: a graph traversal approach [FJP13], and a Branch & Bound approach [BN10]. The approach of [FJP13] requires about 60 GB of memory when the key has 128 bits, and it has not been extended to larger keys. The approach of [BN10] only takes several megabytes of memory, but solving Step 1 requires several days of computation when the key has 128 bits, and several weeks when the key has 192 bits. Of course, each of these problems must be solved only once, and CPU time is not the main issue provided that it is "reasonable". However, during the process of designing new ciphers, this evaluation sometimes needs to be repeated several times. Hence, even though not crucial, a good CPU time is a desirable feature. Another point that should not be neglected is the time needed to design and implement these approaches: To ensure that the computation is completed within a "reasonable" amount of time, it is necessary to reduce branching by introducing clever reasoning. Of course, this hard task is also likely to introduce bugs, and checking the optimality of the computed solutions may not be so easy. Finally, reproducibility may also be an issue. Other researchers may want to adapt these algorithms to other problems, with some common features but also some differences, and this may again be very difficult and time-consuming.

1.1. Declarative approaches for cryptanalytic problems

An appealing alternative to dedicated approaches is to use generic solvers such as Integer Linear Programming (ILP), Boolean satisfiability (SAT) or Constraint Programming (CP). In this case, we have to design a model of the problem (by means of linear inequalities for ILP, Boolean clauses for SAT, and constraints for CP), and this model is automatically solved by generic solvers.

ILP. Many symmetric cryptanalysis problems on different ciphers have been tackled with ILP. For example, ILP has been used to find optimal (related key) differential characteristics against bit-oriented block ciphers such as SIMON, PRESENT or LBlock [SHW+14], to search for impossible differentials [ST17], and to provide security bounds in the design of the new lightweight block cipher SKINNY [BJK+16]. ILP models can only contain linear inequalities. Therefore, it is necessary to transform non-linear operators into sets of linear inequalities. However, the resulting ILP model may not scale well. For example, in [AST+17], authors introduce an ILP model of the non-linear part of a block cipher (such as the SubBytes operation used in

AES). They present some results on SKINNY-128 where the time required to find differential paths is about 15 days.

SAT. SAT and Satisfiability Modulo Theories (SMT) have also been used to solve cryptanalysis problems. For example, in [MP13], Mouha and Preneel use a SAT solver to search for optimal differential characteristics for a type of ciphers called ARX, in which the nonlinear components are different from the SubByte operation used in AES. In [SWW17], Sun *et al.* propose a SAT/SMT model to search for division properties on ARX and word-based block ciphers, and they show on SHACAL-2 that it scales better than ILP. In [KLT15] the authors analyze the general class of functions underlying the Simon block cipher: They derive SAT/SMT models for the exact differential and linear behaviour of Simon-like round functions, and they use SAT/SMT solvers to compute optimal differential and linear characteristics for Simon. The SAT solver used in [KLT15] is CryptoMiniSat [SNC09], which is well suited to solve cryptanalysis problems. In particular, it introduces XOR-clauses to easily model XOR operations, and it uses Gaussian elimination to efficiently propagate these XOR-clauses. These XOR-clauses can be used to model bitwise XOR operations in Step 2. However, they cannot be used to model these operations in Step 1. Indeed, if $1 \oplus 1 = 0$ at a bitwise level (during Step 2), this is no longer true during Step 1 because the XOR of two bytes different from 0 may be either 0 or a value different from 0, depending on whether the two bytes have the same value or not (see Section 3.2). Similarly to ILP, non linear operations such as SubBytes are not straightforward to model by means of clauses. In [Laf18], Lafitte shows how to encode a relation associated with a non linear operation into a set of clauses and, in [SWW18], Sun *et al.* show how to reduce the number of clauses of this kind of encoding by using the same approach as in [AST+17].

CP. CP models have been used in [LCM+17] for modeling algebraic side-channel attack on AES and in [RSM+11] to design the non-linear part of a block cipher with good properties. Recently, we have introduced CP models for finding related-key differential characteristics for AES [GMS16], Midori [GL16], and SKINNY [SGL+17]. These preliminary work have shown us that off-the-shelf CP solvers are able to solve these problems quicker than dedicated approaches. In particular, the CP model of [GMS16] for Step 1 of AES when the key has 128 bits is solved in a few hours by CP solvers such as Gecode [Gec06], Choco [PFL16], or Chuffed [CS14]. This CP model has

allowed us to find a better solution than the one claimed to be optimal in [FJP13, BN10] for 4 rounds of AES when the key has 128 bits. However, the CP model for Step 1 described in [GMS16] is not able to solve all instances of AES within a reasonable amount of time. In [GLMS17, GLMS18], we show how to decompose the hardest instances into independent sub-problems which can be solved in parallel, thus allowing us to solve all AES instances. However, computing optimal differential characteristics with the approach of [GLMS17, GLMS18] still is very time consuming. For example, the hardest instance (for 10 rounds of AES-192) needs more than two months of CPU time to be solved.

1.2. Contributions and outline of the paper

In this paper, we introduce a new CP model for Step 1 and a new decomposition of the solution process in two steps that allow us to solve every instance of AES, for all key lengths, in a few hours: All instances but two are solved in less than one hour, while the two hardest instances are solved in 5 hours and 15 hours, respectively.

In Section 2, we describe the context of this work: the AES ciphering process, the AES related-key differential characteristic problem, the two step solving process that is used in [FJP13, BN10, GMS16], how to use differential characteristics to design related-key attacks, and the basic principles of CP.

In Section 3, we describe the basic CP model of [MSR14] for solving Step 1. This model is a straightforward encoding which associates a constraint between Boolean variables with every AES operation. A weakness of this model is that the XOR constraint at the Boolean level is a poor abstraction of the XOR operation at the Byte level. As a consequence, there is a huge number of truncated differential characteristics, most of which are discarded at Step 2 because they are not valid.

In Section 4, we introduce a first contribution of this paper, which is a new CP model for solving Step 1. The idea is to generate new XOR equations by combining initial equations coming from the key schedule, in a preprocessing step, and to use these new equations to filter the search space. We also tighten the definition of the XOR constraint between Boolean variables by reasoning on differences between bytes. We compare our new model with the model introduced in [GMS16], and we show that it is faster, and that it drastically reduces the number of non valid truncated differential characteristics for the most challenging instance.

In Section 5, we describe the CP model for solving Step 2, which is a straightforward extension of the model of [GMS16] to other key lengths than 128 bits. This model is able to quickly find the optimal byte-consistent solution, given a truncated differential characteristic. However, for some instances, there are many truncated differential characteristics and in this case finding the optimal solution for all truncated differential characteristics becomes time consuming.

In Section 6, we introduce a second contribution of this paper, which is a new decomposition of the solution process in two steps which allows us to solve the full problem much quicker.

In Section 7, we give an overview of the new cryptanalysis results that have been derived from the new related-key differential characteristics computed thanks to our new CP models, and we describe some further work.

2. Background

We start by presenting the AES encryption scheme and introducing the notations used in our modeling. Then, we describe the AES related-key differential characteristic problem, the two step solving process generally used to solve it, and how differential characteristics are used to mount related-key attacks. Finally we recall some basic principles of constraint programming.

Throughout the paper, we note $[i, j]$ the set of all integer values ranging from i to j, *i.e.*, $[i, j] = \{v \in \mathbb{N} : i \leq v \leq j\}$.

2.1. Description of AES

AES ciphers blocks of length $n = 128$ bits, where each block is seen as a 4×4 matrix of bytes, where a byte is a sequence of 8 bits. Given a 4×4 matrix of bytes M, we note $M[j][k]$ the byte at row $j \in [0, 3]$, and column $k \in [0, 3]$. The length of keys is $l \in \{128, 192, 256\}$ bits, and we note AES-l the AES with keys of length l. The only difference when changing the length l of the key is in the KeySchedule operator. We illustrate AES and describe our cryptanalytic models for the case where $l = 128$, such that the key is a 4×4 matrix of bytes. We describe the cases where $l \in \{192, 256\}$ in Appendix.

Like most of today's block ciphers, AES is an iterative process which is composed of r rounds. The number of rounds r depends on the key length: $r = 10$ (resp. 12 and 14) when $l = 128$ (resp. 192 and 256). An AES round has an SPN (Substitution-Permutation Network) structure

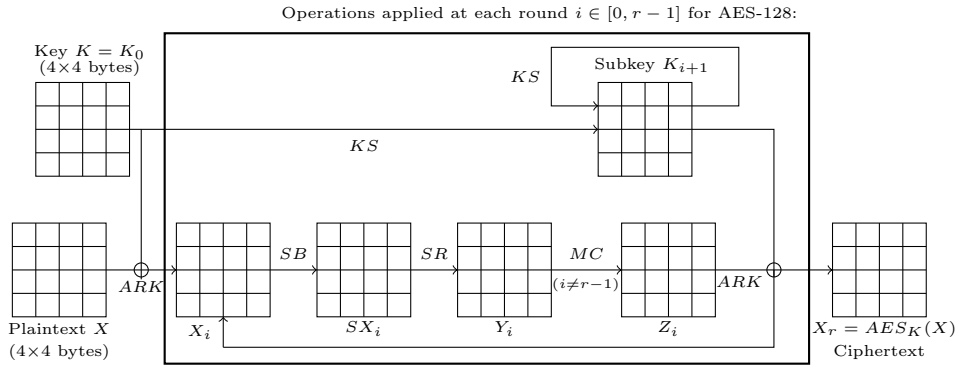

Figure 1: AES ciphering process for 128 bit keys. Each 4×4 array represents a group of 16 bytes. Before the first round, X_0 is obtained by applying ARK on the initial text X and the initial key $K = K_0$. Then, for each round $i \in [0, r-1]$, SB is applied on X_i to obtain SX_i, SR is applied on SX_i to obtain Y_i, MC is applied on Y_i to obtain Z_i (except during the last round when $i = r-1$), KS is applied on K_i to obtain K_{i+1}, and ARK is applied on K_{i+1} and Z_i to obtain X_{i+1}. The ciphertext is X_r.

and is described in Fig. 1 for $l = 128$. It uses the following five operations, that are described in details below: SubBytes (SB), ShiftRows (SR), MixColumns (MC), AddRoundKey (ARK), and KeySchedule (KS). Before the first round, AddRoundKey is applied on the original plaintext X and the initial key $K_0 = K$ to obtain $X_0 = ARK(X, K_0)$. Then, for each round $i \in [0, r-1]$: SubBytes is applied on X_i to obtain $SX_i = SB(X_i)$; ShiftRows is applied on SX_i to obtain $Y_i = SR(SX_i)$; MixColumns is applied on Y_i to obtain $Z_i = MC(Y_i)$ (except during the last round where MixColumns is omitted so that $Z_{r-1} = Y_{r-1}$); KeySchedule is applied on K_i to obtain $K_{i+1} = KS(K_i)$; and AddRoundKey is applied on Z_i and K_{i+1} to obtain $X_{i+1} = ARK(Z_i, K_{i+1})$. The final ciphertext is X_r.

SubBytes. SB is a non-linear permutation which is applied on each byte of X_i separately, according to a look-up table (called S-box) $S : [0, 255] \to [0, 255]$, *i.e.*,

$$\forall i \in [0, r-1], \forall j, k \in [0, 3], SX_i[j][k] = S(X_i[j][k]).$$

ShiftRows. SR rotates on the left by one (resp. two and three) byte position the second (resp. third and fourth) row of SX_i, *i.e.*,

$$\forall i \in [0, r-1], \forall j, k \in [0, 3], Y_i[j][k] = SX_i[j][(k + j)\%4]$$

where % is the modulo operator that returns the remainder of the euclidean division.

MixColumns. MC multiplies each column of the input matrix Y_i by a 4×4 fixed matrix M to obtain each corresponding output column in the matrix Z, *i.e.*,

$$\forall i \in [0, r-2], \forall j, k \in [0, 3], Z_i[j][k] = \bigoplus_{x=0}^{3} M[j][x] \cdot Y_i[x][k]$$

where $\cdot$ is a finite field multiplication operator.

For the last round, MC is not applied and we have $Z_{r-1} = Y_{r-1}$.

Maximum Distance Separable property. The coefficients of the matrix M used by MC have been chosen to ensure a so-called MDS (*Maximum Distance Separable*) property. This property comes from the theory of error correcting codes [MS77] and guarantees the maximal possible diffusion property [Sin06]. For AES, it implies that, for each round $i \in [0, r-2]$ and each column $k \in [0, 3]$, the number of bytes that are different from zero in column k of Y_i or Z_i is either equal to zero or strictly greater than four, *i.e.*,

$$\forall i \in [0, r-2], \forall k \in [0, 3], (\sum_{j=0}^{3} (Y_i[j][k] \neq 0) + (Z_i[j][k] \neq 0)) \in \{0, 5, 6, 7, 8\}.$$

This property also holds when XORing different columns. More precisely, let be $i_1, i_2 \in [0, r-2]$ two round numbers, and $k_1, k_2 \in [0, 3]$ two column numbers. For every row $j \in [0, 3]$, we have

$$\begin{aligned} Z_{i_1}[j][k_1] \oplus Z_{i_2}[j][k_2] &= (\bigoplus_{x=0}^{3} M[j][x] \cdot Y_{i_1}[x][k_1]) \oplus (\bigoplus_{x=0}^{3} M[j][x] \cdot Y_{i_2}[x][k_2]) \\ &= \bigoplus_{x=0}^{3} M[j][x] \cdot (Y_{i_1}[x][k_1] \oplus Y_{i_2}[x][k_2]) \end{aligned}$$

Therefore, the MDS property also holds for the result of the XOR of two different columns, *i.e.*,

$$\forall i_1, i_2 \in [0, r-2], \forall k_1, k_2 \in [0, 3], (\sum_{j=0}^{3} (dY[j] \neq 0) + (dZ[j] \neq 0)) \in \{0, 5, 6, 7, 8\}.$$

where $dY[j] = Y_{i_1}[j][k_1] \oplus Y_{i_2}[j][k_2]$ and $dZ[j] = Z_{i_1}[j][k_1] \oplus Z_{i_2}[j][k_2]$.

AddRoundKey. Before the first round, ARK performs a XOR between the initial plaintext X and the initial key K_0 to obtain X_0:

$$\forall j, k \in [0,3], X_0[j][k] = X[j][k] \oplus K_0[j][k].$$

Then, at each round i, ARK performs a XOR between Z_i and subkey K_{i+1} to obtain X_{i+1}, *i.e.*,

$$\forall i \in [0, r-1], \forall j, k \in [0,3], X_{i+1}[j][k] = Z_i[j][k] \oplus K_{i+1}[j][k].$$

KeySchedule. KS computes the subkey K_{i+1} of each round $i \in [0, r-1]$ from the initial key K. Its exact definition depends on the length l of the key. We describe it for $l = 128$ (see Appendix for keys of length $l \in \{192, 256\}$). The subkey at round 0 is the initial key, *i.e.*, $K_0 = K$. For each round $i \in [0, r-1]$, the subkey K_{i+1} is generated from the previous subkey K_i as follows:

- The first column of K_{i+1} is obtained from the first and last columns of K_i in two steps. First, we apply the SubBytes operation on all bytes of the last column of K_i. We note $SK_i[j][3]$ the resulting byte for row j, *i.e.*,

  $$\forall i \in [0, r-1], \forall j \in [0,3], SK_i[j][3] = S(K_i[j][3]).$$

  Second, we rotate up all bytes in SK_i by one byte position, and then XOR them with those in the first column of K_i. Moreover, a constant c_i is XORed with the byte at row 0.

  $$\begin{aligned} \forall i \in [0, r-1], K_{i+1}[0][0] &= SK_i[1][3] \oplus K_i[0][0] \oplus c_i \\ \forall i \in [0, r-1], \forall j \in [1,3], K_{i+1}[j][0] &= SK_i[(j+1)\%4][3] \oplus K_i[j][0]. \end{aligned}$$

- For the last three columns $k \in [1,3]$, we simply perform XORs:

  $$\forall i \in [0, r-1], \forall j \in [0,3], \forall k \in [1,3], K_{i+1}[j][k] = K_{i+1}[j][k-1] \oplus K_i[j][k].$$

2.2. AES related-key differential characteristics

To mount related-key attacks, we track differences through the ciphering process, where differences are obtained by applying the XOR operator. We note δA the *differential matrix* obtained by applying the XOR operator on two matrices A and A', and for every row j and column k, $\delta A[j][k] = A[j][k] \oplus$

$A'[j][k]$ is called a *differential byte*. The set of all differential bytes is denoted $diffBytes_l$. Among these differential bytes, some of them pass through S-boxes, and we note $Sboxes_l$ this set. When the key length is $l = 128$, these two sets are defined by:

$$\begin{aligned} diffBytes_{128} &= \{\delta X[j][k], \delta X_r[j][k], \delta K_r[j][k] : j,k \in [0,3]\} \\ &\cup \{\delta K_i[j][k], \delta SK_i[j][3], \delta X_i[j][k], \delta SX_i[j][k], \\ &\quad \delta Y_i[j][k], \delta Z_i[j][k] : i \in [0,r-1], j,k \in [0,3]\} \\ Sboxes_{128} &= \{\delta K_i[j][3], \delta X_i[j][k] : i \in [0,r-1], j,k \in [0,3]\} \end{aligned}$$

See Appendix for the definition of $diffBytes_l$ and $Sboxes_l$ when $l \in \{192, 256\}$.

The goal is to find an optimal related-key differential characteristics, *i.e.*, a byte value for every differential byte in $diffBytes_l$ such that the probability of observing δX_r given δX and δK_0 is maximal.

This problem would be easy to solve (and the probability would be equal to 1) if every operation applied during the ciphering process were linear. However, AES is composed of three linear operations (SR, MC, and ARK), and one non linear operation (SB). Moreover, the key schedule KS combines linear XOR operations with the non linear SB operation.

Propagation of differences by the linear operations. The linear operators only move differences to other places, and we can deterministically compute the output difference of a linear operator given its input difference. For SR and MC, given two matrices A_1 and A_2, we have $SR(A_1) \oplus SR(A_2) = SR(A_1 \oplus A_2)$ and $MC(A_1) \oplus MC(A_2) = MC(A_1 \oplus A_2)$. This implies that the output difference of SR is $\delta Y_i = SR(\delta SX_i)$ when the input difference is δSX_i, and the output difference of MC is $\delta Z_i = MC(\delta Y_i)$ when the input difference is δY_i. Similarly, given four matrices A_1, A_2, A_3, and A_4, we have $ARK(A_1, A_2) \oplus ARK(A_3, A_4) = ARK(A_1 \oplus A_3, A_2 \oplus A_4)$. Therefore, the output difference of ARK is $\delta X_{i+1} = ARK(\delta Z_i, \delta K_{i+1})$ when the input differences are δZ_i and δK_{i+1}.

Propagation of differences by the non-linear operation SB. This property no longer holds for SB which applies to a byte B an S-box transformation $S(B)$ such that, given two bytes B and B', $S(B \oplus B')$ is not necessarily equal to $S(B) \oplus S(B')$. Therefore, given an input differential byte δB_{in}, we cannot deterministically compute the output difference after passing through S-boxes. We can only compute probabilities. More precisely, for every couple of differential bytes $(\delta B_{in}, \delta B_{out}) \in [0, 255]^2$, we can compute the probability

that the input difference δB_{in} becomes the output difference δB_{out}, which is the proportion of byte couples (B, B') such that $\delta B_{in} = B \oplus B'$ and $\delta B_{out} = S(B) \oplus S(B')$. More precisely, this probability is denoted $p_S(\delta B_{out}|\delta B_{in})$ and is defined by

$$p_S(\delta B_{out}|\delta B_{in}) = \frac{\#\{(B, B') \in [0, 255]^2 \mid (B \oplus B' = \delta B_{in}) \wedge (S(B) \oplus S(B') = \delta B_{out})\}}{256}$$

For the AES S-box, most of the times the probability p_S is equal to $\frac{0}{256}$ or $\frac{2}{256}$, and rarely to $\frac{4}{256}$ [DR13]. The only case where p_S is equal to 1 is when there is no difference, *i.e.*, $p_S(0|0) = 1$: There is no difference in the output ($\delta B_{out} = 0$) if and only if there is no difference in the input ($\delta B_{in} = 0$), since the S-Box is bijective.

Probability of a valuation of differential bytes. To compute the probability of a given valuation of all differential bytes, we first have to check that all linear operators are satisfied by the valuation. If this is not the case, then the probability is equal to 0. Otherwise it is equal to the product of the transition probabilities of all differential bytes that pass through S-boxes, *i.e.*,

$$p = \prod_{\delta B \in Sboxes_l} p_S(\delta SB|\delta B) \tag{1}$$

We refer the reader to [BS91] for more details on differential characteristics.

2.3. Two step solving process

The three approaches described in [BN10], [FJP13], and [GMS16] compute optimal differential characteristics in two steps.

In a first step, each differential byte $\delta B \in \mathit{diffBytes}_l$ is abstracted with a Boolean variable ΔB such that $\Delta B = 0 \Leftrightarrow \delta B = 0$ and $\Delta B = 1 \Leftrightarrow \delta B \in [1, 255]$. In other words, each Boolean variable assigned to 1 gives the position of a difference. The goal is to find all truncated differential characteristics (*i.e.*, all Boolean solutions) that minimize the number of differences passing through S-boxes (*i.e.*, that minimize $\sum_{\delta B \in Sboxes_l} \Delta B$). We say that an S-box $\delta B \in Sboxes_l$ is *active* whenever there is a difference passing through it, *i.e.*, $\Delta B = 1$.

In a second step, for each truncated differential characteristic, we search for actual byte values that satisfy the AES operations and that maximize the probability p defined in Eq.(1). When a Boolean variable ΔB is equal to 0 in the truncated differential characteristic, there is only one possible

Algorithm 1: Computation of optimal differential characteristics

Input: The size l of the key and the number r of rounds
Output: An optimal differential characteristic c^*

```
1  begin
2      v* ← Step1-opt(l, r)
3      v ← v*
4      c* ← null
5      repeat
6          T ← Step1-enum(l, r, v)
7          for each truncated differential characteristic t ∈ T do
8              c ← Step2(l, r, t)
9              if c ≠ null and (c* = null or p(c) > p(c*)) then c* ← c;
10         v ← v + 1
11     until c* ≠ null and p(c*) ≥ 2^{-6v};
12     return c*
```

value for δB, which is 0. However, when $\Delta B = 1$, there are 255 possible values for δB. Note that some truncated differential characteristics are not valid and cannot be transformed into byte solutions because truncated differential characteristics only satisfy Boolean abstractions of the actual AES operations. These characteristics are said to be *byte-inconsistent*.

The complete procedure to find optimal differential characteristics is described in Algorithm 1. It first calls function *Step1-opt* to compute the minimal number v^* of active S-boxes in a truncated differential characteristic (line 2), and it initializes v to this minimal number of active S-boxes. Then, it calls function *Step1-enum* to compute the set T of all truncated differential characteristics such that the number of active S-boxes is equal to v (line 6). For each truncated differential characteristic $t \in T$, it calls function *Step2* (line 8): if t is not byte-consistent, *Step2* returns *null*; otherwise, it returns the optimal differential characteristic c associated with t. If this optimal differential characteristic has a greater probability than c^*, then it updates c^* (line 9). Hence, when exiting from the loop lines 7-9, if c^* is equal to *null* (because all truncated differential characteristics in T are byte-inconsistent), we have to increment v and iterate again on lines 6 to 10; otherwise, $p(c^*)$ is the largest probability with v active S-boxes and we have $2^{-7v} \leq p(c^*) \leq 2^{-6v}$ (because each S-box has a probability which is equal to 2^{-7} or 2^{-6}). However, it may be possible that $p(c^*)$ is not maximal: there may exist a solution with

higher probability with $v+1$ active S-boxes if $p(c^*) < 2^{-6(v+1)}$. In this case, we have to increment v and iterate again on lines 6 to 10 to check whether there exists a higher probability with one more active S-box.

In this paper, we describe CP models for implementing functions *Step1-opt*, *Step1-enum*, and *Step2*. *Step1-opt* and *Step1-enum* use the same model which is described in Sections 3 and 4. The model of *Step2* is described in Section 5. In Section 6, we introduce a new decomposition in two steps that speeds-up the solution process.

2.4. From differential characteristics to related-key attacks

Let us first clarify the relation between the differential characteristic $c*$ computed by Algo. 1 and an optimal differential. What cryptanalysts would like to have is an optimal differential, *i.e.*, input and output differences that maximize the probability of observing the output difference given the input difference. Differential characteristics have been introduced to simplify the problem by specifying all intermediate differences (after each step of the ciphering process). Several differential characteristics may have the same input and output differences and only differ on intermediate differences. Hence, the probability of a differential is the sum of the probabilities of all differential characteristics that share its input and output differences. Thus, finding the differential characteristic with the best probability gives us a lower bound on the expected differential probability. This notion of expected differential probability evaluates the average behavior of the differential taken on average on all the key space, assuming the independence of intermediate probabilities, the stochastic equivalence of the keys and other simplifying assumptions such as those coming from the related-key setting, for example.

In summary, the probability of the differential characteristic computed by Algo. 1 (denoted p) is an approximation for a lower bound of the expected differential probability.

Let us now explain how the differences δX, δK, and δX_r of the differential characteristic $c*$ computed by Algo. 1 may be used to construct a so-called distinguisher. To this aim, let us assume that we can query an oracle: given a plaintext X, this oracle returns $C = Enc_K(X)$ such that C is either the result of ciphering X by r rounds of the AES with an unknown and randomly chosen key K, or it is a random permutation. The attacker randomly chooses a set $S = \{X^1, \ldots, X^m\}$ of m plaintexts. For each plaintext $X^i \in S$, she asks the oracle to compute $C^i = Enc_K(X^i)$ and $C^{i\prime} = Enc_{K'}(X^{i\prime})$ where $X^{i\prime} = X^i \oplus \delta X$, and $K' = K \oplus \delta K$. If the oracle implements the AES,

then she can expect to find a pair $(X^i, X^{i\prime})$ such that $C^i \oplus C^{i\prime} = \delta X_r$ after trying roughly $1/p$ random input pairs. As p is much larger than 2^{-n}, this is much sooner than expected for a randomly selected permutation. In other words, the differential characteristic $c*$ allows the attacker to know if the Enc function of the oracle is r rounds of the AES or a random permutation: we have built a distinguisher on r rounds, *i.e.*, we have exhibited a particular property that distinguishes a random permutation from the AES with a complexity equal to $\mathcal{O}(1/p)$.

Finally, this distinguisher may be used to mount a related-key differential attack on $r + 1$ rounds. This last step is difficult to explain in a few lines. The basic idea is to add an extra round at the end of the r rounds and to recover some key bits of the subkey K_{r+1} by exploiting the differences in the ciphertexts. We refer the reader to [BS91] for more details on related-key attacks.

2.5. Reminders on Constraint Programming

We briefly recall basic principles of CP and we refer the reader to [RBW06] for more details.

CP is used to solve Constraint Satisfaction Problems (CSPs). A CSP is defined by a triple (X, D, C) such that X is a finite set of variables, D is a function that maps every variable $x_i \in X$ to its domain $D(x_i)$ (that is, the finite set of values that may be assigned to x_i), and C is a set of constraints (that is, relations between some variables which restrict the set of values that may be assigned simultaneously to these variables).

Constraints may be defined in extension, by listing all allowed (or forbidden) tuples of the relation, or in intention, by using mathematical operators. Let us consider for example a CSP with $X = \{x_1, x_2, x_3\}$ such that $D(x_1) = D(x_2) = D(x_3) = \{0, 1\}$, and let us consider a constraint that ensures that the sum of the variables in X is different from 1. This constraint may be defined by a table constraint that enumerates all allowed tuples:

$$(x_1, x_2, x_3) \in \{(0, 0, 0), (0, 1, 1), (1, 0, 1), (1, 1, 0), (1, 1, 1)\}.$$

Conversely, it may be defined by enumerating all forbidden tuples:

$$(x_1, x_2, x_3) \notin \{(1, 0, 0), (0, 1, 0), (0, 0, 1)\}.$$

Finally, it may be defined by using arithmetic operators: $x_1 + x_2 + x_3 \neq 1$.

Solving a CSP involves assigning values to variables such that all constraints are satisfied. More formally, an *assignment* $\mathcal{A}$ is a function which maps each variable $x_i \in X$ to a value $\mathcal{A}(x_i) \in D(x_i)$. An assignment $\mathcal{A}$ *satisfies* (resp. *violates*) a constraint $c \in C$ if the tuple defined by the values assigned to the variables of c in $\mathcal{A}$ belongs (resp. does not belong) to the relation defined by c. An assignment is *consistent* (resp. *inconsistent*) if it satisfies all the constraints (resp. violates some constraints) of the CSP. A *solution of a CSP* is a consistent assignment.

An objective function may be added to a CSP, thus defining a Constrained Optimization Problem (COP). This objective function is defined on some variables of X and the goal is to find the solution that optimizes (minimizes or maximizes) the objective function.

CP languages provide high-level features to define CSPs and COPs in a declarative way. Then, these problems are solved by generic constraint solvers which are usually based on a systematic exploration of the search space: Starting from an empty assignment, they incrementally extend a partial consistent assignment by choosing a non-assigned variable and a consistent value for it until either the current assignment is complete (a solution has been found) or the current assignment cannot be extended without violating constraints (the search must backtrack to a previous choice point and try another extension). To reduce the search space, this exhaustive exploration of the search space is combined with constraint propagation techniques: Each time a variable is assigned to a value, constraints are propagated to filter the domains of the variables that are not yet assigned, *i.e.*, to remove values that are not consistent with respect to the current assignment. When constraint propagation removes all values from a domain, the search must backtrack.

Different levels of constraint propagation may be considered; some enforce stronger consistencies than others, *i.e.*, they remove more values from the domains. A widely used propagation level enforces *Generalized Arc Consistency (GAC)*: a constraint c is GAC if for each variable x_i of c and each value $v_i \in D(x_i)$, there exists a tuple t in the cartesian product of the domains of the variables of c such that t satisfies c and x_i is assigned to v_i in t (t is called a *support* of $x_i = v_i$).

Let us consider for example the constraint $x_1 + x_2 + x_3 \neq 1$. If $D(x_1) = D(x_2) = \{0\}$ and $D(x_3) = \{0, 1\}$, then the constraint is not GAC because there is no support of $x_3 = 1$ (the only tuple in $D(x_1) \times D(x_2) \times D(x_3)$ that satisfies the constraint is $(0, 0, 0)$). Hence, to enforce GAC, we must remove 1 from $D(x_3)$.

A key point to speed up the solving process of CSPs is to define the order in which variables are assigned, and the order in which values are assigned to these variables, when building the search tree. CP languages allow the user to specify this through variable and value ordering heuristics.

3. Basic CP model for Step 1

Functions *Step1-opt* and *Step1-enum* used in Algorithm 1 for computing optimal differential characteristics share the same CP model. The only difference is in the goal of the solving process: in *Step1-opt* the goal is to search for a solution that optimizes a given variable called obj_{Step1}, whereas in *Step1-enum* the goal is to enumerate all solutions when the variable obj_{Step1} is assigned to a given value.

In this section, we describe a first CP model for *Step1-opt* and *Step1-enum*, called CP_{Basic}, which has been introduced in [MSR14] and is derived in a straightforward way from the definition of the AES operations.

3.1. Variables of CP_{Basic}

CP_{Basic} does not associate a Boolean variable with every differential byte $\delta B \in diffBytes_l$. Indeed, during Step 2, the initial differential plaintext δX, the last round differential subkey δK_r, and the final differential ciphertext δX_r can be deterministically computed given the values of all other differential bytes:

- δX is obtained by XORing δX_0 and δK_0.
- δK_r is obtained by applying the key schedule to δK_{r-1} for AES-128, and to δK_{r-1} and δK_{r-2} for AES-192 and AES-256. Note that for the bytes δB that pass through S-boxes during this last round, we deterministically choose for δSB the value that maximizes $p_S(\delta SB|\delta B)$.
- δX_r is obtained by XORing δZ_{r-1} and δK_r.

Hence, CP_{Basic} associates a Boolean variable ΔB with every differential byte $\delta B \in diffBytes_l \setminus \{\delta X[j][k], \delta K_r[j][k], \delta X_r[j][k] : j, k \in [0, 3]\}$. Each Boolean variable ΔB is assigned to 0 if $\delta B = 0$, and to 1 otherwise.

We also define an integer variable obj_{Step1} which corresponds to the number of active S-boxes. The domain of this variable is $D(obj_{Step1}) = [1, \frac{l}{6}]$. Indeed, the smallest possible value is 1 because we need to have at least one

active S-box to have a differential characteristic (we forbid the obvious solution such that δX and δK only contain bytes set to 0, meaning that there is no difference in the initial plaintext and key). The largest possible value is $\frac{l}{6}$ because the highest probability $p_S(\delta_{out}|\delta_{in})$ to pass through the AES S-box is $2^{-6} = \frac{4}{256}$ when $\delta_{in} \neq 0$. Therefore, the greatest probability of a differential characteristic with obj_{Step1} active S-boxes is $2^{-6*obj_{Step1}}$. As we want a differential characteristic which is more efficient than the key exhaustive search, its probability must be greater than 2^{-l} (as explained in Section 2.4) and, therefore, obj_{Step1} must not be larger than $\frac{l}{6}$.

3.2. Definition of a XOR *constraint*

As many AES transformations use the XOR operator, we define a constraint to model it at the Boolean level.

During Step 2, when reasoning at the byte level, the XOR operator is applied on each bit of each byte using the following table:

x	y	$x \oplus y$
0	0	0
0	1	1
1	0	1
1	1	0

However, during Step 1, each byte is abstracted by a Boolean value indicating whether the byte is equal to 0 or not, and we must consider an abstraction of this operator. More precisely, let us consider three differential bytes δB_1, δB_2 and δB_3 such that $\delta B_1 \oplus \delta B_2 = \delta B_3$ (or equivalently, $\delta B_1 \oplus \delta B_2 \oplus \delta B_3 = 0$). If two of these bytes are equal to 0, then we know for sure that the third one is also equal to 0 (because $0 \oplus 0 = 0$). Also, if one differential byte is equal to 0 and a second one is different from 0, then we know for sure that the third one is different from 0 (because $0 \oplus \delta B_i = \delta B_i$). However, when two differential bytes are different from 0, then we cannot know if the third one is equal to 0 or not (because if the two bytes have the same value then the third one is equal to 0, whereas if they have different values the third one is different from 0). Hence, when abstracting differential bytes δB_1, δB_2 and δB_3 with Boolean variables ΔB_1, ΔB_2 and ΔB_3 (which only model the fact that there is a difference or not), we obtain the following definition of the XOR constraint:

$$XOR(\Delta B_1, \Delta B_2, \Delta B_3) \Leftrightarrow \Delta B_1 + \Delta B_2 + \Delta B_3 \neq 1$$

(C_1) $obj_{Step1} = \sum_{\delta B \in Sboxes_l} \Delta B$

(C_2) $\forall \delta B \in Sboxes_l, \Delta SB = \Delta B$

(C_3) $\forall i \in [0, r-2], \forall j, k \in [0, 3], XOR(\Delta Z_i[j][k], \Delta K_{i+1}[j][k], \Delta X_{i+1}[j][k])$

(C_4) $\forall i \in [0, r-1], \forall j, k \in [0, 3], \Delta Y_i[j][k] = \Delta SX_i[j][(j+k)\%4]$

(C_5) $\forall i \in [0, r-2], \forall k \in [0, 3], \left(\sum_{j=0}^{3} \Delta Y_i[j][k] + \Delta Z_i[j][k]\right) \in \{0, 5, 6, 7, 8\}$

(C_6) $\forall j, k \in [0, 3], \Delta Z_{r-1}[j][k] = \Delta Y_{r-1}[j][k]$

(C_7) $\forall i \in [0, r-1], \forall j \in [0, 3],$
$XOR(\Delta K_{i+1}[j][0], \Delta K_i[j][0], \Delta SK_i[(j+1)\%4][3])$

(C_8) $\forall i \in [0, r-1], \forall j \in [0, 3], \forall k \in [1, 3],$
$XOR(\Delta K_{i+1}[j][k], \Delta K_{i+1}[j][k-1], \Delta K_i[j][k])$

Figure 2: Constraints of the CP_{Basic} model for Step 1 of AES-128

where + stands for the integer addition. In other words, either the three variables are false (no difference), or at least two variables are true (at least two differences).

3.3. Constraints of CP_{Basic}

The constraints of CP_{Basic} are listed in Fig. 2 and are described below.

Number of active S-boxes. Constraint (C_1) ensures that obj_{Step1} is equal to the number of active S-boxes, *i.e.*, the number of ΔB variables that are associated with differential bytes in $Sboxes_l$ and that are assigned to 1.

SubBytes. Constraint (C_2) ensures that there is a difference in the output differential byte of an S-box iff there is a difference in the input differential byte. Indeed, SB has no effect on the presence/absence of differences during Step 1 because the S-box is bijective and $(B \oplus B' \neq 0) \Leftrightarrow (S(B) \oplus S(B') \neq 0)$.

AddRoundKey. ARK is modeled by the XOR constraint (C_3).

ShiftRows. SR is modeled by the equality constraint (C_4) that simply links the shifted bytes.

MixColumns. MC cannot be modeled at the Boolean level, as knowing where differences hold in Y_i is not enough to determine where they hold in Z_i. However, the MDS property is modeled by constraint (C_5): It ensures that the number of differences in a same column of Y_i and Z_i is equal to 0 or greater than 4. Constraint (C_6) models the fact that MC is not applied during the last round.

KeySchedule. KS is modeled by XOR constraints (combined with a rotation of the bytes of SK for some columns). When $l = 128$, this corresponds to constraints (C_7) and (C_8). The constant c_i which is XORed with $SK_i[1][3]$ and $K_i[0][0]$ when defining $K_{i+1}[0][0]$ does not appear in $(C7)$: it is canceled when XORing $K_{i+1}[0][0]$ with $K'_{i+1}[0][0]$ to define $\delta K_{i+1}[0][0]$. Indeed, we have:

$$\begin{aligned}\delta K_{i+1}[0][0] &= K_{i+1}[0][0] \oplus K'_{i+1}[0][0] \\ &= SK_i[1][3] \oplus K_i[0][0] \oplus c_i \oplus SK'_i[1][3] \oplus K'_i[0][0] \oplus c_i \\ &= \delta SK_i[1][3] \oplus \delta K_i[0][0]\end{aligned}$$

See Appendix for the definition of the KS constraints when $l \in \{192, 256\}$.

3.4. Goal

The same model is used to solve two problems, *i.e.*, *Step1-opt* and *Step1-enum*. The difference between these problems is in their goals:

- For *Step1-opt*, the goal is to find the minimum number of active S-boxes, *i.e.*, minimize obj_{Step1};
- For *Step1-enum*, the goal is to enumerate all solutions when obj_{Step1} is assigned to a given value v. Each solution corresponds to a truncated differential characteristic with v active S-boxes.

3.5. Ordering heuristics

As the goal is to minimize the number of active S-boxes (for *Step1-opt*) or enumerate all solutions with a small number of active S-boxes (for *Step1-enum*), we define ordering heuristics as follows: first assign variables associated with bytes that pass through S-boxes (those in $Sboxes_l$), and first try to assign them to 0.

3.6. Performance of CP_{Basic}

CP_{Basic} is complete in the sense that for any solution at the byte level (on δ variables), there exists a solution of CP_{Basic} at the Boolean level (on Δ variables). However, experiments reported in [GMS16] have shown us that there is a huge number of solutions of CP_{Basic} which are byte inconsistent and do not correspond to solutions at the byte level. For example, for AES-128, when the number of rounds is $r = 3$, the optimal solution of *Step1-opt* has $obj_{Step1} = 3$ active S-boxes, and *Step1-enum* enumerates more than five hundred truncated differential characteristics with three active S-boxes. However, none of them is byte-consistent. Actually, the optimal byte-consistent truncated differential characteristic has five active S-boxes. In this case, most of the solving time is spent at generating useless truncated differential characteristics which are discarded in Step 2.

4. CP_{XOR}: New CP model for Step 1

In [GMS16], we have described another model for Step 1, called CP_{EQ}, that drastically reduces the number of inconsistent truncated differential characteristics for AES-128. In this section, we describe a new model for Step 1, called CP_{XOR}, and we show that it is more efficient than CP_{EQ}.

A weakness of CP_{Basic} comes from the fact that the XOR constraint between Boolean variables is a poor abstraction of the XOR relation at the byte level, because whenever two XORed bytes are different from zero, we cannot know whether the result is equal to zero or not. In Section 4.1, we show how to tighten this abstraction by generating new XOR equations, obtained by combining initial equations coming from the key schedule. In Section 4.2, we describe the variables of CP_{XOR}, and we introduce new Boolean variables that model differences at the byte level. In Section 4.3, we describe the constraints of CP_{XOR}, and we show how to tighten the definition of the AES operations at the Boolean level by reasoning on differences at the byte level. In Section 4.4 we discuss some implementation issues and in Section 4.5, we experimentally compare CP_{XOR} with CP_{EQ}.

4.1. Generation of new XORs

Every subkey differential byte $\delta K_i[j][k]$ either comes from the initial differential key δK, or is obtained by XORing two differential bytes according to the key schedule rules. Hence, the whole key schedule implies a set of $16*(r-1)$ (resp. $16*(r-1)-8$ and $16*(r-2)$) XOR equations for AES-128

(resp. AES-192 and AES-256), where each of these equations involves three differential bytes. We propose to combine these initial equations to infer new equations.

Let us consider, for example, the three equations that define $\delta K_1[0][3]$, $\delta K_2[0][2]$ and $\delta K_2[0][3]$, respectively, for AES-128:

$$\begin{aligned} \delta K_0[0][3] \oplus \delta K_1[0][2] \oplus \delta K_1[0][3] &= 0 \quad (2) \\ \delta K_1[0][2] \oplus \delta K_2[0][1] \oplus \delta K_2[0][2] &= 0 \quad (3) \\ \delta K_1[0][3] \oplus \delta K_2[0][2] \oplus \delta K_2[0][3] &= 0 \quad (4) \end{aligned}$$

These equations share bytes: $\delta K_1[0][2]$ for Eq. (2) and (3), $\delta K_1[0][3]$ for Eq. (2) and (4), and $\delta K_2[0][2]$ for Eq. (3) and (4). We may combine Eq. (2), (3), and (4) by XORing them, and exploit the fact that $B \oplus B = 0$ for any byte B, to generate the following equation:

$$\delta K_0[0][3] \oplus \delta K_2[0][1] \oplus \delta K_2[0][3] = 0 \quad (5)$$

This new equation is redundant at the byte level, as (2) $\wedge$ (3) $\wedge$ (4) $\Rightarrow$ (5). However, at the Boolean level, the propagation of the XOR constraint corresponding to Eq. (5) detects inconsistencies which are not detected when only considering the XOR constraints corresponding to Eq. (2), (3), and (4). Let us consider, for example, the case where $\Delta K_0[0][3] = 1$, $\Delta K_2[0][1] = \Delta K_2[0][3] = 0$, and all other Boolean variables still have 0 and 1 in their domains. In this case, the propagation of XOR constraints associated with Eq. (2), (3), and (4) does not detect an inconsistency as only one variable is assigned for each constraint, and for the two other variables, we can always choose values such that the sum is different from 1. However, at the byte level, $\delta K_0[0][3]$ cannot be assigned to a value different from 0 when $\delta K_2[0][1] = \delta K_2[0][3] = 0$. Indeed, in this case, Eq. (3) and (4) imply:

$$\begin{aligned} & \delta K_1[0][2] \oplus \delta K_2[0][2] = \delta K_1[0][3] \oplus \delta K_2[0][2] = 0 \\ \Rightarrow \quad & \delta K_1[0][2] = \delta K_2[0][2] = \delta K_1[0][3] \\ \Rightarrow \quad & \delta K_0[0][3] \oplus \delta K_1[0][2] \oplus \delta K_1[0][3] = \delta K_0[0][3] \end{aligned}$$

As a consequence, if $\delta K_0[0][3] \neq 0$, we cannot satisfy Eq. (2). This inconsistency is detected when propagating the XOR constraint associated with Eq. (5), as it ensures that $\Delta K_0[0][3] + \Delta K_2[0][1] + \Delta K_2[0][3] \neq 1$.

Hence, we propose to combine XOR equations of the key schedule to generate new equations. More precisely, given two equations $\delta B_1 \oplus \ldots \oplus \delta B_n = 0$

and $\delta B'_1 \oplus \ldots \oplus \delta B'_m = 0$ such that $\{B_1, \ldots, B_n\} \cap \{B'_1, \ldots, B'_m\} \neq \emptyset$, we generate the equation:

$$\bigoplus_{B \in \{B_1,\ldots,B_n\} \cup \{B'_1,\ldots,B'_m\} \setminus \{B_1,\ldots,B_n\} \cap \{B'_1,\ldots,B'_m\}} \delta B = 0.$$

This new equation is recursively combined with existing ones to generate other equations until no more equation can be generated.

For example, from Eq. (2), (3), and (4), we generate the following equations:

$$\text{From (2) and (3):} \quad \delta K_0[0][3] \oplus \delta K_1[0][3] \oplus \delta K_2[0][1] \oplus \delta K_2[0][2] = 0 \quad (6)$$

$$\text{From (2) and (4):} \quad \delta K_0[0][3] \oplus \delta K_1[0][2] \oplus \delta K_2[0][2] \oplus \delta K_2[0][3] = 0 \quad (7)$$

$$\text{From (3) and (4):} \quad \delta K_1[0][2] \oplus \delta K_1[0][3] \oplus \delta K_2[0][1] \oplus \delta K_2[0][3] = 0 \quad (8)$$

Then, from Eq. (2) and (8) (or, equivalently, from Eq. (3) and (7) or from Eq. (4) and (6)), we generate Eq. (5). As no new equation can be generated from Eq. (2), (3), (4), (5), (6), (7), and, (8), the process stops.

The number of new equations that may be generated grows exponentially with respect to the number r of rounds. For example, for AES-128, when $r = 3$ (resp. $r = 4$), the total number of new equations that may be generated is 988 (resp. 16332). When further increasing r to 5, the number of new equations becomes so large that we cannot generate them within a time limit of one hour.

To avoid this combinatorial explosion, we only generate equations that involve at most four differential bytes. Indeed, the basic Boolean XOR constraint associated with an equation simply states that the sum of the Boolean variables must be different from 1. In Section 4.3.1, we show how to strengthen this constraint by reasoning on differences at the byte level when the XOR constraint involves no more than four variables. In this case, each XOR constraint is very efficiently handled by simply channeling three pairs of Boolean variables. Preliminary experiments have shown us that these strengthened XOR constraints both reduce the number of choice points and speed-up the solution process, and that further adding XOR constraints for equations that involve more than four variables (to forbid that their sum is equal to one) does not significantly reduce the number of choice points and often increases time.

For AES-128 (resp. AES-192 and AES-256) with $r = 10$ (resp. $r = 12$ and $r = 14$) rounds, the number of initial equations coming from the key schedule

is 144 (resp. 168 and 192). From these initial equations, we generate 122 (resp. 168 and 144) new equations that involve three differential bytes, and 1104 (resp. 1696 and 1256) new equations that involve four differential bytes.

The algorithm that generates equations has been implemented in Picat [ZKF15], and the time spent by Picat to search for all equations of length smaller than or equal to four is always smaller than 0.1 seconds. This time is negligible compared to the time needed to solve Step 1.

We note $xorEq_l$ the set of all equations (both initial and generated equations) coming from the key schedule when the key length is l.

Note that $xorEq_l$ actually contains all possible equations with at most four differential bytes implied by the key schedule algorithm. In other words, our procedure does not miss implied equations even if it does not allow the generation of intermediate equations of length greater than 4. We have checked this by exhaustively generating all possible equations with at most four differential key bytes and, for each equation that does not belong to $xorEq_l$, we have proven that it is not a logical consequence of the initial set of equations of the key schedule (by demonstrating that $xorEq_l$ is consistent with the negation of the equation). The whole checking procedure (performed by a program written in C) for all possible equations, is done in a few minutes for the three possible key lengths.

4.2. Variables of CP_{XOR}

All variables of CP_{Basic} are also variables of CP_{XOR}.

We introduce new Boolean variables that are used to tighten the Boolean abstraction by reasoning on differences at the byte level: given two differential bytes δB_1 and δB_2, the Boolean variable $diff_{\delta B_1, \delta B_2}$ is equal to 1 if $\delta B_1 \neq \delta B_2$, and to 0 otherwise. We do not define a *diff* variable for every couple of differential bytes in $diffBytes_l$, but restrict our attention to couples of bytes for which it is useful to know whether they are equal or not.

More precisely, we consider three separate sets of differential bytes and we only compare differential bytes that belong to a same set.

- The first set, called DK, contains bytes coming from δK and δSK matrices. The *diff* variables associated with these bytes are used to tighten the constraints associated with the equations of $xorEq_l$, as explained in Section 4.3.1.

- The second and third sets, called DY and DZ, contain bytes coming from δY and δZ matrices, respectively. The *diff* variables associated

with these bytes are used to propagate the MDS property of MixColumns at the byte level, as explained in Section 4.3.2.

For each of these three sets, we consider a separate subset for each row $j \in [0, 3]$:

- For DK, we know that every initial XOR equation due to the key schedule either involves three bytes on a same row j (*i.e.*, $\Delta K_{i+1}[j][k]$, $\Delta K_{i+1}[j][k-1]$, and $\Delta K_i[j][k]$), or it involves two bytes on a same row j (*i.e.*, $\Delta K_i[j][0]$, and $\Delta K_{i+1}[j][0]$), and a byte that has just passed through an S-box on the next row ($(j+1)\%4$) (*i.e.*, $\Delta SK_i[(j+1)\%4][3]$). As we cannot know if the input and output differences of S-boxes are equal or not (*i.e.*, if $\delta K_i[(j+1)\%4][3]$ is equal to $\delta SK_i[(j+1)\%4][3]$ or not), we can limit *diff* variables to couples of differential bytes that occur on a same row of δK matrices, or on two consecutive rows of δK and δSK matrices.

- For DY and DZ, the MDS property implies relations between columns of DY and DZ and, in Section 4.3.2, we use a generalization of this property that XORs bytes of a same row for different columns. As these XORs are only performed on bytes that occur in a same row, we can limit *diff* variables to couples of differential bytes that occur in a same row of δY matrices (for DY) and δZ matrices (for DZ).

Hence, for each row $j \in [0, 3]$, we define the three following sets:

$$\begin{aligned} DK_j &= \{\delta K_i[j][k], \delta SK_i[(j+1)\%4][3] : i \in [1, r], k \in [0, 3]\} \\ DY_j &= \{\delta Y_i[j][k] : i \in [0, r-2], k \in [0, 3]\} \\ DZ_j &= \{\delta Z_i[j][k] : i \in [0, r-2], k \in [0, 3]\}. \end{aligned}$$

Given these sets, we define the *diff* variables as follows: For each set $D \in \{DK_j, DY_j, DZ_j : j \in [0, 3]\}$, and for each pair of differential bytes $\{\delta B_1, \delta B_2\} \subseteq D$, we define a Boolean variable $diff_{\delta B_1, \delta B_2}$.

4.3. Constraints of CP_{XOR}

The constraints of CP_{XOR} are listed in Fig. 3. Constraints (C'_1) to (C'_6) are identical to Constraints (C_1) to (C_6) of CP_{Basic}. Constraints (C'_2), (C'_4), (C'_6), (C'_7), (C'_{10}), and (C'_{11}) are equalities of the form $\Delta_i = \Delta_j$: they allow us to rename variables in order to simplify the description of the model without changing the performance of the solvers.

(C'_1) $obj_{Step1} = \sum_{\delta B \in Sboxes_l} \Delta B$

(C'_2) $\forall \delta B \in Sboxes_l, \Delta SB = \Delta B$

(C'_3) $\forall i \in [0, r-2], \forall j, k \in [0,3], XOR(\Delta Z_i[j][k], \Delta K_{i+1}[j][k], \Delta X_{i+1}[j][k])$

(C'_4) $\forall i \in [0, r-1], \forall j, k \in [0,3], \Delta Y_i[j][k] = \Delta SX_i[j][(j+k)\%4]$

(C'_5) $\forall i \in [0, r-2], \forall k \in [0,3], \left(\sum_{j=0}^{3} \Delta Y_i[j][k] + \Delta Z_i[j][k]\right) \in \{0,5,6,7,8\}$

(C'_6) $\forall j, k \in [0,3], \Delta Z_{r-1}[j][k] = \Delta Y_{r-1}[j][k]$

(C'_7) $\forall D \in \{DK_j, DY_j, DZ_j : j \in [0,3]\}, \forall \{\delta B_1, \delta B_2\} \subseteq D,$
$diff_{\delta B_1, \delta B_2} = diff_{\delta B_2, \delta B_1}$

(C'_8) $\forall D \in \{DK_j, DY_j, DZ_j : j \in [0,3]\}, \forall \{\delta B_1, \delta B_2, \delta B_3\} \subseteq D,$
$diff_{\delta B_1, \delta B_2} + diff_{\delta B_2, \delta B_3} + diff_{\delta B_1, \delta B_3} \neq 1$

(C'_9) $\forall D \in \{DK_j, DY_j, DZ_j : j \in [0,3]\}, \forall \{\delta B_1, \delta B_2\} \subseteq D,$
$diff_{\delta B_1, \delta B_2} + \Delta B_1 + \Delta B_2 \neq 1$

(C'_{10}) $\forall (\delta B_1 \oplus \delta B_2 \oplus \delta B_3 = 0) \in xorEq_l,$
$(diff_{\delta B_1, \delta B_2} = \Delta B_3) \wedge (diff_{\delta B_1, \delta B_3} = \Delta B_2) \wedge (diff_{\delta B_2, \delta B_3} = \Delta B_1)$

(C'_{11}) $\forall (\delta B_1 \oplus \delta B_2 \oplus \delta B_3 \oplus \delta B_4 = 0) \in xorEq_l,$
$(diff_{\delta B_1, \delta B_2} = diff_{\delta B_3, \delta B_4}) \wedge (diff_{\delta B_1, \delta B_3} = diff_{\delta B_2, \delta B_4}) \wedge (diff_{\delta B_1, \delta B_4} = diff_{\delta B_2, \delta B_3})$

(C'_{12}) $\forall i_1, i_2 \in [0, r-2], \forall k_1, k_2 \in [0,3],$
$\sum_{j=0}^{3} diff_{\delta Y_{i_1}[j][k_1], \delta Y_{i_2}[j][k_2]} + diff_{\delta Z_{i_1}[j][k_1], \delta Z_{i_2}[j][k_2]} \in \{0,5,6,7,8\}$

(C'_{13}) $\forall i_1, i_2 \in [0, r-2], \forall j, k_1, k_2 \in [0,3],$
$diff_{\delta K_{i_1+1}[j][k_1], \delta K_{i_2+1}[j][k_2]} + diff_{\delta Z_{i_1}[j][k_1], \delta Z_{i_2}[j][k_2]} + \Delta X_{i_1+1}[j][k_1] + \Delta X_{i_2+1}[j][k_2] \neq 1$

Figure 3: Constraints of the CP_{XOR} model for Step 1

Constraints (C'_7) and (C'_8) ensure symmetry and transitivity on *diff* variables. For each set $D \in \{DK_j, DY_j, DZ_j : j \in [0,3]\}$, they ensure that the *diff* variables associated with couples of bytes in D define an equivalence

relation.

Constraint (C'_9) relates *diff* and Δ variables. Indeed, whenever two Boolean variables ΔB_1 and ΔB_2 are equal to 0, then we know for sure that there is no difference at the byte level, *i.e.*, $\delta B_1 = \delta B_2$. Similarly, whenever $\Delta B_1 \neq \Delta B_2$, we know for sure that there is a difference at the byte level, *i.e.*, $\delta B_1 \neq \delta B_2$.

Constraints (C'_{10}) and (C'_{11}) propagate the XOR equations of $xorEq_l$ and are described in Section 4.3.1. Constraint (C'_{12}) propagates the MDS property at the byte level and is described in Section 4.3.2. Constraint (C'_{13}) propagates ARK at the byte level and is described in Section 4.3.3.

4.3.1. Constraints associated with the XOR *equations of* $xorEq_l$

In the basic model, every XOR constraint involves exactly three Boolean variables, and ensures that the sum of these variables is different from 1.

In Section 4.1, we have shown how to generate new XOR equations from initial key schedule equations, and these new equations either involve three or four differential bytes. Hence, we need to extend the XOR constraint for the case where we have four Boolean variables. A straightforward extension is to simply state that the sum of the Boolean variables must be different from 1. Indeed, a XOR equation that involves four differential bytes cannot be satisfied if exactly one of these four differential bytes is different from zero. However, this is a poor abstraction of the relation at the byte level. We propose to tighten Boolean XOR constraints associated with the Key Schedule by exploiting *diff* variables.

Constraint (C'_{10}) corresponds to the case of equations that involve three differential bytes. For each equation $\delta B_1 \oplus \delta B_2 \oplus \delta B_3 = 0$ in $xorEq_l$, it ensures that whenever two differential bytes of $\{\delta B_1, \delta B_2, \delta B_3\}$ have different values, then the third one is different from 0 and therefore its associated Boolean variable is equal to 1. Note that this constraint combined with Constraint (C'_9) ensures that $\Delta B_1 + \Delta B_2 + \Delta B_3 \neq 1$. Indeed, if $\Delta B_1 = 1$ and $\Delta B_2 = \Delta B_3 = 0$, then Constraint ($C'_9$) implies that $diff_{\delta B_2, \delta B_3} = 0$, which is inconsistent with the fact that $diff_{\delta B_2, \delta B_3}$ must be equal to ΔB_1.

Constraint (C'_{11}) corresponds to the case of equations that involve four differential bytes. For each equation $\delta B_1 \oplus \delta B_2 \oplus \delta B_3 \oplus \delta B_4 = 0$ in $xorEq_l$, it ensures that whenever two differential bytes of $\{\delta B_1, \delta B_2, \delta B_3, \delta B_4\}$ are equal then the two other ones must also be equal. Again, this constraint combined with Constraint (C'_9) ensures that $\Delta B_1 + \Delta B_2 + \Delta B_3 + \Delta B_4 \neq 1$. Indeed, if $\Delta B_1 = 1$ and $\Delta B_2 = \Delta B_3 = \Delta B_4 = 0$, then Constraint ($C'_9$) implies that $diff_{\delta B_2, \delta B_3} = 0$ and $diff_{\delta B_1, \delta B_4} = 1$, which is inconsistent with the fact that

$\mathit{diff}_{\delta B_2,\delta B_3}$ must be equal to $\mathit{diff}_{\delta B_1,\delta B_4}$.

4.3.2. Propagation of MDS at the byte level

As seen in Section 2.1 (paragraph "Maximum Distance Separable property"), the MDS property also holds for the result of the XOR of two different columns of δY and δZ. Hence, for all i_1, i_2 in $[0, r-2]$, and for all k_1, k_2 in $[0, 3]$, we have:

$$\sum_{j=0}^{3}(\delta Y_{i_1}[j][k_1] \oplus \delta Y_{i_2}[j][k_2] \neq 0) + (\delta Z_{i_1}[j][k_1] \oplus \delta Z_{i_2}[j][k_2] \neq 0) \in \{0, 5, 6, 7, 8\}.$$

This property is modelled by constraint (C'_{12}).

4.3.3. Propagation of ARK at the byte level

ARK implies the following equations: $\forall i_1, i_2 \in [0, r-2], \forall j, k_1, k_2 \in [0, 3]$,

$$\begin{aligned} \delta K_{i_1+1}[j][k_1] \oplus \delta Z_{i_1}[j][k_1] &= \delta X_{i_1+1}[j][k_1] \\ \delta K_{i_2+1}[j][k_2] \oplus \delta Z_{i_2}[j][k_2] &= \delta X_{i_2+1}[j][k_2]. \end{aligned}$$

By XORing these two equations, we obtain:

$$\begin{aligned} & \delta K_{i_1+1}[j][k_1] \oplus \delta K_{i_2+1}[j][k_2] \\ \oplus \quad & \delta Z_{i_1}[j][k_1] \oplus \delta Z_{i_2}[j][k_2] \\ = \quad & \delta X_{i_1+1}[j][k_1] \oplus \delta X_{i_2+1}[j][k_2]. \end{aligned}$$

From this equation, we infer that it is not possible to have exactly one of the three following inequalities which is true:

$$\begin{aligned} \delta K_{i_1+1}[j][k_1] &\neq \delta K_{i_2+1}[j][k_2] \\ \delta Z_{i_1}[j][k_1] &\neq \delta Z_{i_2}[j][k_2] \\ \delta X_{i_1+1}[j][k_1] &\neq \delta X_{i_2+1}[j][k_2] \end{aligned}$$

Constraint (C'_{13}) ensures this property: The first two inequalities are modeled by $\mathit{diff}_{\delta K_{i_1+1}[j][k_1],\delta K_{i_2+1}[j][k_2]}$ and $\mathit{diff}_{\delta Z_{i_1}[j][k_1],\delta Z_{i_2}[j][k_2]}$; For the third inequality, we exploit the fact that if $\Delta X_{i_1+1}[j][k_1] + \Delta X_{i_2+1}[j][k_2] = 1$ then $\delta X_{i_1+1}[j][k_1] \neq \delta X_{i_2+1}[j][k_2]$.

4.4. Implementation

CP_{XOR} only uses common and widely implemented constraints. Therefore, it may be easily implemented with any existing CP libraries. In order to ease the comparison between different solvers, we have implemented CP_{XOR} with a high-level modeling language called MiniZinc [NSB+07]: MiniZinc models are translated into a simple subset of MiniZinc called FlatZinc, using a compiler provided by MiniZinc, and most existing CP solvers have developed FlatZinc interfaces (currently, there are fifteen CP solvers which have FlatZinc interfaces).

We ran experiments with Gecode [Gec06], Choco [PFL16], Chuffed [CS14], and Picat-SAT [ZKF15]: Gecode and Choco are CP libraries which solve CSPs by combining search with constraint propagation; Chuffed is a lazy clause hybrid solver that combines features of finite domain propagation and Boolean satisfiability; Picat-SAT translates CSPs into Boolean satisfiability formulae, and then uses the SAT solver Lingeling [Bie14] to solve it.

Gecode and Choco are clearly outperformed by Chuffed and Picat-SAT for both *Step1-opt* and *Step1-enum*. This shows us that clause learning is a key ingredient for solving these problems. Picat-SAT and Chuffed have complementary performance. For the optimisation problem *Step1-opt*, Picat-SAT is always the fastest solver. For the enumeration problem *Step1-enum*, Chuffed is faster than Picat-SAT on small instances, *i.e.*, all AES-128 instances, and AES-192 (resp. AES-256) instances when the number of rounds r is lower than 5 (resp. 7). However, Picat-SAT is faster than Chuffed on larger instances and Chuffed is not able to solve AES-192 (resp. AES-256) instances within a 24 hour time limit when $r \geq 9$ (resp. $r \geq 11$).

The good performance of Picat-SAT is not surprising given that most variables are Boolean variables. If we exclude equality constraints, there are only two different kinds of constraints:

- (C'_3), (C'_8), (C'_9), and (C'_{13}) are of the form: $\sum_{\Delta_i \in S} \Delta_i \neq 1$;
- (C'_1), (C'_5) and (C'_{12}) are of the form: $\sum_{\Delta_i \in S} \Delta_i = v$ where v is the integer variable obj_{Step1} for (C'_1) and an integer variable whose domain is $\{0, 5, 6, 7, 8\}$ for (C'_5) and (C'_{12});

where S is a set of Boolean variables. These two kinds of constraints are easily translated into Boolean formulae by combining at-most and at-least encodings which are widely studied in the SAT community [ZK17]. Note that we cannot use the XOR-clauses introduced in CryptoMiniSat [SNC09]

for modeling the XOR constraint in Step 1 because the semantics of the XOR operation is changed when abstracting every byte by a Boolean value.

As CP_{XOR} only uses sum constraints, we can also translate it into an Integer Linear Programming (ILP) model in a very straightforward way, by using an encoding similar to the one introduced in [MWGP12], for example. Each constraint $\sum_{\Delta_i \in S} \Delta_i \neq 1$ is translated into $\#S$ inequalities: for each $\Delta_j \in S$, we add the inequality $-\Delta_j + \sum_{\Delta_i \in S \setminus \{\Delta_j\}} \Delta_i \geq 0$. Each constraint $\sum_{\Delta_i \in S} \Delta_i = v$ such that v is an integer variable whose domain is $\{0, 5, 6, 7, 8\}$ is encoded by introducing a new Boolean variable $x \in \{0, 1\}$ and adding the following inequalities:

$$\sum_{\Delta_i \in S} \Delta_i \geq 5x$$
$$\forall \Delta_i \in S, x \geq \Delta_i$$

When $x = 0$, every Δ_i is constrained to be equal to 0 by $x \geq \Delta_i$; Otherwise, the first inequality ensures that at least 5 variables of S are set to 1. We have used Gurobi [Opt18] to solve this ILP model, and experiments have shown us that it is not competitive with Picat-SAT.

4.5. Comparison of CP_{XOR} with CP_{EQ}

The model introduced in [GMS16], called CP_{EQ}, is an improvement of CP_{Basic}. Like CP_{XOR}, CP_{EQ} exploits equality relationships at the byte level to better propagate the MDS property. Constraints (C'_1) to (C'_9) and Constraints (C'_{12}) and (C'_{13}) are common to CP_{XOR} and CP_{EQ}[1].

However, in CP_{EQ} we do not infer new XOR equations from the initial equations of the key schedule, as explained in Section 4.1, and therefore Constraints (C'_{10}) and (C'_{11}) are not used in CP_{EQ}. Instead of generating new equations, CP_{EQ} uses the key schedule rules to pre-compute, for each differential byte $\delta K_i[j][k]$, a set $V(i, j, k)$ of differential bytes such that $\delta K_i[j][k]$ is equal to the result of XORing all bytes in $V(i, j, k)$. Then, for each differential byte $\delta K_i[j][k]$, a variable $V_1(i, j, k)$ is constrained to be equal to the subset of bytes of $V(i, j, k)$ that are different from 0. These V_1 variables are used to infer that two differential bytes are equal when their corresponding V_1

[1]In [GMS16], CP_{EQ} is defined by using *eq* variables instead of *diff* variables, such that each $diff_{\delta B_1, \delta B_2}$ variable that occurs in a constraint of CP_{XOR} must be replaced by $1 - eq_{\delta B_1, \delta B_2}$ in CP_{EQ}.

variables are equal, and that $\Delta K_i[j][k]$ is equal to 0 (resp. 1) when $V_1(i,j,k)$ is empty (resp. contains only one variable).

In this section, we experimentally compare CP_{XOR} with CP_{EQ}.

Experimental setup. All experiments have been performed on a single core of a server with an Intel Xeon E5-2687W v4 CPU at 3.00GHz.

We compare CP_{XOR} and CP_{EQ} on the two problems described in Section 2.3, *i.e.*, *Step1-opt*, that aims at finding the minimal value of obj_{Step1}, and *Step1-enum*, that aims at enumerating all truncated differential characteristics when the value of obj_{Step1} is fixed to a given value v.

We consider 23 instances denoted AES-l-r where $l \in \{128, 192, 256\}$ is the key length and r is the number of rounds: $r \in [3, 5]$ (resp. $[3, 10]$ and $[3, 14]$) when $l = 128$ (resp. 192 and 256). We do not consider values of r larger than 5 (resp. 10) when $l = 128$ (resp. 192) because for these values the maximal probability becomes smaller than 2^{-l}.

CP_{XOR} and CP_{EQ} are both implemented[2] with MiniZinc [NSB+07], and we report experimental results obtained with Picat-SAT which is the best performing solver among all the considered solvers (as discussed in Section 4.4).

Results for Step1-opt. For each instance, Table 1 reports the optimal value v^* of Obj_{Step1} computed by *Step1-opt*. This optimal value may be different for CP_{XOR} and CP_{EQ} as they consider different abstractions of the KS XOR operations. In practice, for all instances but one, both models find the same optimal value, and for this optimal value there exists at least one byte-consistent truncated differential characteristic (see Table 2) so that the repeat loop (lines 6-10) of Algorithm 1 is executed only once. However, for AES-192-10, the minimal value of obj_{Step1} is equal to 27 with CP_{EQ} whereas it is equal to 29 with CP_{XOR}. As a consequence, if we use CP_{EQ} to solve *Step1-opt*, the repeat loop of Algorithm 1 is executed three times: v is successively assigned to 27, 28, and 29, and for each of these values, we need to solve *Step1-enum* and *Step2*. When $v = 27$ (resp. 28), *Step1-enum* with CP_{EQ} finds 92 (resp. 1436) truncated differential characteristics which are all byte-inconsistent so that *Step2* returns *null* for each of them. When $v = 29$, some truncated differential characteristics are byte-consistent and the repeat loop

[2]These models are available on https://gitlab.inria.fr/source_code/aes-cryptanalysis-cp-xor-2019/.

	Step1-opt				*Step1-enum*			
	CP_{EQ}		CP_{XOR}		CP_{EQ}		CP_{XOR}	
	v^*	t_1^{opt}	v^*	t_1^{opt}	$\#T$	t_1^{enum}	$\#T$	t_1^{enum}
AES-128-3	5	4	5	**3**	4	6	4	**4**
AES-128-4	12	21	12	**14**	8	74	8	**38**
AES-128-5	17	44	17	**33**	1113	32340	1113	**22869**
AES-192-3	1	3	1	**2**	15	16	15	**10**
AES-192-4	4	8	4	**5**	4	12	4	**7**
AES-192-5	5	14	5	**8**	2	13	2	**9**
AES-192-6	10	34	10	**18**	6	65	6	**45**
AES-192-7	13	72	13	**37**	4	98	4	**66**
AES-192-8	18	205	18	**73**	8	752	8	**333**
AES-192-9	24	2527	24	**520**	240	43359	240	**13524**
AES-192-10	27	3715	**29**	**3285**	27548	-	**602**	**216120**
AES-256-3	1	3	1	3	33	39	33	**29**
AES-256-4	3	8	3	**7**	14	38	14	**25**
AES-256-5	3	13	3	**8**	4	21	4	**15**
AES-256-6	5	25	5	**17**	3	29	3	**20**
AES-256-7	5	48	5	**47**	1	22	1	**15**
AES-256-8	10	61	10	**49**	3	76	3	**52**
AES-256-9	15	172	15	**106**	16	705	16	**430**
AES-256-10	16	236	16	**112**	4	385	4	**224**
AES-256-11	20	488	20	**286**	4	705	4	**312**
AES-256-12	20	625	20	**140**	4	1228	4	**463**
AES-256-13	24	1621	24	**822**	4	1910	4	**597**
AES-256-14	24	2179	24	**682**	4	1722	4	**607**

Table 1: Comparison of CP_{EQ} and CP_{XOR} for solving Step 1. For each instance, we display the results with CP_{EQ} and CP_{XOR} for *Step1-opt* (optimal value v^* of obj_{Step1}, and time t_1^{opt} in seconds) and *Step1-enum* (number $\#T$ of truncated differential characteristics when obj_{Step1} is assigned to the value v^* found with CP_{XOR}, and time t_1^{enum} in seconds). We report '-' when the time exceeds two weeks.

is stopped. For this instance, CP_{XOR} is able to infer that there is no byte-consistent truncated differential characteristic with 27 or 28 active S-boxes, and it returns 29, which is the smallest possible value for which there are byte-consistent truncated differential characteristics.

When comparing CPU times, we note that CP_{XOR} is always at least as fast as CP_{EQ}: CP_{XOR} is able to solve every instance but one (AES-192-10) in at most 822 seconds, whereas CP_{EQ} needs up to 2527 seconds to solve these instances. For instance AES-192-10, CP_{XOR} is a bit faster than CP_{EQ} (3285 seconds instead of 3715), while it finds a better solution that has 29 active S-boxes instead of 27.

Results for Step1-enum. In Table 1, we report results for solving *Step1-enum* when v is fixed to the optimal value v^* found when solving *Step1-opt* with CP_{XOR}. CP_{XOR} is always faster than CP_{EQ}, and it is able to solve all instances but three in less than 607 seconds. The three most challenging instances are AES-128-5, AES-192-9, and AES-192-10, which are solved by CP_{XOR} in less than 7 hours, 4 hours and 60 hours, respectively. CP_{EQ} is able to solve AES-128-5 and AES-192-9 in less than 9 and 12 hours, respectively. However, AES-192-10 cannot be solved by CP_{EQ} within two weeks. Actually, for this instance, CP_{XOR} strongly reduces the number of solutions: There are 602 truncated differential characteristics with CP_{XOR} instead of 27548 with CP_{EQ}[3].

Note that reducing the number of truncated differential characteristics is very important to reduce the total solving time as for each characteristic of Step 1, we need to search for an optimal byte solution (or prove that it is not byte-consistent, which is the case of every characteristic found with CP_{EQ} but not with CP_{XOR}).

4.6. Discussion

Experimental results show us that adding new XOR constraints (obtained by combining the initial equations coming from the key schedule) allow us to tighten the Boolean abstraction and speed-up the solution process. Our new model only uses common and widely implemented constraints and it has been implemented in MiniZinc. Therefore, it can be solved by any CP solver that accepts MiniZinc models and it can be easily translated in any constraint-based modeling language. As a counterpart, using this model involves to pre-compute new XOR equations. If this pre-computation step has a negligible time complexity compared to the solving process, it may limit the re-usability of the proposed approach for other cryptanalysis problems.

[3]The number of solutions with CP_{EQ} has been found by decomposing *Step1-enum* into independent sub-problems which have been solved in parallel (see [GLMS17]).

Another possibility to strengthen the CP model would have been to introduce a new global constraint. Global constraints are a key point of CP success: they both ease the modeling step by providing compact ways for declaring constraints, and speed-up the solution process by providing dedicated propagators. In our context, we could replace constraints (C'_{10}) and (C'_{11}) by a single constraint $globalXOR(\{\Delta K_i[j][k] : i \in [0, r], j, k \in [0, 3]\})$. In this case, we must implement a propagator that filters the domains of $\Delta K_i[j][k]$ variables in order to forbid Boolean assignments that do not satisfy the key schedule XOR equations at the byte level. A key point is to find a good compromise between the strength of the filtering and its time-complexity, and this usually involves an important work of design and implementation. This also implies a loss of universality of the model as it can only be solved by solvers for which a propagator dedicated to *globalXOR* has been implemented.

5. CP model for Step 2

Given a truncated differential characteristic computed by *Step1-enum*, Step 2 aims at searching for the byte values with the highest differential probability (or proving that the characteristic is not byte-consistent). In this section, we describe the CP model introduced in [GMS16] for AES-128, extended to AES-192 and AES-256 in a straightforward way.

5.1. Variables of the Step 2 model

For each differential byte $\delta B \in diffByte_l$, we define an integer variable. The domain of each of these variables depends on whether it has a corresponding Boolean variable in the Step 1 model:

- Each differential byte $\delta B \in \{\delta X[j][k], \delta K_r[j][k], \delta X_r[j][k] : j, k \in [0, 3]\}$ has no Boolean counterpart in the Step 1 model (because its value is deterministically inferred from the values of other variables). In this case, the domain is $D(\delta B) = [0, 255]$.

- Each differential byte $\delta B \in diffByte_l \setminus \{\delta X[j][k], \delta K_r[j][k], \delta X_r[j][k] : j, k \in [0, 3]\}$ has a Boolean counterpart ΔB in the Step 1 model. In this case, the domain is $D(\delta B) = \{0\}$ if $\Delta B = 0$, and $D(\delta B) = [1, 255]$ otherwise.

As we look for a byte-consistent solution with maximal probability, we declare an integer variable $P_{\delta B}$ for each differential byte $\delta B \in Sboxes_l$: This variable corresponds to the base 2 logarithm of the probability $p_S(\delta SB|\delta B)$ of obtaining the S-box output difference δSB when the S-box input difference is δB. The domain of $P_{\delta B}$ depends on the value of ΔB in the truncated differential characteristic: If $\Delta B = 0$, then $p_S(0|0) = 1$ and therefore $D(P_{\delta B}) = \{0\}$; otherwise, $p_S(\delta SB|\delta B) \in \{\frac{2}{256}, \frac{4}{256}\}$ and $D(P_{\delta B}) = \{-7, -6\}$ (the constraint associated with the SubBytes operation forbids couples $(\delta B, \delta SB)$ such that $p_S(\delta SB|\delta B) = 0$).

Finally, we introduce an integer variable obj_{Step2} which corresponds to the base 2 logarithm of the probability of the differential characteristic. The domain of obj_{Step2} is derived from the number of differences that pass trough S-boxes in the truncated differential characteristic, *i.e.*, $D(obj_{Step2}) = [-7 \cdot v, -6 \cdot v]$ where $v = \sum_{\delta B \in Sboxes_l} \Delta B$.

5.2. Constraints of the Step 2 model

The constraints basically follow the AES operations to relate variables, as described in Section 3 for Step 1, but consider the definition of the operations at the byte level, instead of the Boolean level.

A main difference is that the SubBytes operation, which has no effect at the Boolean level, must be modeled at the byte level. This is done thanks to a ternary table constraint which extensively lists all triples (X, Y, P) such that there exist two bytes B_1 and B_2 whose difference before and after passing through S-Boxes is equal to X and Y, respectively, and such that P is the probability of this transformation: For all $\delta B \in Sboxes_l$, we add the constraint

$$(\delta B, \delta SB, P_{\delta B}) \in \{(X, Y, P) \quad | \quad \exists (B_1, B_2) \in [0, 255] \times [0, 255], X = B_1 \oplus B_2, \\ Y = S(B_1) \oplus S(B_2), P = \log_2(p_S(Y|X))\}.$$

This table constraint contains $1 + 255 * 127$ tuples. The first tuple is $(0, 0, 0)$ and it corresponds to the case where the input differential byte δB is 0: In this case, the output differential byte δSB is also 0, and $p_S(0|0) = 1$. All other tuples correspond to non null input differences: There are 255 possible non null input differences, and for each of them there are 127 possible output differences (one for which the probability is equal to 2^{-6}, and 126 for which the probability is equal to 2^{-7}).

All other constraints are defined in a rather straightforward way, using table constraints.

5.3. Objective function

The goal is to find a byte-consistent solution with maximal differential probability. As we consider logarithms, this amounts to searching for a solution that maximizes the sum of all $P_{\delta B}$ variables. Hence, we constrain obj_{Step2} to be equal to the sum of all $P_{\delta B}$ variables:

$$obj_{Step2} = \sum_{\delta B \in Sboxes_l} P_{\delta B}$$

and we define the objective function as the maximization of obj_{Step2}.

5.4. Implementation

Our Step 2 model only uses table constraints. These constraints are straightforward to implement with a CP library, and dedicated propagators have been designed for efficiently propagating them. Indeed, table constraints are a hot research topic and a key point for the success of CP (see, *e.g.*, [CY10, DHL+16, VLS18]). Hence, we have implemented our Step 2 model with Choco [PFL16], using the *domOverWDeg* variable ordering heuristic and the *lastConflict* search strategy (that are predefined in Choco).

It is possible to model Step 2 using other kinds of approaches, such as ILP or SAT; however, these approaches do not directly support table constraints and do not have dedicated algorithms for propagating them. Therefore, they are less straightforward to use. In the next two paragraphs, we describe some recent work that introduce ILP or SAT encodings for solving similar cryptanalysis problems.

Encoding table constraints with ILP. In [AST+17] Abdelkhalek *et al.* describe how to encode the S-box differential table which relates 8-bit input differences with 8-bit output differences. In a first step, they generate one inequality for each 16-bit tuple that does not belong to the table (in order to forbid this tuple). For the AES S-box, there are 33150 forbidden tuples. In a second step, they reduce the number of inequalities. As the initial set of inequalities corresponds to a product-of-sum representation of Boolean functions, the minimum set of inequalities corresponds to the set of prime implicants of the relation, and it can be computed by using the Quine-McCluskey algorithm [Qui55, MJ56]. However, as computing the minimum set is an $\mathcal{NP}$-hard problem, they use the heuristic algorithm called Espresso [BSVMH84] to compute an approximate solution. For the AES S-box, this algorithm reduces the number of inequalities from 33150 to 8302. Note that these inequalities must be added for each active S-box.

Encoding table constraints with SAT. In [Laf18], Lafitte describes how to encode a relation $R \subseteq \{0,1\}^n$ (such that R is composed of k words of $\{0,1\}^n$) as a SAT formula with $2^n - k$ clauses: Each of these clauses is a disjunction of n litterals which forbids one word of $\{0,1\}^n \setminus R$. This encoding may be used to model the S-box differential relation and, in this case, we have 33150 clauses for each byte that passes through an active S-box. In [SWW18], Sun *et al.* show how to reduce the number of clauses by using the same approach as in [AST+17].

In [Wal00], Walsh shows that enforcing arc consistency on a binary constraint filters more values than unit propagation on the kind of SAT encoding proposed in [Laf18, SWW18]. This explains why SAT solvers on this encoding are usually outperformed by CP solvers. However, in [Bac07], Bacchus introduces a SAT encoding of an arbitrary constraint (defined by a set of allowed tuples) which enforces GAC and is linear in the number of allowed tuples. Also, the set of allowed tuples may be represented by a binary decision diagram (BDD) [Bry86], and BDDs may be encoded into clauses which enforce GAC, as described by Eén and Sörensson in [ES06], for example. These encodings should improve performance of SAT solvers (and also ILP solvers as similar encodings could be used for ILP too) for computing optimal differential characteristics.

5.5. Experimental evaluation

In Table 2, we display the CPU time needed by Choco to search for optimal differential characteristics, given the set of truncated differential characteristics computed by *Step1-enum*. For all instances but three (AES-128-5, AES-192-9, and AES-192-10), the time needed to find the optimal solution given one truncated differential characteristic is smaller than 100 seconds, and the number of truncated differential characteristics is smaller than 20, so that the optimal solution for all truncated differential characteristics is found in less than 500 seconds. However, for AES-128-5 (resp. AES-192-9 and AES-192-10), the average solving time per truncated differential characteristic is 210 (resp. 147 and 92) seconds, and there are 1113 (resp. 240 and 602) truncated differential characteristics so that the total solving time for all truncated differential characteristics exceeds 65 hours (resp. 9 hours and 15 hours). Note that for these three instances most truncated differential characteristics are not byte consistent: Among the 1113 (resp. 240 and 602) characteristics enumerated by *Step1-enum*, only 97 (resp. 13 and 202) are byte-consistent.

	$\#T$	$\#B$	p	t_2	$\frac{t_2}{\#T}$
AES-128-3	4	2	2^{-31}	10	2.5
AES-128-4	8	8	2^{-75}	40	5
AES-128-5	1113	97	2^{-105}	235086	211.2
AES-192-3	15	15	2^{-6}	15	1
AES-192-4	4	4	2^{-24}	13	3.3
AES-192-5	2	2	2^{-30}	11	5.5
AES-192-6	6	6	2^{-60}	35	5.8
AES-192-7	4	4	2^{-78}	46	11.5
AES-192-8	8	8	2^{-108}	119	14.9
AES-192-9	240	80	2^{-146}	35254	146.9
AES-192-10	602	202	2^{-176}	55310	91.9
AES-256-3	33	33	2^{-6}	26	0.8
AES-256-4	14	14	2^{-18}	25	1.8
AES-256-5	4	4	2^{-18}	12	3
AES-256-6	3	3	2^{-30}	11	3.7
AES-256-7	1	1	2^{-30}	9	8.8
AES-256-8	3	1	2^{-60}	19	6.3
AES-256-9	16	16	2^{-92}	457	28.6
AES-256-10	4	4	2^{-98}	160	40
AES-256-11	4	4	2^{-122}	178	44.5
AES-256-12	4	4	2^{-122}	237	59.3
AES-256-13	4	4	2^{-146}	244	61
AES-256-14	4	4	2^{-146}	302	75.5

Table 2: Results of Choco for solving Step 2. For each instance, we display: the number $\#T$ of truncated differential characteristics, the number $\#B$ of byte-consistent truncated differential characteristics, the maximal probability $p = 2^{obj_{Step2}}$ among all byte-consistent truncated differential characteristics, the total time t_2 for solving Step 2 for all truncated differential characteristics, and the average time $\frac{t_2}{\#T}$ for solving Step 2 for one characteristic. Times are in seconds.

6. New two-step decomposition

Every Boolean solution computed during Step 1 corresponds to a truncated differential characteristic. Given one of these Boolean solutions, Step 2 is solved rather quickly, as shown in Table 2. However, in some cases, the number of Boolean solutions is quite large, and solving Step 2 for all Boolean

solutions becomes time-consuming.

To reduce the number of Boolean solutions, we propose to shift the frontier between Steps 1 and 2. More precisely, we modify the output of *Step1-enum*: Instead of enumerating all Boolean solutions, we focus on the variables associated with bytes that pass through S-boxes, *i.e.*, we enumerate all assignments of Boolean variables associated with differential bytes in $Sboxes_l$. For AES-128, this amounts to enumerating all assignments of $\Delta X_i[j][k]$ and $\Delta K_i[j][3]$ that belong to Boolean solutions (in other words, we do not enumerate the values of $\Delta K_i[j][k]$ with $k \in [0, 2]$, and we do not enumerate the values of $\Delta Y_i[j][k]$ and $\Delta Z_i[j][k]$). Hence, every Boolean assignment computed by the new Step 1 no longer corresponds to a truncated differential characteristic (as Boolean variables associated with bytes that do not pass through S-boxes are not assigned), but to a non empty set of truncated differential characteristics (all truncated differential characteristics that share the same values for Boolean variables associated with bytes that pass through S-boxes).

Step 2 is adapted to integrate the fact that the new Step 1 does not assign values to some variables: The domain of a variable δB associated with a Boolean variable ΔB which is not assigned in the new Step 1 is $D(\delta B) = [0, 255]$.

This simple reduction of the scope of the new Step 1 greatly reduces the number of different assignments (as many assignments only differ on values assigned to ΔY_i, ΔZ_i, or $\Delta K_i[j][k]$ with $k \in [0, 2]$), without increasing the size of the search space to explore in the new Step 2. Indeed, the values of δY_i, δZ_i, and $\delta K_i[j][k]$ with $k \in [0, 2]$ are deterministically inferred from the values of the variables associated with inputs and outputs of S-boxes (*i.e.*, δX_i, δSX_i, $\delta K_i[j][3]$, and $\delta SK_i[j][3]$ for AES-128).

In Table 3, we give the results obtained with this new decomposition. For the new Step 1, we compare CP_{EQ} and CP_{XOR} for solving *Step1-enum* with Picat-SAT. Again, CP_{XOR} is faster than CP_{EQ}, and it is able to solve all instances but two in less than 7 minutes. The two hardest instances are AES-128-5, which is solved in 24 minutes, and AES-192-10, which is solved in less than 4 hours. For this last instance, CP_{EQ} did not terminate after three days. We therefore stopped it, and report the number of Boolean assignments it found within these three days in Table 3.

Both CP_{EQ} and CP_{XOR} find the same number of Boolean assignments ($\#T$) for all instances but one. For AES-192-10, there are 7 Boolean assignments with CP_{XOR}, and at least 40 with CP_{EQ} (as CP_{EQ} has enumerated

	New Step 1				New Step 2			Total time	
	CP_{EQ}		CP_{XOR}						
	$\#T$	t_1^{enum}	$\#T$	t_1^{enum}	$\#B$	t_2	$\frac{t_2}{\#T}$	*seq*	*par*
AES-128-3	2	3	2	**2**	2	7	3.5	12	9
AES-128-4	1	16	1	**8**	1	13	12.6	35	35
AES-128-5	103	2651	103	**1409**	27	52313	507.9	53755	3388
AES-192-3	14	14	14	**8**	14	19	1.4	29	11
AES-192-4	2	7	2	**4**	2	7	3.5	16	13
AES-192-5	1	9	1	**4**	1	4	3.8	16	16
AES-192-6	2	27	2	**11**	2	14	7.0	43	36
AES-192-7	1	41	1	**17**	1	7	7.4	61	61
AES-192-8	1	221	1	**57**	1	8	8.2	138	138
AES-192-9	3	1720	3	**386**	3	109	36.3	1015	942
AES-192-10	$\geq$40	-	7	**13558**	7	281	40.1	17124	16892
AES-256-3	33	34	33	**23**	33	36	1.1	62	27
AES-256-4	10	25	10	**14**	10	24	2.4	45	23
AES-256-5	4	20	4	**10**	4	15	3.8	33	22
AES-256-6	3	27	3	**12**	3	16	5.3	45	34
AES-256-7	1	21	1	**8**	1	7	7.4	62	62
AES-256-8	2	55	2	**18**	2	14	7.0	81	74
AES-256-9	4	203	4	**63**	4	69	17.3	238	186
AES-256-10	1	99	1	**41**	1	45	45.3	198	198
AES-256-11	1	320	1	**77**	1	28	27.8	391	391
AES-256-12	1	258	1	**89**	1	35	35.2	264	264
AES-256-13	1	694	1	**140**	1	46	46.0	1008	1008
AES-256-14	1	1087	1	**97**	1	35	34.8	814	814

Table 3: Results with the new decomposition. For each instance, we display: the results for the new Step 1 with CP_{EQ} and CP_{XOR} ($\#T$ = number of Boolean assignments computed by *Step1-enum*; t_1^{enum} = time spent by Picat-SAT to solve *Step1-enum*), the results for the new Step 2 ($\#B$ = number of Byte-consistent Boolean assignments; t_2 = time spent by Choco for all Boolean assignments; t_{max} = worst time per Boolean assignment), and total time for solving the whole problem by Algorithm 1 with CP_{XOR} and the new decomposition (*seq* = time when using a single core; *par* = time when using $\#T$ cores in parallel for solving Step 2). Times are in seconds. We report '-' when time exceeds 3 days.

40 Boolean assignments within the CPU time limit of 3 days). As expected, there are less Boolean assignments with the new decomposition than with the original one for many instances. In some cases, the reduction is drastic: from 1113 (resp. 240 and 602) to 103 (resp. 3 and 7) for AES-128-5 (resp. AES-192-9 and AES-192-10). As a consequence, the time needed to enumerate all Boolean assignments is also smaller with the new decomposition. In some cases, the speed-up is important. For example, with CP_{XOR}, *Step1-enum* is solved in more than 6 (resp. 3, and 60) hours with the initial decomposition for AES-128-5 (resp. AES-192-9, and AES-192-10), whereas it is solved in less than 24 (7, and 226) minutes with the new decomposition.

For all instances but one (AES-128-5), every Boolean assignment computed by *Step1-enum* is byte consistent. However, for AES-128-5, only 27 Boolean assignments, among the 103 computed by *Step1-enum*, are byte consistent.

The time spent by Choco to find the optimal differential characteristic given one Boolean assignment ($\frac{t_2}{\#T}$) is rather comparable to the one displayed in Table 2, for the initial decomposition: it is larger for 9 instances, equal for 3 instances, and smaller for 10 instances. As there are less Boolean assignments with the new decomposition than with the initial one, the total time for Step 2 (t_2) is often smaller and, for the most challenging instances it is much smaller: it is larger than 65 hours (resp. 9 hours, and 15 hours) with the initial decomposition for AES-128-5 (resp. AES-192-9, and AES-192-10), whereas it is smaller than 15 hours (9 minutes, and 4 hours) with the new decomposition.

The last two columns of Table 3 give the time needed to solve the whole problem as described in Algorithm 1, *i.e.*, solve *Step1-opt* with CP_{XOR}, then solve *Step1-enum* with CP_{XOR} and the new Step 1, and finally solve the new Step 2 for each Boolean assignment computed by *Step1-enum* (for all instances, the loop lines 6 to 10 is executed only once as the exit condition is satisfied when $v = v^*$). The *seq* column gives the time when using a single core of a server with an Intel Xeon E5-2687W v4 CPU at 3.00GHz. The *par* column gives the time when Step 2 is solved in parallel for each Boolean assignment, using $\#T$ cores of the same server. This time has been estimated by adding the times of *Step1-opt* and *Step1-enum* with the maximal time needed to solve *Step2* for one Boolean assignment, over the $\#T$ Boolean assignments computed by *Step1-enum*.

The total *seq* time is smaller than one hour for all instances but two, and 11 instances are solved in less than one minute. The two hardest instances are

AES-128-5 (which is solved in less than 15 hours), and AES-192-10 (which is solved in less than 5 hours). When using $\#T$ cores to solve Step 2 in parallel for each Boolean assignment, the total solving time is reduced for all instances that have more than one Boolean assignment. In particular, the solving time is reduced to less than one hour for instance AES-128-5.

This is a clear improvement with respect to dedicated approaches as the approach of [FJP13] cannot be extended to keys of size $l > 128$ bits, due to its memory complexity, and the approach of [BN10] needs several weeks to solve Step 1 for AES-192.

As already pointed out in [GMS16], for AES-128-4, we have found a byte-consistent solution with $obj_{Step1} = 12$ and a probability equal to 2^{-79}. This solution is better than the solution claimed to be optimal in [BN10] and [FJP13]: In these papers, authors address the same problem as us and they say that the best byte-consistent solution has $obj_{Step1} = 13$, and a probability equal to 2^{-81}.

Finally, we have found better solutions for AES-256. In particular, we have computed the actual optimal differential characteristic for AES-256-14, and its probability is 2^{-146}, instead of 2^{-154} for the one given in [BKN09].

7. Conclusion

We introduced new CP models for finding optimal differential characteristics for AES: We introduced a new decomposition of the solving process in two steps, that allows us to reduce the number of Boolean assignments, and a new CP model for solving the first step, that exploits new XOR constraints inferred from the key schedule to reduce the number of byte-inconsistent Boolean assignments. These new CP models have allowed us to solve all instances in a few hours.

For AES-128, we have found a better differential characteristic for $r = 4$ rounds, with a greater probability, but it cannot be used to improve attacks as the best attacks in the related-key and chosen-key models exploit the optimal 5 round related-key differential characteristic.

For AES-192 and AES-256, the optimal differential characteristics computed with our new models allowed us to mount new related-key attacks, related-key distinguishers and q-multicollisions. In particular, we improved the related-key distinguisher and the basic related-key differential attack on the full AES-256 by a factor 2^6 and the q-multicollisions by a factor 2 (see [GLMS18] for more details).

These cryptanalytic problems open new and exciting challenges for the CP community. In particular, these problems are not easy to model. More precisely, naive CP models (such as the one described in Section 3) are easy to design but they may not scale well. The addition of new XOR equations and the introduction of *diff* variables that model inequality relations at the byte level drastically improve the solving process, but these constraints are not straightforward to find and implement. Hence, a challenge is to define new CP frameworks, dedicated to this kind of cryptanalytic problems, in order to ease the development of efficient CP models for these problems.

Among the most challenging cryptanalytic problems that we want to look at, we may cite the ones studied in [CHP+18] that directly implement particular attacks from related key differential characteristics and the recent results obtained using ILP solvers for studying the so-called division properties in an other symmetric key encryption primitive, the stream ciphers [TIHM17].

Finally, developing efficient methods to evaluate the security of block cipher is a first step towards the computer aided design of such primitives. This is a challenging and promising direction of research that will help us to have security by design.

Acknowledgements. This research was conducted with the support of the FEDER program of 2014-2020, the region council of Auvergne-Rhône-Alpes, the GDR-IA, and the ANR (DeCrypt ANR-18-CE39-0007). We thank the reviewers for their comments that helped us improving the paper, Jérémie Detrey for implementing the C code that checks the completeness of the sets $xorEq_l$, Charles Prud'homme and Jean-Guillaume Fages for their technical support on the use of Choco, and Neng-Fa Zhou for his technical support on the use of Picat.

[AST+17] Ahmed Abdelkhalek, Yu Sasaki, Yosuke Todo, Mohamed Tolba, and Amr M. Youssef. MILP modeling for (large) s-boxes to optimize probability of differential characteristics. *IACR Trans. Symmetric Cryptol.*, 2017(4):99–129, 2017.

[Bac07] Fahiem Bacchus. Gac via unit propagation. In *Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science, pages 133–147. Springer, 2007.

[Bie14] Armin Biere. Yet another local search solver and lingeling and friends entering the sat competition 2014. pages 39–40, 01 2014.

[Bih93] Eli Biham. New types of cryptoanalytic attacks using related keys (extended abstract). In *Advances in Cryptology - EUROCRYPT '93*, volume 765 of *Lecture Notes in Computer Science*, pages 398–409. Springer, 1993.

[BJK+16] Christof Beierle, Jérémy Jean, Stefan Kölbl, Gregor Leander, Amir Moradi, Thomas Peyrin, Yu Sasaki, Pascal Sasdrich, and Siang Meng Sim. The SKINNY family of block ciphers and its low-latency variant MANTIS. In *Advances in Cryptology - CRYPTO 2016 Part II*, volume 9815 of *LNCS*, pages 123–153. Springer, 2016.

[BKN09] Alex Biryukov, Dmitry Khovratovich, and Ivica Nikolic. Distinguisher and related-key attack on the full AES-256. In *Advances in Cryptology - CRYPTO 2009*, volume 5677 of *LNCS*, pages 231–249. Springer, 2009.

[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 *Advances in Cryptology - EUROCRYPT 2010*, volume 6110 of *LNCS*, pages 322–344. Springer, 2010.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[BS91] Eli Biham and Adi Shamir. Differential cryptoanalysis of feal and n-hash. In *Advances in Cryptology - EUROCRYPT '91*, volume 547 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 1991.

[BSVMH84] Robert King Brayton, Alberto L. Sangiovanni-Vincentelli, Curtis T. McMullen, and Gary D. Hachtel. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.

[CHP+18] Carlos Cid, Tao Huang, Thomas Peyrin, Yu Sasaki, and Ling Song. Boomerang connectivity table: A new cryptanalysis tool. In *Advances in Cryptology - EUROCRYPT 2018*, volume 10821

of *Lecture Notes in Computer Science*, pages 683–714. Springer, 2018.

[CS14] Geoffrey Chu and Peter J. Stuckey. Chuffed solver description, 2014. Available at `http://www.minizinc.org/challenge2014/description\_chuffed.txt`.

[CY10] Kenil C. K. Cheng and Roland H. C. Yap. An mdd-based generalized arc consistency algorithm for positive and negative table constraints and some global constraints. *Constraints*, 15(2):265–304, 2010.

[DHL+16] Jordan Demeulenaere, Renaud Hartert, Christophe Lecoutre, Guillaume Perez, Laurent Perron, Jean-Charles Régin, and Pierre Schaus. Compact-table: Efficiently filtering table constraints with reversible sparse bit-sets. In Michel Rueher, editor, *Principles and Practice of Constraint Programming - 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings*, volume 9892 of *Lecture Notes in Computer Science*, pages 207–223. Springer, 2016.

[DR13] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES-the advanced encryption standard*. Springer Science & Business Media, 2013.

[ES06] Niklas Eén and Niklas Sörensson. Translating pseudo-boolean constraints into SAT. *JSAT*, 2(1-4):1–26, 2006.

[FIP01] FIPS 197. Advanced Encryption Standard. Federal Information Processing Standards Publication 197, 2001. U.S. Department of Commerce/N.I.S.T.

[FJP13] Pierre-Alain Fouque, Jérémy Jean, and Thomas Peyrin. Structural evaluation of AES and chosen-key distinguisher of 9-round AES-128. In *Advances in Cryptology - CRYPTO 2013 - Part I*, volume 8042 of *LNCS*, pages 183–203. Springer, 2013.

[Gec06] Gecode Team. Gecode: Generic constraint development environment, 2006. Available from `http://www.gecode.org`.

[GL16] David Gérault and Pascal Lafourcade. Related-key cryptanalysis of midori. In *Progress in Cryptology - INDOCRYPT 2016*, volume 10095 of *LNCS*, pages 287–304, 2016.

[GLMS17] David Gérault, Pascal Lafourcade, Marine Minier, and Christine Solnon. Revisiting AES related-key differential attacks with constraint programming. Cryptology ePrint Archive, Report 2017/139, Extended version of [GLMS18], 2017. `https://eprint.iacr.org/2017/139`.

[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.

[GMS16] David Gérault, Marine Minier, and Christine Solnon. Constraint programming models for chosen key differential cryptanalysis. In *Principles and Practice of Constraint Programming - CP 2016*, volume 9892 of *LNCS*, pages 584–601. Springer, 2016.

[KLT15] Stefan Kölbl, Gregor Leander, and Tyge Tiessen. Observations on the SIMON block cipher family. In *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part I*, volume 9215 of *Lecture Notes in Computer Science*, pages 161–185. Springer, 2015.

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

[Laf18] Frédéric Lafitte. Cryptosat: a tool for sat-based cryptanalysis. *IET Information Security*, 12(6):463–474, 2018.

[LCM+17] Fanghui Liu, Waldemar Cruz, Chujiao Ma, Greg Johnson, and Laurent Michel. A tolerant algebraic side-channel attack on AES using CP. In *Principles and Practice of Constraint Programming - 23rd International Conference, CP 2017, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, volume 10416 of *Lecture Notes in Computer Science*, pages 189–205. Springer, 2017.

[MJ56] E. J. McCluskey Jr. Minimization of boolean functions*. *Bell System Technical Journal*, 35(6):1417–1444, 1956.

[MP13] Nicky Mouha and Bart Preneel. A proof that the ARX cipher salsa20 is secure against differential cryptanalysis. *IACR Cryptology ePrint Archive*, 2013:328, 2013.

[MS77] Florence Jessie MacWilliams and Neil James Alexander Sloane. *The theory of error-correcting codes*, volume 16. Elsevier, 1977.

[MSR14] Marine Minier, Christine Solnon, and Julia Reboul. Solving a Symmetric Key Cryptographic Problem with Constraint Programming. In *13th International Workshop on Constraint Modelling and Reformulation (ModRef), in conjunction with CP'14*, pages 1–13, 2014.

[MWGP12] Nicky Mouha, Qingju Wang, Dawu Gu, and Bart Preneel. Differential and linear cryptanalysis using mixed-integer linear programming. In Chuan-Kun Wu, Moti Yung, and Dongdai Lin, editors, *Information Security and Cryptology*, pages 57–76. Springer, 2012.

[NSB+07] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard CP modelling language. In *Principles and Practice of Constraint Programming - CP 2007*, volume 4741 of *LNCS*, pages 529–543. Springer, 2007.

[Opt18] Gurobi Optimization. Gurobi optimizer reference manual, 2018.

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

[Qui55] W. V. Quine. A way to simplify truth functions. *The American Mathematical Monthly*, 62(9):627–631, 1955.

[RBW06] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., New York, NY, USA, 2006.

[RSM+11] Venkatesh Ramamoorthy, Marius-Calin Silaghi, Toshihiro Matsui, Katsutoshi Hirayama, and Makoto Yokoo. The design of cryptographic s-boxes using csps. In *Principles and Practice of Constraint Programming - CP 2011 - 17th International Conference, CP 2011*, volume 6876 of *Lecture Notes in Computer Science*, pages 54–68. Springer, 2011.

[SGL+17] Siwei Sun, David Gérault, Pascal Lafourcade, Qianqian Yang, Yosuke Todo, Kexin Qiao, and Lei Hu. Analysis of aes, skinny, and others with constraint programming. In *24th International Conference on Fast Software Encryption*, 2017.

[SHW+14] Siwei Sun, Lei Hu, Peng Wang, Kexin Qiao, Xiaoshuang Ma, and Ling Song. Automatic security evaluation and (related-key) differential characteristic search: Application to simon, present, lblock, DES(L) and other bit-oriented block ciphers. In *Advances in Cryptology - ASIACRYPT 2014 Part I*, volume 8873 of *LNCS*, pages 158–178. Springer, 2014.

[Sin06] R. Singleton. Maximum distance -nary codes. *IEEE Trans. Inf. Theor.*, 10(2):116–118, September 2006.

[SNC09] Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT solvers to cryptographic problems. In *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009*, volume 5584 of *Lecture Notes in Computer Science*, pages 244–257. Springer, 2009.

[ST17] Yu Sasaki and Yosuke Todo. New impossible differential search tool from design and cryptanalysis aspects - revealing structural properties of several ciphers. In *Advances in Cryptology - EUROCRYPT 2017*, volume 10212 of *Lecture Notes in Computer Science*, pages 185–215, 2017.

[SWW17] Ling Sun, Wei Wang, and Meiqin Wang. Automatic search of bit-based division property for ARX ciphers and word-based division property. In *Advances in Cryptology - ASIACRYPT 2017*, pages 128–157, 2017.

[SWW18] Ling Sun, Wei Wang, and Meiqin Wang. More accurate differential properties of led64 and midori64. *IACR Transactions on Symmetric Cryptology*, 2018(3):93–123, 2018.

[TIHM17] Yosuke Todo, Takanori Isobe, Yonglin Hao, and Willi Meier. Cube attacks on non-blackbox polynomials based on division property. In *Advances in Cryptology - CRYPTO 2017*, volume 10403 of *Lecture Notes in Computer Science*, pages 250–279. Springer, 2017.

[VLS18] Hélène Verhaeghe, Christophe Lecoutre, and Pierre Schaus. Compact-mdd: Efficiently filtering (s)mdd constraints with reversible sparse bit-sets. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018*, pages 1383–1389, 2018.

[Wal00] Toby Walsh. SAT v CSP. In *Principles and Practice of Constraint Programming*, volume 1894 of *Lecture Notes in Computer Science*, pages 441–456. Springer, 2000.

[ZK17] Neng-Fa Zhou and Håkan Kjellerstrand. Optimizing SAT encodings for arithmetic constraints. In *Principles and Practice of Constraint Programming - 23rd International Conference, CP 2017*, volume 10416 of *Lecture Notes in Computer Science*, pages 671–686. Springer, 2017.

[ZKF15] Neng-Fa Zhou, Hakan Kjellerstrand, and Jonathan Fruhman. *Constraint Solving and Planning with Picat*. Springer, 2015.

APPENDIX

Extension to AES-192 and AES-256

Description of KeySchedule *when* $l \in \{192, 256\}$

AES-192

For AES-192, the initial key K has 6 columns. The first four columns of K are used to initialize the four columns of K_0, *i.e.*, for each row $j \in [0,3]$ and each column $k \in [0,3]$, $K_0[j][k] = K[j][k]$. The last two columns of K are used to initialize the first two columns of K_1, *i.e.*, for each row $j \in [0,3]$ and each column $k \in [0,1]$, $K_1[j][k] = K[j][k+4]$.

The last two columns of K_1, and the four columns of all following subkeys are defined as follows.

Columns 1 and 3 are always obtained by performing a XOR between bytes of the previous and the 6^{th} previous column:

$$\begin{aligned} \forall i \in [2,r], \forall j \in [0,3], K_i[j][1] &= K_{i-2}[j][3] \oplus K_i[j][0] \\ \forall i \in [1,r], \forall j \in [0,3], K_i[j][3] &= K_{i-1}[j][1] \oplus K_i[j][2] \end{aligned}$$

Columns 0 and 2 are also obtained by performing a XOR between bytes of the previous and the 6^{th} previous column, but some of these XORs are combined with SubBytes operations, word rotations, and constant additions, depending on the value of $i\%3$. For column 0, if the round number $i \in [1,r]$ is such that $i\%3 \neq 0$, then

$$\forall j \in [0,3], K_i[j][0] = K_{i-2}[j][2] \oplus K_{i-1}[j][3]$$

otherwise

$$\begin{aligned} K_i[0][0] &= K_{i-2}[0][2] \oplus SK_{i-1}[1][3] \oplus c_i \\ \forall j \in [1,3], K_i[j][0] &= K_{i-2}[j][2] \oplus SK_{i-1}[(j+1)\%4][3] \end{aligned}$$

where c_i is a constant, and $SK_{i-1}[j][3]$ is the result of applying the SubBytes operation on $K_{i-1}[j][3]$, *i.e.*

$$\forall j \in [0,3], SK_{i-1}[j][3] = S(K_{i-1}[j][3])$$

For column 2, if the round number $i \in [2,r]$ is such that $i\%3 \neq 1$, then

$$\forall j \in [0,3], K_i[j][2] = K_{i-1}[j][0] \oplus K_i[j][1]$$

otherwise

$$\forall j \in [0,3], K_i[j][2] = K_{i-1}[j][0] \oplus SK_i[(j+1)\%4][1]$$

where $SK_i[j][1]$ is the result of applying the SubBytes operation on $K_i[j][1]$, *i.e.*

$$\forall j \in [0,3], SK_i[j][1] = S(K_i[j][1])$$

AES-256

For AES-256, the initial key K has 8 columns, and these columns are used to initialize K_0 and K_1, *i.e.*, for each row $j \in [0,3]$ and each column $k \in [0,3]$, $K_0[j][k] = K[j][k]$ and $K_1[j][k] = K[j][k+4]$.

Then, for each round $i \in [2, r-1]$, the subkey K_{i+1} is generated from the previous subkeys K_i and K_{i-1} as follows:

- The first column of K_{i+1} is obtained from the first and last columns of K_i in two steps. First, we apply the SubBytes operation on all bytes of the last column of K_i. We note $SK_i[j][3]$ the resulting byte for row j, *i.e.*,

  $$\forall i \in [1, r-1], \forall j \in [0,3], SK_i[j][3] = S(K_i[j][3]).$$

  Then, depending on whether i is odd or even, we either simply perform a XOR, or combine this XOR with a rotation and a constant addition: for each $i \in [1, r-1]$, if $i\%2 = 0$ then

  $$\begin{aligned} K_{i+1}[0][0] &= SK_i[1][3] \oplus K_i[0][0] \oplus c_i \\ \forall j \in [1,3], K_{i+1}[j][0] &= SK_i[(j+1)\%4][3] \oplus K_i[j][0] \end{aligned}$$

  else

  $$\forall j \in [1,3], K_{i+1}[j][0] = SK_i[j][3] \oplus K_i[j][0].$$

- For the last three columns $k \in [1,3]$, we simply perform XORs:

  $$\forall i \in [1, r-1], \forall j \in [0,3], \forall k \in [1,3], K_{i+1}[j][k] = K_{i-1}[j][k-1] \oplus K_i[j][k].$$

Definition of diffBytes_l *and* $Sboxes_l$ *when* $l \in \{192, 256\}$
AES-192

$$
\begin{aligned}
diffBytes_{192} = \ & \{ \quad \delta X[j][k], \delta K_i[j][k], \delta X_i[j][k], \delta SX_i[j][k], \\
& \quad \delta Y_i[j][k], \delta Z_i[j][k] : i \in [0, r], j \in [0, 3], k \in [0, 3]\} \\
& \cup \\
& \{ \quad \delta SK_i[j][1] : i \in [1, r], j \in [0, 3], i\%3 = 1\} \\
& \cup \\
& \{ \quad \delta SK_i[j][3] : i \in [1, r], j \in [0, 3], i\%3 = 2\} \\
Sboxes_{192} = \ & \{ \quad \delta K_i[j][1] : i \in [1, r-1], j \in [0, 3], i\%3 = 1\}, \\
& \cup \\
& \{ \quad \delta K_i[j][3] : i \in [1, r-1], j \in [0, 3], i\%3 = 2\} \\
& \cup \\
& \{ \quad \delta X_i[j][k] : i \in [0, r-1], j \in [0, 3], k \in [0, 3]\}
\end{aligned}
$$

AES-256

$$
\begin{aligned}
diffBytes_{256} = \ & \{ \quad \delta X[j][k], \delta K_i[j][k], \delta X_i[j][k], \delta SX_i[j][k], \\
& \quad \delta Y_i[j][k], \delta Z_i[j][k] : i \in [0, r], j \in [0, 3], k \in [0, 3]\} \\
& \cup \\
& \{ \quad \delta SK_i[j][3] : i \in [1, r], j \in [0, 3], k \in [0, 3]\} \\
Sboxes_{256} = \ & \{ \quad \delta K_i[j][3] : i \in [1, r-1], j \in [0, 3]\}, \\
& \cup \\
& \{ \quad \delta X_i[j][k] : i \in [0, r-1], j \in [0, 3], k \in [0, 3]\}
\end{aligned}
$$

Definition of constraints related to KS for Step 1 when $l \in \{192, 256\}$

Constraints (C_7) and (C_8) of Fig. 2 correspond to the key schedule for AES-128. For AES-192 and AES-256, these two constraints must be replaced by the constraints listed below.

AES-192

$$
\begin{array}{lcl}
\forall i \in [0, r-1], \forall j \in [0,3]: & \text{if} & (i-1)\%3 = 2 \\
& \text{then} & XOR(\Delta K_i[j][0], \Delta SK_{i-1}[(j+1)\%4][3], \Delta K_{i-2}[j][2]) \\
& \text{else} & XOR(\Delta K_i[j][0], \Delta K_{i-1}[j][3], \Delta K_{i-2}[j][2]) \\
\forall i \in [2, r-1], \forall j \in [0,3]: & & XOR(\Delta K_i[j][1], \Delta K_{i-2}[j][3], \Delta K_i[j][0]) \\
\forall i \in [1, r-1], \forall j \in [0,3]: & \text{if} & i\%3 = 1 \\
& \text{then} & XOR(\Delta K_i[j][2], \Delta SK_i[(j+1)\%4][1], \Delta K_{i-1}[j][0]) \\
& \text{else} & XOR(\Delta K_i[j][2], \Delta K_i[j][1], K_{i-1}[j][0]) \\
\forall i \in [1, r-1], \forall j \in [0,3]: & & XOR(\Delta K_i[j][3], \Delta K_{i-1}[j][1], \Delta K_i[j][2])
\end{array}
$$

AES-256

$$
\begin{array}{rcl}
\forall i \in [1, r-1]: & & XOR(\Delta K_{i+1}[0][0], \Delta SK_i[1][3], \Delta K_i[0][0]) \\
\forall i \in [1, r-1], \forall j \in [1,3]: & \text{if} & i\%2 = 0 \\
& \text{then} & XOR(\Delta K_{i+1}[j][0], \Delta SK_i[(j+1)\%4][3], \Delta K_i[j][0]) \\
& \text{else} & XOR(\Delta K_{i+1}[j][0], \Delta SK_i[j][3], \Delta K_i[j][0]) \\
\forall i \in [1, r-1], \forall j \in [0,3], \forall k \in [1,3]: & & XOR(\Delta K_{i+1}[j][k], \Delta K_{i-1}[j][k-1], \Delta K_i[j][k]).
\end{array}
$$