



HAL
open science

Automatic Generation of Declarative Models for Differential Cryptanalysis

Luc Libralesso, François Delobel, Pascal Lafourcade, Christine Solnon

► **To cite this version:**

Luc Libralesso, François Delobel, Pascal Lafourcade, Christine Solnon. Automatic Generation of Declarative Models for Differential Cryptanalysis. CP 2021 - 27th International Conference on Principles and Practice of Constraint Programming, Oct 2021, Montpellier, France. 10.4230/LIPIcs.CP.2021.19 . hal-03320980

HAL Id: hal-03320980

<https://hal.science/hal-03320980>

Submitted on 16 Aug 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

1 Automatic Generation of Declarative Models 2 for Differential Cryptanalysis

3 **Luc Libralesso**

4 LIMOS, CNRS UMR 6158, University Clermont Auvergne, France

5 **François Delobel**

6 LIMOS, CNRS UMR 6158, University Clermont Auvergne, France

7 **Pascal Lafourcade**

8 LIMOS, CNRS UMR 6158, University Clermont Auvergne, France

9 **Christine Solnon**

10 INSA Lyon, CITI, INRIA CHROMA, F-69621 Villeurbanne, France

11 — Abstract —

12 When designing a new symmetric block cipher, it is necessary to evaluate its robustness against
13 differential attacks. This is done by computing Truncated Differential Characteristics (TDCs) that
14 provide bounds on the complexity of these attacks. TDCs are often computed by using declarative
15 approaches such as CP (Constraint Programming), SAT, or ILP (Integer Linear Programming).
16 However, designing accurate and efficient models for these solvers is a difficult, error-prone and
17 time-consuming task, and it requires advanced skills on both symmetric cryptography and solvers.

18 In this paper, we describe a tool for automatically generating these models, called TAGADA (Tool
19 for Automatic Generation of Abstraction-based Differential Attacks). The input of TAGADA is an
20 operational description of the cipher by means of black-box operators and bipartite Directed Acyclic
21 Graphs (DAGs). Given this description, we show how to automatically generate constraints that
22 model operator semantics, and how to generate MiniZinc models. We experimentally evaluate our
23 approach on two different kinds of differential attacks (*e.g.*, single-key and related-key) and four
24 different symmetric block ciphers (*e.g.*, the AES (Advanced Encryption Standard), Craft, Midori,
25 and Skinny). We show that our automatically generated models are competitive with state-of-the-art
26 approaches. These automatically generated models constitute a new benchmark composed of eight
27 optimization problems and eight enumeration problems, with instances of increasing size in each
28 problem. We experimentally compare CP, SAT, and ILP solvers on this new benchmark.

29 **2012 ACM Subject Classification** Security and privacy → Cryptanalysis and other attacks

30 **Keywords and phrases** Constraint Programming, SAT, ILP, Differential Cryptanalysis

31 **Digital Object Identifier** 10.4230/LIPIcs.CP.2021.19

32 **Funding** This work has been partially supported by the French government research program
33 “Investissements d’Avenir” through the IDEX-ISITE initiative 16-IDEX-0001 (CAP 20-25) and the
34 IMobS3 Laboratory of Excellence (ANR-10-LABX-16-01) and the French ANR PRC grant MobiS5
35 (ANR-18-CE39-0019), DECRYPT (ANR-18-CE39-0007), SEVERITAS (ANR-20-CE39-0005).

1 Introduction

Symmetric cryptography provides algorithms for ciphering a text given a secret key. Differential cryptanalysis is a well-known attack technique that aims at checking if the key can be guessed by introducing differences and studying their propagation during the ciphering process [6]. To evaluate the robustness of a new ciphering algorithm towards differential attacks, we compute *Truncated Differential Characteristics (TDCs)* as initially proposed by Knudsen in [20], where sequences of bits are abstracted by Boolean values in order to locate differences (without computing their exact values). We first solve an optimization problem (called Step1-opt) that aims at finding a TDC that has a minimal number of differences that pass through non-linear operators. This provides bounds on the complexity of differential attacks, and in some cases these bounds are large enough to ensure security. When bounds are not large enough, we have to solve an enumeration problem (called Step1-enum) that aims at finding all TDCs that have a given number of differences that pass through non-linear operators. Finally, for each enumerated TDC, we have to compute a *Maximum Differential Characteristic (MDC)*, *i.e.*, find difference values that have the largest probability given their positions identified in the TDC. MDCs are then used to design attacks. Computing an MDC given a TDC is a problem that is efficiently tackled by CP solvers (thanks to table constraints) [16]. Step1-opt and Step1-enum are much more challenging problems. They may be solved by using declarative approaches such as CP (Constraint Programming), SAT, or ILP (Integer Linear Programming) [11]. However, designing accurate and efficient models for these solvers is a difficult, error-prone and time-consuming task, and it requires advanced skills in both symmetric cryptography and combinatorial optimization.

Contributions and Overview of the Paper

In this paper, we describe a tool (called TAGADA) that automatically generates MiniZinc models for solving Step1-opt and Step1-enum problems given a cipher description. In Section 2, we introduce a unifying framework for describing symmetric block ciphers by means of elementary operators and bipartite Directed Acyclic Graphs (DAGs) that specify how these operators are combined. In Section 3, we formally define Step1-opt and Step1-enum problems, and we describe existing approaches for solving these problems.

In Section 4, we describe the input format of TAGADA which is based on the framework introduced in Section 2. Operator semantics are specified by functions which may be black boxes extracted from an existing implementation of the cipher. The DAG is specified in a JSON file. As the creation of this file may be tedious, TAGADA includes a set of functions for easing its generation. TAGADA also includes a function for automatically transforming the input description into an operational cipher. Hence, the correctness of the description is tested by comparing the outputs of the automatically generated cipher with the outputs of the original implementation of the cipher.

In Section 5, we describe how TAGADA automatically generates MiniZinc [21] models for computing TDCs. One key point is to define constraints associated with operators. In existing models, these constraints have been crafted by researchers, and some of these constraints require to have advanced knowledge on both symmetric cryptography and mathematical modelling. We show how to automatically generate these constraints from the functions that describe operator semantics. We also automatically improve models by both enriching and shaving the DAG.

In Section 6, we experimentally evaluate these models for two kinds of differential attacks, *i.e.*, single-key and related-key, and four ciphering algorithms, *i.e.*, the AES, Craft, Midori

82 and Skinny. We report results obtained with ILP, SAT and CP solvers. We also compare the
 83 automatically generated models with state-of-the-art hand-crafted models, and we show that
 84 TAGADA models are competitive with them.

85 Notations

86 We denote $[n, m]$ the set of all integer values ranging from n to m . Sequences of bits are
 87 denoted by x, y, z, \dots (possibly sub-scripted). The length of a sequence x is denoted $\#x$.
 88 The bitwise XOR operator is denoted \oplus . Tuples are denoted t (possibly sub-scripted), and
 89 the arity of a tuple t is denoted $\#t$. We denote $[0, 1]^{k \times p}$ the set of all possible tuples of k -bit
 90 sequences of arity p . Given two tuples of bit sequences $t = (y_1, \dots, y_n)$ and $t' = (y'_1, \dots, y'_n)$,
 91 we denote $t \oplus t'$ the tuple corresponding to $(y_1 \oplus y'_1, \dots, y_n \oplus y'_n)$.

92 2 Unifying Description of Symmetric Block Ciphers

93 The best-known symmetric block cipher is the AES (Advanced Encryption Standard),
 94 which is the standard for block ciphers since 2001 [12]. There exist many other symmetric
 95 block ciphers, that have been designed for previous competitions or the ongoing lightweight
 96 cryptography standardization competition organized by the NIST (*National Institute of*
 97 *Standards and Technology*). Some ciphers are designed for devices with limited computational
 98 resources, for example: Craft [5], Deoxys [19], Gift [2], Midori [1], Present [8], Skinny [4],
 99 Simon and Speck [3].

100 As our goal is to design a generic tool that automatically generates a model for computing
 101 TDCs from the description of a cipher, we describe these ciphers in a unified way, by means of
 102 DAGs. This unifying description is our first step towards automatic differential cryptanalysis.

103 2.1 Ciphering Operators

104 The encryption of a plaintext is achieved by applying elementary ciphering operators. Each
 105 operator o has a tuple of input parameters denoted $t_{in}(o)$ and a tuple of output parameters
 106 denoted $t_{out}(o)$ such that each parameter is a bit sequence, *i.e.*, $t_{in}(o) = (x_1, \dots, x_{\#t_{in}(o)})$
 107 and $t_{out}(o) = (y_1, \dots, y_{\#t_{out}(o)}) = o(x_1, \dots, x_{\#t_{in}(o)})$. Without loss of generality, we assume
 108 that all bit sequences have the same length k (if this is not the case, we may split sequences
 109 so that they all have the same length). Typically, $k = 8$ (resp. $k = 4$) and k -bit sequences
 110 correspond to bytes (resp. nibbles).

111 ► **Example 1.** The AES uses four elementary operators that operate on bytes (*i.e.*, $k = 8$):

- 112 ■ XOR, such that $t_{in}(xor) = (x_1, x_2)$, $t_{out}(xor) = (y_1)$, and $y_1 = x_1 \oplus x_2$;
- 113 ■ ShiftRows, denoted SR_s with $s \in [0, 3]$, such that $t_{in}(SR_s) = (x_1, x_2, x_3, x_4)$, $t_{out}(SR_s) =$
 114 (y_1, y_2, y_3, y_4) , and $\forall i \in [1, 4], y_i = x_{1+(i+s)\%4}$ where $\%$ is the modulo operation (in other
 115 words, SR_s simply shifts the positions of the four input bytes);
- 116 ■ MixColumns, denoted MC , such that $t_{in}(MC) = (x_1, x_2, x_3, x_4)$, $t_{out}(MC) = (y_1, y_2, y_3,$
 117 $y_4)$, and $\forall i \in [1, 4], y_i = (M_{i,1} \otimes x_1) \oplus (M_{i,2} \otimes x_2) \oplus (M_{i,3} \otimes x_3) \oplus (M_{i,4} \otimes x_4)$ where $M_{i,j}$
 118 are constant coefficients, and \otimes is a finite field multiplication;
- 119 ■ SubBytes, denoted S , such that $t_{in}(S) = (x_1)$, $t_{out}(S) = (y_1)$, and y_1 is obtained from x_1
 120 by using a substitution that is represented by a look-up table, called S-Box.

121 More generally, there are two main categories of operators that ensure two main concepts
 122 identified by Shannon in [24]: Non-linear operators that ensure confusion, and linear operators
 123 that ensure diffusion. Non-linear operators are either S-Boxes (like the AES SubBytes) or

124 non-linear arithmetic operations (like in ARX¹ structures). The most common linear
 125 operations used in symmetric ciphers are: multiplication by a MDS (Maximum Distance
 126 Separable) matrix (like the AES MixColumns), bit permutations, XOR and rotation (like
 127 the AES ShiftRows). Every linear operator o satisfies the following property: $\forall t, t' \in$
 128 $[0, 1]^{k \cdot \#t_{in}(o)}, o(t) \oplus o(t') = o(t \oplus t')$.

129 2.2 Description of a Cipher with a DAG

130 Given a plaintext and a key, a cipher returns a ciphertext. The plaintext and the key are
 131 bit-sequences, and we assume that they have been split into k -bit sequences. The ciphertext
 132 is computed by applying operators, and this process may be described by a DAG that
 133 contains two different kinds of vertices denoted P and O , respectively: each vertex in P
 134 corresponds to a parameter and is a k -bit sequence, whereas each vertex in O corresponds to
 135 an operator. Arcs connect operators to their input and output parameters: the predecessors
 136 (resp. successors) of an operator o are denoted $pred(o)$ (resp. $succ(o)$) and they correspond
 137 to input (resp. output) parameters. As parameters are ordered, $pred(o)$ and $succ(o)$ are
 138 tuples (instead of sets) and the order is represented by arc labels: an incoming arc (x, o)
 139 (resp. outgoing arc (o, x)) is labelled with $i \in [1, \#t_{in}(o)]$ (resp. $i \in [1, \#t_{out}(o)]$), meaning
 140 that x is the i^{th} input (resp. output) parameter in $pred(o)$ (resp. $succ(o)$).

141 Some input parameters have no predecessor in the DAG. These input parameters either
 142 correspond to k -bit sequences that are resulting from the plaintext or the key, or to constant
 143 values. The set of input parameters that are constant values is denoted C .

144 Most ciphers are iterative processes composed of r rounds. This round decomposition
 145 does not appear in the DAG as it is not necessary for automatically generating models.

146 ► **Example 2.** We display in Fig. 1 the DAG that describes the first AES round.

147 3 Optimization and Enumeration of TDCs

148 We first define MDCs in Section 3.1; then we define TDCs in Section 3.2; and finally, we
 149 define the two problems addressed in this paper, Step1-opt and Step1-enum, in Section 3.3.

150 3.1 Maximum Differential Characteristics

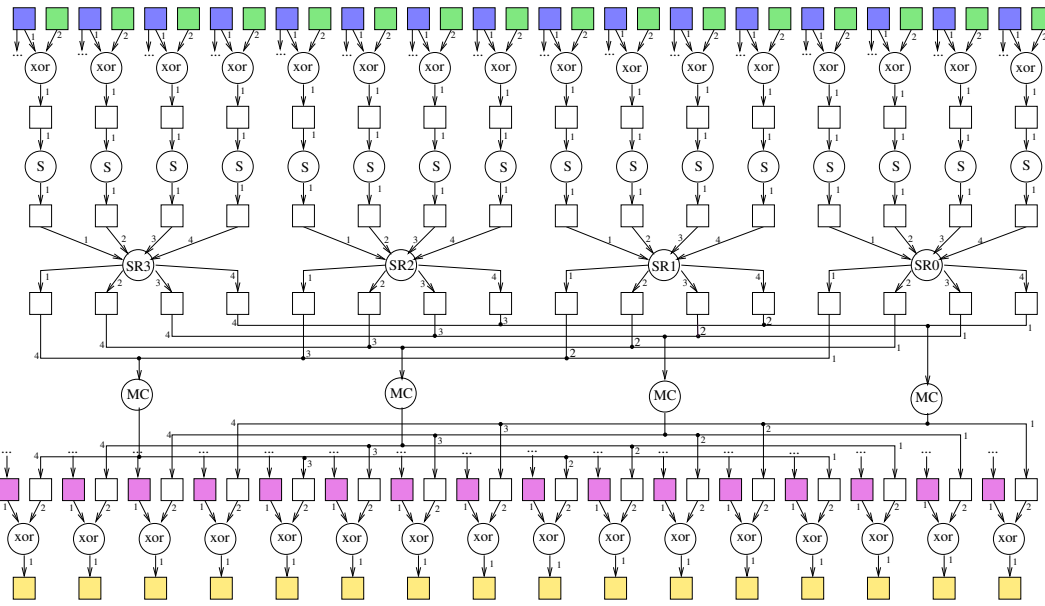
151 To design differential attacks, we study the propagation of differences during the ciphering
 152 process. To introduce differences in a k -bit sequence x , we XOR it with another k -bit sequence
 153 x' , and we denote δx the resulting differential sequence, *i.e.*, $\delta x = x \oplus x'$. When $\delta x = 0$,
 154 there is no difference (*i.e.*, $x = x'$) whereas when $\delta x \neq 0$ there are differences (*i.e.*, $x \neq x'$).
 155 Similarly, we denote δt the differential tuple obtained by XORing the elements of the two
 156 tuples t and t' , *i.e.*, $\delta t = t \oplus t'$. By abuse of language, we say that a tuple δt is equal to 0
 157 whenever all its elements are equal to 0, *i.e.*, δt does not contain differences.

158 Given an operator o , some input/output differences are more likely to occur than others,
 159 and this is quantified by means of differential probabilities.

160 ► **Definition 3** (Differential probability of an operator). *The probability that an operator o*
 161 *transforms an input difference δt_{in} into an output difference δt_{out} is*

$$162 \quad p_o(\delta t_{out} | \delta t_{in}) = \frac{\#\{(t, t') \in [0, 1]^{k \cdot \#t_{in}(o)} \times [0, 1]^{k \cdot \#t_{in}(o)} : \delta t_{in} = t \oplus t' \wedge \delta t_{out} = o(t) \oplus o(t')\}}{2^{k \cdot \#t_{in}(o)}}$$

¹ ARX schemes use only modular Addition, Rotation and XOR.



■ **Figure 1** DAG of the first round of the AES for 128-bit keys. Bytes are represented with squares, and operators with circles. The input key and plaintext have 128 bits and are split into 16 bytes colored in blue and green, respectively. Yellow squares correspond to the text state after one encryption round. Pink squares correspond to the first round sub-key and are obtained from the blue squares by applying operations which are not displayed to avoid overloading the figure (these operations are: 16 XORs, 4 SubBytes, and 1 XOR with a constant).

163 This probability is equal to 0 or 1 for linear operators. More precisely, for any linear
 164 operator o , $p_o(\delta t_{out}|\delta t_{in}) = 1$ if $o(\delta t_{in}) = \delta t_{out}$ and $p_o(\delta t_{out}|\delta t_{in}) = 0$ otherwise. This
 165 comes from the fact that for any linear operator o and any input parameters t and t' ,
 166 $o(t) \oplus o(t') = o(t \oplus t')$.

167 When an operator o is not linear, p_o may be different from 0 and 1 and the only case
 168 where $p_o(\delta t_{out}|\delta t_{in})=1$ is when $\delta t_{in}=\delta t_{out}=0$. In all other cases, it is strictly smaller than 1.

169 ► **Example 4.** For the AES, all operators but SubBytes are linear. For SubBytes, the
 170 probability $p_S(\delta t_{out}|\delta t_{in})$ belongs to $\{0, 2^{-6}, 2^{-7}, 1\}$.

171 Let us now formally define what is an MDC.

172 ► **Definition 5 (MDC).** Given a DAG that describes a cipher, a differential characteristic
 173 is a function $\delta : P \setminus C \rightarrow [0, 1]^k$ that associates a differential sequence δx_i with every non-
 174 constant parameter $x_i \in P \setminus C$. The probability of a differential characteristic is obtained by
 175 multiplying, for each operator $o \in O$, the probability $p_o(\delta succ(o)|\delta pred(o))$ where δt denotes
 176 the tuple obtained by replacing every parameter x_i that occurs in t by δx_i if $x_i \in P \setminus C$, and
 177 by 0 if $x_i \in C$.

178 An MDC is a differential characteristic with maximum probability.

179 3.2 Truncated Differential Characteristics

180 MDCs are usually computed in two steps, as initially proposed by Knudsen in [20]: First, we
 181 search for TDCs, and then we compute MDCs associated with TDCs.

182 A TDC is a solution to an abstract problem. More precisely, the abstraction of a k-bit
 183 differential sequence δx is a Boolean value denoted ΔX such that $\Delta X = 1$ iff δx contains a

184 difference, *i.e.*, $\delta x \neq 0$. Similarly, the abstraction of a differential tuple $\delta t = (\delta x_1, \dots, \delta x_i)$
 185 is the Boolean tuple $\Delta t = (\Delta x_1, \dots, \Delta x_i)$ such that Δx_j is the abstraction of δx_j for each
 186 $j \in [1, i]$.

187 ► **Definition 6 (TDC).** *Given a bipartite DAG that describes a cipher, a TDC is a function*
 188 $\Delta : P \setminus C \rightarrow \{0, 1\}$ *that associates a Boolean value Δx_i with every non-constant parameter*
 189 $x_i \in P \setminus C$.

190 *A concretization of a TDC Δ is a differential characteristic δ such that, for each non-*
 191 *constant parameter $x \in P \setminus C$, $\Delta x = 0 \Leftrightarrow \delta x = 0$. Δ is concretizable if it has at least one*
 192 *concretization, the probability of which is different from 0.*

193 Finding a concretization of a TDC that has a maximal probability (or proving that the
 194 TDC cannot be concretized) is efficiently tackled by CP solvers thanks to table constraints
 195 (see, *e.g.*, [16]). However, there exists an exponential number of candidate TDCs with respect
 196 to the number of non-constant parameters in $P \setminus C$. Hence, the key point for an efficient
 197 solution process is to reduce as much as possible the number of candidate TDCs. This is
 198 done by adding constraints that prevent the generation of non concretizable TDCs as much
 199 as possible, without removing any concretizable TDC.

200 ► **Example 7 (XOR).** If $\delta y_1 = \delta x_1 \oplus \delta x_2$, then it is not possible to have only one sequence
 201 in $\{\delta x_1, \delta x_2, \delta y_1\}$ which contains a difference. Therefore, we can add the constraint $\Delta x_1 +$
 202 $\Delta x_2 + \Delta y_1 \neq 1$ for each XOR operator.

203 ► **Example 8 (MC).** There is no straightforward constraint that may be associated with
 204 *MC* as knowing which input parameters contain differences is not enough to know which
 205 output parameters contain differences: To answer this question, we must know the exact
 206 values of the input differences. However, *MC* usually satisfies the MDS property [25] that
 207 relates the number of input differences with the number of output differences. The exact
 208 definition of this relation depends on the constant coefficients $M_{i,j}$. For the AES, this relation
 209 is: among the four input differences $\delta x_1, \dots, \delta x_4$ and the four output differences $\delta y_1, \dots, \delta y_4$,
 210 either all differences are equal to 0, or at least five of them are different from 0. Hence, we
 211 can add the constraint $\sum_{i=1}^4 \Delta X_i + \Delta Y_i \in \{0, 5, 6, 7, 8\}$ for each *MC* operator.

212 ► **Example 9 (SR_s).** SR_s simply moves bytes. Therefore, we can add an equality constraint
 213 between the corresponding Boolean variables, *i.e.*, $\forall i \in [1, 4], \Delta y_i = \Delta x_{1+(i+s)\%4}$.

214 ► **Example 10 (S).** S is not linear, and we cannot deterministically compute the output
 215 difference δy_1 given the input difference δx_1 . However, as the look-up table is a bijection, we
 216 know that $\delta x_1 = 0 \Leftrightarrow \delta y_1 = 0$. Therefore, we can add the constraint $\Delta x_1 = \Delta y_1$ for each S
 217 operator.

218 3.3 Definition of Step1-opt and Step1-enum Problems

219 As the probability $p_o(\delta t_{out} | \delta t_{in})$ associated with a non-linear operator o is equal to 1 whenever
 220 $\delta t_{out} = \delta t_{in} = 0$ whereas it is very small otherwise (*e.g.*, smaller than or equal to 2^{-6} for
 221 the AES Sbox), we can compute an upper bound on an MDC by computing a lower bound
 222 on the number of *active* non-linear operators in a TDC, where an operator is said to be
 223 active whenever its input/output differential tuples are different from 0. More precisely,
 224 let $s(\Delta)$ be the number of active non-linear operators in a TDC Δ (*i.e.*, $s(\Delta) = \#\{o \in$
 225 $O : o \text{ is not linear} \wedge \delta pred(o) \neq 0\}$), and let s^* be the minimal value of $s(\Delta)$ for all possible
 226 TDCs Δ . If the maximal probability of an active non-linear operator is equal to p , then

227 the probability of an MDC is upper bounded by p^{s^*} . For example, for the AES this upper
 228 bound is $2^{-6 \cdot s^*}$. In some cases, this upper bound is small enough to ensure the security of
 229 the cipher with respect to differential attacks, and it is not necessary to actually compute
 230 MDCs. Most papers that introduce new ciphering algorithms demonstrate the security of
 231 their cipher with respect to differential attacks only by computing this upper bound (*e.g.*,
 232 [5]). When the upper bound p^{s^*} is large enough to allow mounting differential attacks, we
 233 have to enumerate all possible TDCs that have a given number of active non-linear operators,
 234 and we have to search for an MDC for each of these TDCs.

235 *Step1-opt* is the problem that aims at computing s^* whereas *Step1-enum* is the problem
 236 that aims at enumerating all TDCs that have a given number of active non-linear operators.

237 There exist different kinds of differential attacks, depending on where differences can be
 238 injected. In this paper, we consider *Single-key* attacks, where differences are only injected in
 239 the clear text (*i.e.*, for each k -bit sequence x_i coming from the input key, we have $\Delta x_i = 0$),
 240 and *Related-key* attacks, where differences can be injected in both the plaintext and the key.

241 3.4 Existing Approaches for Solving Step1-opt and Step1-enum

242 Two dedicated approaches have been proposed to solve these problems: An approach based
 243 on dynamic programming (*e.g.*, for AES [13] and Skinny [11]), and an approach based on
 244 Branch & Bound (*e.g.*, for AES [7]). The dynamic programming approach is rather efficient,
 245 but it runs out of memory for large instances (*e.g.*, when the key has more than 128 bits
 246 for the AES); the Branch & Bound approach has no memory issue but needs weeks to solve
 247 middle size instances and cannot be used to solve all instances within a reasonable amount
 248 of time.

249 Also, ILP, CP, or SAT are commonly used to solve Step1-opt and Step1-enum: on
 250 Skinny [11], Craft [18], Deoxys [26, 10], AES [23, 16], and Midori [15], for example.

251 While ILP/CP/SAT approaches require less programming work than dedicated ones,
 252 they still require designing mathematical models. In particular, it is necessary to find
 253 constraints that limit the number of non concretizable TDCs as much as possible, and this
 254 can be time-consuming. In this paper, we present an automatic way to generate models for
 255 Step1-opt and Step1-enum.

256 4 Description of a Symmetric Block Cipher with Tagada

257 The DAG associated with a cipher (see Section 2) must be described in a JSON file. This
 258 file first specifies a list of parameters such that each parameter has one attribute, *i.e.*, its
 259 name (which must be unique). Then, it specifies a list of operators such that each operator
 260 has three attributes, *i.e.*, its list of input parameters, its list of output parameters, and its
 261 UID (a unique identifier) that must correspond to an executable function.

262 ► **Example 11** (JSON representation of a XOR followed by a SubBytes).

```
263 { "parameters": [ {"name": "X00"}, {"name": "K00"}, {"name": "ARK00"}, {"name": "S00"} ],
264   "operators": [ {"uid": "xor_2_1", "in": ["X00", "K00"], "out": ["ARK00"]},
265                 {"uid": "s_1_1", "in": ["ARK00"], "out": ["S00"]} ] }
```

266 The UIDs `xor_2_1` and `s_1_1` correspond to computable functions: `xor_2_1` reads two k -bit
 267 sequences and outputs their XOR, and `s_1_1` reads one k -bit sequence and returns the
 268 substitution associated with it according to the S-Box.

269 Some patterns may be repeated in the DAG. For example, let us consider the DAG describing
 270 the first round of the AES displayed in Fig. 1. At the top level of this DAG, there are 16 XORs

271 which correspond to the *AddRoundKey* (*ARK*) step, where each byte of the text (in blue) is
 272 XORed with the corresponding byte of the key (in green). As it is tedious to write 16 times
 273 the JSON representation of one XOR operation, TAGADA provides functions corresponding to
 274 meta-operators, where a meta-operator is a classical combination of operators.

275 ► **Example 12** (ARK meta-operator). The ARK meta-operator has 3 groups of parameters:
 276 the first group corresponds to the 16 input text bytes; the second to the 16 input key
 277 bytes; and the third to the 16 output parameters. This meta-operator generates the JSON
 278 description of 16 XORs such that each XOR has two input parameters coming from the first
 279 and the second group, and one output parameter from the third group.

280 These meta-operators strongly simplify the definition of the JSON file. For example, the
 281 JSON file corresponding to 4 rounds of the AES contains 364 parameters and 288 operators.
 282 This file is generated by approximately 100 lines of code when using meta-operators.

283 To test the JSON file, TAGADA provides a function that has three input parameters,
 284 *i.e.*, a JSON file F describing a cipher, a plaintext X and a key K , and that returns the
 285 ciphertext obtained when ciphering X with K according to F (this computation is done by
 286 performing a topological sort to order DAG operators, and applying operators in this order).
 287 This function allows us to test the correctness of the JSON file with the *initialization vectors*,
 288 *i.e.*, a set of (key, plaintext, ciphertext) triples that are usually provided by cipher authors
 289 to validate that implementations are correct. Moreover, these vectors are mandatory for the
 290 authors of all candidates to NIST's competitions.

291 5 Automatic Generation of Models with Tagada

292 We show how TAGADA automatically generates state-of-the-art MiniZinc models for solving
 293 Step1-opt and Step1-enum problems given JSON files that describe ciphers. This is done
 294 in four steps: (i) generation of constraints from the black boxes associated with operators
 295 (Section 5.1); (ii) simplification of the DAG (Section 5.2); (iii) extension of the DAG
 296 (Section 5.3); and (iv) generation of the model from the DAG and the constraints (Section 5.4).

297 5.1 Automatic Generation of Constraints

298 As pointed out in Section 3.2, the key point for an efficient process is to tighten the abstraction
 299 to prevent as much as possible the generation of non concretizable TDCs. For non-linear
 300 operators, we add a constraint to ensure that $\Delta x_1 = \Delta y_1$ where x_1 is the input parameter
 301 and y_1 is the output parameter because $\delta x_1 = 0 \Leftrightarrow \delta y_1 = 0$ for all non-linear operators.

302 For linear operators, we have to add constraints and, in all existing works, these constraints
 303 have been manually derived from a careful analysis of operators, as illustrated in Ex. 7 to 9.
 304 While this has led to efficient models, this was also time-consuming and error-prone. Hence,
 305 we propose to automatically generate table constraints for which domain consistency can be
 306 efficiently achieved. Tables are generated by using the functions that provide operational
 307 definitions of these operators. More precisely, the constraint associated with an operator o is
 308 the relation \mathcal{R}_o of arity $\#t_{in}(o) + \#t_{out}(o)$ which contains every boolean tuple corresponding
 309 to possible difference positions for the input/output parameters of o . As $o(t) \oplus o(t') = o(t \oplus t')$
 310 for any $t, t' \in [0, 1]^{k * \#t_{in}(o)}$, we can build \mathcal{R}_o from the black-box definition of o as follows.

311 ► **Definition 13** (Relation \mathcal{R}_o associated with an operator o).

312 $\mathcal{R}_o = \{(\Delta(x_1), \dots, \Delta(x_{\#t_{in}(o)}), \Delta(y_1), \dots, \Delta(y_{\#t_{out}(o)})) : \exists(x_1, \dots, x_{\#t_{in}(o)}) \in [0, 1]^{k * \#t_{in}(o)},$
 313 $(y_1, \dots, y_{\#t_{out}(o)}) = o(x_1, \dots, x_{\#t_{in}(o)})\}$ where $\forall x \in [0, 1]^k$, $\Delta(x)$ denotes the Boolean abstrac-
 314 tion of x , *i.e.*, $\Delta(x) = 0 \Leftrightarrow x = 0$.

315 To compute this relation, we must (i) enumerate every possible k -bit sequence for
 316 every input parameter of o ; (ii) for each enumerated combination of input parameters,
 317 call o to compute output parameter values; and (iii) compute the abstract Boolean values
 318 $\Delta(x_i)$ and $\Delta(y_j)$ from their corresponding concrete values x_i and y_j . Hence, the time
 319 complexity for building \mathcal{R}_o is $\mathcal{O}(t \cdot 2^{k \cdot \#t_{in}(o)})$ where t is the time complexity of o . Moreover,
 320 k is either equal to 4 or 8, and the number of input parameters, $\#t_{in}(o)$, is usually very
 321 small: $\#t_{in}(o)$ is always smaller than or equal to four for all ciphers we are aware of.
 322 Hence, the relation is rather quickly computed. In the worst case, the relation contains all
 323 possible binary tuples of arity $\#t_{in}(o) + \#t_{out}(o)$. Hence, the space complexity of \mathcal{R}_o is
 324 $\mathcal{O}((\#t_{in}(o) + \#t_{out}(o)) \cdot 2^{\#t_{in}(o) + \#t_{out}(o)})$.

325 Note that the relation is computed only once for each black box (identified by its UID),
 326 even if the operator is used more than once in the DAG. Also, some operators are shared by
 327 multiple ciphers (such as XOR which is used by all ciphers). In this case, we only need to
 328 compute the relation once, and we can memorize it for future usage.

329 ► **Example 14** (\mathcal{R}_{xor}). The relation associated with XOR contains all triples $(\Delta(x_1), \Delta(x_2),$
 330 $\Delta(x_1 \oplus x_2))$ such that $x_1, x_2 \in [0, 1]^k$. We obtain the following relation: $\mathcal{R}_{xor} = \{0, 0, 0), (0, 1,$
 331 $1), (1, 0, 1), (1, 1, 0), (1, 1, 1)\}$. Note that the constraint $(\Delta x_1, \Delta x_2, \Delta y_1) \in \mathcal{R}_{xor}$ has exactly
 332 the same semantics as the constraint $\Delta x_1 + \Delta x_2 + \Delta y_1 \neq 1$ which is usually added to model
 333 XORs in Step1-opt and Step1-enum models.

334 ► **Example 15** (\mathcal{R}_{MC}). The relation associated with MC contains all tuples $(\Delta(x_1), \Delta(x_2),$
 335 $\Delta(x_3), \Delta(x_4), \Delta(y_1), \Delta(y_2), \Delta(y_3), \Delta(y_4))$ such that $\forall i \in [1, 4], y_i = (M_{i,1} \otimes x_1) \oplus (M_{i,2} \otimes$
 336 $x_2) \oplus (M_{i,3} \otimes x_3) \oplus (M_{i,4} \otimes x_4)$. This relation, for the AES MixColumns, contains 102 tuples
 337 and has exactly the same semantics as the constraint associated with the famous MDS
 338 property, *i.e.*, it contains only tuples such that the number of 1s belongs to $\{0, 5, 6, 7, 8\}$.

339 5.2 Simplification of the DAG

340 Before generating a MiniZinc model from the DAG, we simplify it by applying shaving rules
 341 that are described in this section. Each rule removes one or more vertices (and their incident
 342 edges), and rules are iteratively applied until reaching a fixed point.

343 Rule 1: Merging Equal Parameters

344 When building a relation \mathcal{R}_o from the black box that defines o , we can search for every couple
 345 of input/output parameters (x_i, y_j) with $i \in [1, \#t_{in}(o)]$ and $j \in [1, \#t_{out}(o)]$ such that x_i
 346 is always equal to y_j : before starting the construction of the relation, we initialize a Boolean
 347 variable eq_{x_i, y_j} to true; then, for each generated tuple of input parameters, if $x_i \neq y_j$ we set
 348 eq_{x_i, y_j} to false. This does not change the time complexity for building the relation.

349 We use a list L_{eq} to store all couples of parameter vertices that are related by an equality
 350 relation. Before starting the shaving process, L_{eq} is initialized by traversing the DAG: for
 351 each operator vertex o and each couple of parameter vertices $(x_i, y_j) \in pred(o) \times succ(o)$, if
 352 $eq_{x_i, y_j} = true$, we add (x_i, y_j) to L_{eq} . Rule 1 is triggered whenever L_{eq} is not empty, and it
 353 is defined as follows.

354 ► **Definition 16** (Rule 1). *If $L_{eq} \neq \emptyset$, then (i) compute equivalence classes corresponding*
 355 *to all binary equality relations contained in L_{eq} (using a union-find data structure) and*
 356 *reinitialize L_{eq} to the empty set, (ii) merge all vertices of the DAG that belong to a same*
 357 *equivalence class, and (iii) remove every operator vertex that is no longer connected to a*
 358 *parameter vertex.*

359 ► **Example 17** (SR_s). When building the relation \mathcal{R}_{SR_s} , we infer that eq_{x_i, y_j} is true whenever
 360 $j = 1 + (i + s) \% 4$. When considering the DAG displayed in Fig. 1, this allows us to merge
 361 each of the four predecessors of SR_s vertices with its corresponding successor and, finally, to
 362 remove each SR_s vertex.

363 Rule 2: Suppressing Constant Parameters

364 When an operator vertex o has an input parameter x_i that has a constant value c , then this
 365 parameter is replaced with 0 in the differential characteristic because $c \oplus c = 0$ (see Def. 3)
 366 and, therefore, it can be removed from the DAG. Moreover, if all input parameters of o are
 367 constants, its outputs are also constants and o can be removed from the DAG.

368 We use a list L_C to store all parameter vertices that have constant values. Before starting
 369 the shaving process, L_C is initialized with the set C of constant parameters. Rule 2 is
 370 triggered whenever L_C is not empty, and it is defined as follows.

371 ► **Definition 18** (Rule 2). *If $L_C \neq \emptyset$, then repeat the three following steps until $L_C = \emptyset$:*
 372 *(i) choose one operator vertex o such that $pred(o) \cap L_C \neq \emptyset$;*
 373 *(ii) remove from the DAG and from L_C every parameter vertex $x_i \in L_C \cap pred(o)$;*
 374 *(iii) if $pred(o) = \emptyset$, then remove o from the DAG and add every parameter vertex in $succ(o)$*
 375 *to L_C , else update the relation \mathcal{R}_o and update L_{eq} if new equality relations can be inferred;*

376 ► **Example 19** (XOR with a constant value). Let us consider a XOR operator with one output
 377 parameter y_1 and two input parameters x_1 and x_2 such that x_1 is a constant (*i.e.*, $x_1 \in C$).
 378 This operator is used in the key schedule of the AES, for example. In this case, x_1 is removed
 379 from the DAG, the relation associated with this operator becomes $\{(0, 0), (1, 1)\}$, and we
 380 add the couple (x_2, y_1) to the list L_{eq} .

381 Rule 3: Suppressing Free Parameters

382 When an output parameter vertex x has no successor and its predecessor o is a linear operator,
 383 then we can remove both o and x from the DAG because we can deterministically compute
 384 the output difference δx of o given the differences of all input parameters of o .

385 Similarly, when an input parameter vertex x has no predecessor, and it has only one
 386 successor which is a linear operator, we can also remove both o and x from the DAG because
 387 we can deterministically compute the input difference δx of o given the differences of all
 388 other input parameters of o and the difference of its output parameter.

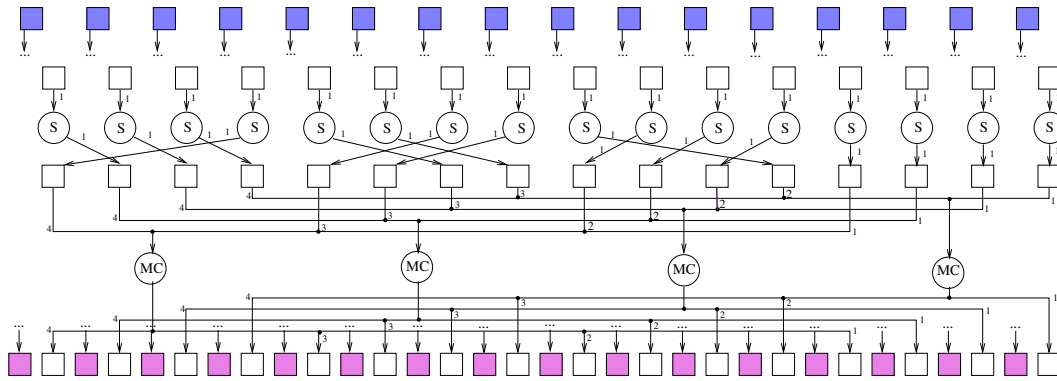
389 More formally, Rule 3 is defined as follows.

390 ► **Definition 20** (Rule 3). *If there exists a parameter vertex x such that the out-degree of x*
 391 *is equal to 0 and the predecessor of x is a linear operator, then remove x and the predecessor*
 392 *of x from the DAG.*

393 *If there exists a parameter vertex x such that the in-degree of x is equal to 0, the out-degree*
 394 *of x is equal to 1, and the successor of x is a linear operator, then remove x and the successor*
 395 *of x from the DAG.*

396 ► **Example 21.** Let us consider the DAG displayed in Fig. 1. Every yellow vertex has no
 397 successor and its predecessor is a linear operator (*i.e.*, a XOR). Hence, we can remove all
 398 yellow vertices, and all XOR operators that are predecessors of yellow vertices.

399 Also, every green vertex (corresponding to one byte of the plaintext) has no predecessor
 400 and one successor which is a linear operator (*i.e.*, a XOR). Hence, we can remove all green
 401 vertices, and all XOR operators that are successors of green vertices.



■ **Figure 2** Shaved DAG obtained from the DAG of Fig. 1 after applying Rules 1, 2, and 3.

402 Note that we cannot remove vertices that precede S operators, though they have no more
 403 predecessors once we have removed XOR operators that succeeded green vertices, because S
 404 is not linear. The shaved DAG obtained from the DAG of Fig. 1 after applying Rules 1, 2,
 405 and 3 is displayed in Fig. 2. We do not apply the shaving rules on vertices associated with
 406 the key vertices (in blue and pink) as we have not displayed the operator vertices that are
 407 used to compute pink vertices from blue ones in Fig. 1.

408 5.3 Extension of the DAG

409 A basic CP model may be generated from the shaved DAG (this will be explained in
 410 Section 5.4). However, the resulting model is often not tight enough, *i.e.*, the bound provided
 411 by Step1-opt is smaller than the actual value and/or many solutions of Step1-enum cannot
 412 be concretized into differential characteristics with strictly positive probabilities. In this
 413 section, we show how to tighten this model by extending the DAG.

414 5.3.1 Generation of New Vertices and Edges from Existing Operators

415 In [17, 16, 23], Step1-opt and Step1-enum models are tightened by exploiting the fact that,
 416 if $t_1 = MC(t_2)$ and $t_3 = MC(t_4)$ (where $t_1, t_2, t_3,$ and t_4 are tuples of arity 4), then
 417 $t_1 \oplus t_3 = MC(t_2 \oplus t_4)$. As a consequence, the MDS property also holds on $t_1 \oplus t_3$ and $t_2 \oplus t_4$,
 418 *i.e.*, the number of k -bit sequences in $t_1 \oplus t_3$ and $t_2 \oplus t_4$ that are different from 0 is either
 419 equal to 0 or strictly greater than 4. Hence, a new variable (called *diff* variable in [16]) is
 420 added for each parameter of each couple of MC operators. These *diff* variables are related
 421 with initial parameters by adding XOR constraints. Finally, constraints that ensure the MDS
 422 property are added for these new *diff* variables.

423 In TAGADA, we generalize this idea to all linear operators. Indeed, for any kind of linear
 424 operator identified by its UID u , we have $u(t_1) \oplus u(t_2) = u(t_1 \oplus t_2)$. Therefore, for each
 425 pair of operator vertices $o_1, o_2 \in O$ such that the UID of o_1 and o_2 is u , we can add a new
 426 operator vertex whose UID is u and whose input and output parameter vertices are obtained
 427 by XORing input and output parameter vertices of o_1 and o_2 . More precisely, let $pred(o_1) =$
 428 $(x_{1,1}, \dots, x_{1,\#t_{in}(u)})$, $succ(o_1) = (y_{1,1}, \dots, y_{1,\#t_{out}(u)})$, $pred(o_2) = (x_{2,1}, \dots, x_{2,\#t_{in}(u)})$, and
 429 $succ(o_2) = (y_{2,1}, \dots, y_{2,\#t_{out}(u)})$. We extend the DAG as follows:

- 430 ■ For each $i \in [1, \#t_{in}(u)]$, we add a new parameter vertex $x_{3,i}$ corresponding to the result
 431 of XORing $x_{1,i}$ and $x_{2,i}$, *i.e.*, we add a new XOR vertex whose predecessors are $x_{1,i}$ and
 432 $x_{2,i}$ and whose successor is $x_{3,i}$;

- 433 ■ For each $j \in [1, \#t_{out}(u)]$, we add a new parameter vertex $y_{3,j}$ corresponding to the result
 434 of XORing $y_{1,j}$ and $y_{2,j}$, *i.e.*, we add a new XOR vertex whose predecessors are $x_{1,i}$ and
 435 $x_{2,i}$ and whose successor is $x_{3,i}$;
 - 436 ■ We add a new operator vertex o_3 such that the UID of o_3 is u , the predecessors of o_3 are
 437 $x_{3,1}, \dots, x_{3, \#t_{in}(u)}$, and the successors of o_3 are $y_{3,1}, \dots, y_{3, \#t_{out}(u)}$.
- 438 This may be done for each kind of linear operator except XOR (as this is useless in this case).
 439 As this step may drastically increase the size of the DAG, it is optional, and the user can
 440 choose the kind of linear operator that should be considered for this step.

441 5.3.2 Generation of New XORs

442 XOR equations may be combined to generate new equations. For example, consider two XOR
 443 equations: $a \oplus b \oplus c = 0$, and $b \oplus c \oplus d = 0$. By XORing these two equations, we obtain a
 444 new equation $a \oplus d = 0$. This new equation is redundant when computing MDCs, but it
 445 tightens the abstraction when computing TDCs. Indeed, let Δi be the boolean abstraction of
 446 each k -bit sequence $i \in \{a, b, c, d\}$. If we only post the two constraints $(\Delta a, \Delta b, \Delta c) \in \mathcal{R}_{xor}$
 447 and $(\Delta b, \Delta c, \Delta d) \in \mathcal{R}_{xor}$ (where \mathcal{R}_{xor} is the relation defined in Ex. 14), then it is possible
 448 to assign Δa , Δb , and Δc to 1, and Δd to 0 because $(1, 1, 1) \in \mathcal{R}_{xor}$ and $(1, 1, 0) \in \mathcal{R}_{xor}$.
 449 However, if we add the constraint $(\Delta a, \Delta d) \in \{(0, 0), (1, 1)\}$, then this assignment is no
 450 longer consistent.

451 This trick was introduced in [16] for the AES, but it has been limited to XORs that occur
 452 in the key schedule. In TAGADA, we generalize it to all XORs. Let $adj(o) = pred(o) \cup succ(o)$
 453 be the set of input and output parameters of an operator vertex o . For each couple of operator
 454 vertices (o_1, o_2) such that both o_1 and o_2 are XORs that share at least one common parameter
 455 (*i.e.*, $adj(o_1) \cap adj(o_2) \neq \emptyset$), we compute the set $S = (adj(o_1) \cup adj(o_2)) \setminus (adj(o_1) \cap adj(o_2))$
 456 (corresponding to parameters that are adjacent to o_1 or o_2 but not to both o_1 and o_2). If
 457 S does not contain more than n_{max} parameters, then we add a new operator vertex o_3 to
 458 the DAG, and we add an edge between each parameter vertex in S and o . This process is
 459 recursively applied, until no more vertex can be added.

460 n_{max} is a given integer value that is used to control the growth of the DAG: when $n_{max} = 0$,
 461 no new XOR operator is added to the DAG; the larger n_{max} , the more XOR operators are
 462 added.

463 For all possible values of $\#S \in [0, n_{max}]$, we have to generate the relation associated with
 464 a XOR of $\#S$ parameters, as described in Section 5.1. Also, we infer equality relations and
 465 apply Rule 1 (as described in Section 5.2) to merge vertices of the DAG that belong to a
 466 same equivalence class.

467 5.4 Generation of the MiniZinc Model from the DAG

468 Given a DAG, we generate a MiniZinc model as follows:

- 469 ■ We declare a Boolean variable Δx for each parameter vertex x of the DAG;
- 470 ■ We add a constraint $\Delta(pred(o), succ(o)) \in \mathcal{R}_o$ for each operator vertex o (where $\Delta(pred(o),$
 471 $succ(o))$ is the tuple of Boolean variables associated with parameters in $pred(o)$ and
 472 $succ(o)$);
- 473 ■ We declare an integer variable s which corresponds to the number of active non-linear
 474 operators in the TDC, and we add a constraint $s = \sum_{x \in NL} \Delta x$ where NL contains the
 475 set of parameter vertices that are predecessors of a non-linear operator vertex.

model	Midori (35)			AES (25)			SKINNY (56)			CRAFT (38)		
	#d	#o	#e	#d	#o	#e	#d	#o	#e	#d	#o	#e
$n_{max}=0$	18			12			0	24	22	0	38	38
$n_{max}=1$	18			12			0	25	22	0	38	38
$n_{max}=2$	18			12			0	25	22	0	38	38
$n_{max}=3$	18			12			0	25	22	0	38	38
$n_{max}=4$	18			12			0	24	22	0	38	38
$n_{max}=5$	–	–	–	12			–	–	–	–	–	–
$n_{max}=0$ MC	18			12			–	–	–	0	38	38
$n_{max}=1$ MC	18			12			–	–	–	0	38	38
$n_{max}=2$ MC	18			12			–	–	–	0	38	38
$n_{max}=3$ MC	18			12			–	–	–	0	38	38
$n_{max}=4$ MC	0	35	34	0	23	21	–	–	–	0	37	37
$n_{max}=5$ MC	–	–	–	0	24	21	–	–	–	–	–	–

Table 1 Model performance summary of Picat-SAT on the 35 Midori instances, 25 AES instances, 56 SKINNY instances and 38 CRAFT instances, for different values of n_{max} ranging from 0 to 5. The 6 first (resp. last) rows give results without (resp. with) selecting MC. #d corresponds to the number of instances where the model is not tight enough. When #d=0, we report the number of instances that are solved within 1 hour for Step1-opt (#o) and Step1-enum (#e), and we highlight the best values. We report – when models have not been generated because DAGs are too large.

476 For Step1-opt, the goal is to minimize s , and we add the constraint $s \geq 1$ because TDCs
 477 must contain at least one active non-linear operator. For Step1-enum, s is assigned to the
 478 number of active non-linear operators, and the goal is to enumerate all solutions.

479 6 Experimental Results

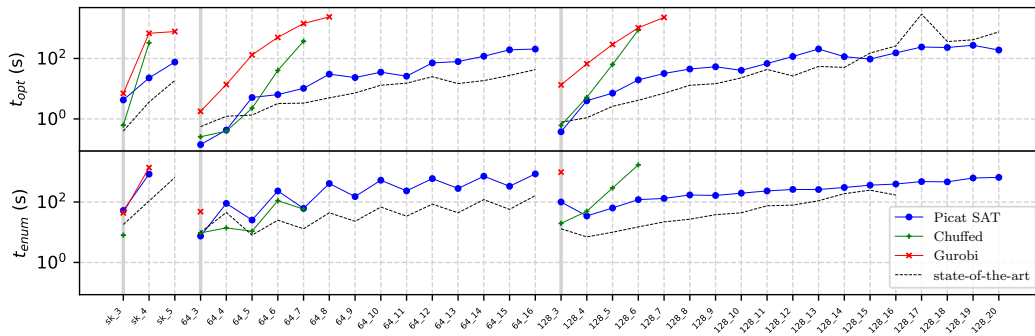
480 We performed all experiments on a PC with a Xeon Gold 5118 (2.30 GHz) with 24 cores and
 481 32 GB of RAM. Each experiment used only one thread, and we ran 20 of them in parallel to
 482 speed up the computations. All the source-code and results are available online^{2 3}.

483 We consider four symmetric block ciphers for which there exist recent differential crypt-
 484 analysis results, *i.e.*, the AES [16], Midori [14], Skinny [11], and Craft [18]. For each cipher,
 485 there are different instances that are obtained by considering either single-key or related-key
 486 attacks, by changing the size of the key for related-key attacks of ciphers that have different
 487 key lengths (*i.e.*, 64 and 128 for Midori, 128, 192, and 256 for the AES), and by changing the
 488 number r of rounds of the ciphering process, starting from $r = 3$ up to the largest value for r
 489 considered in the literature. We obtain 35 (resp. 25, 56, and 38) instances for Midori (resp.
 490 the AES, Skinny, and Craft). Finally, for each instance, we solve two different problems, *i.e.*,
 491 Step1-opt and Step1-enum. Hence, our benchmark contains 308 instances.

492 TAGADA has a parameter n_{max} that is used to control the maximum size of new generated
 493 XOR equations (see Section 5.3.2). It is also possible to select the linear operators for which
 494 we infer new vertices and edges as explained in Section 5.3.1. In the four considered ciphers,
 495 the only linear operator that can be selected is *MC* as *SR* is removed during the DAG
 496 shaving step. Increasing n_{max} and/or selecting *MC* tightens the abstraction, but it also

² Tagada: https://gitlab.limos.fr/iia_lulibral/tagada/

³ models and results: https://gitlab.limos.fr/iia_lulibral/experiment-results



■ **Figure 3** CPU time of Picat-SAT, Chuffed and Gurobi on the model generated by TAGADA for Midori instances when $n_{max} = 4$ and MC is selected (top plot for Step1-opt and bottom plot for Step1-enum). State-of-the art is the handcrafted model of [14] run with Picat-SAT.

497 increases the number of variables and constraints in the generated model.

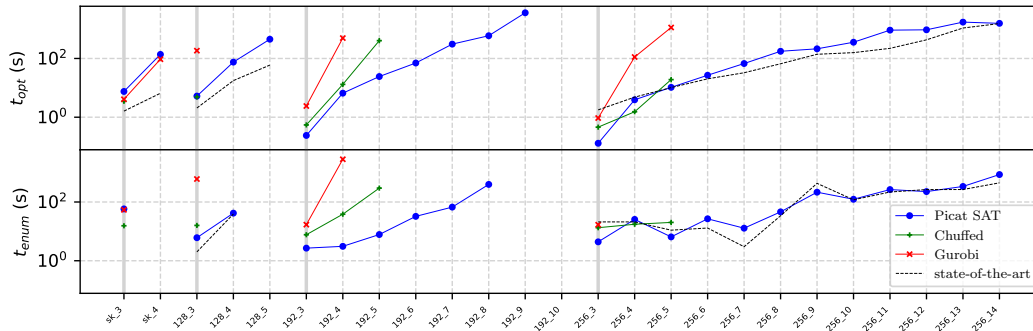
498 In Table 1, we report the number of instances for which the generated model is not tight
 499 enough (*i.e.*, the bound computed by Step1-opt is smaller than the best known bound) for
 500 different values of n_{max} and with or without selecting MC . This shows us that the best
 501 parameter setting depends on the cipher: For Midori and the AES, it is necessary to select
 502 MC and to set n_{max} to a value larger than or equal to 4 to generate a model that is tight
 503 enough for all instances; For Skinny and Craft, the generated model is tight enough even
 504 when $n_{max} = 0$ and MC is not selected.

505 In Table 1, we also report the number of instances that are solved within one hour of
 506 CPU time by Picat-SAT [27] whenever the model is tight enough (it is meaningless to report
 507 these results when models are not tight enough, as they do not solve the same problem).
 508 When increasing n_{max} , the model has more constraints, and the number of new constraints
 509 grows exponentially with n_{max} . In [16] and [14], this parameter has been fixed to 4 for the
 510 handcrafted models, and this seems to be a rather good setting. However, for the AES,
 511 one more instance is solved when increasing n_{max} to 5, and for Skinny one more instance
 512 is solved when decreasing n_{max} to 3. For Midori, Skinny and Craft, when $n_{max} = 5$ the
 513 number of new constraints is so large that we have not run the resulting models. As models
 514 are automatically generated by TAGADA, the user can easily fiddle with parameters to find
 515 the settings that generate the tightest and most efficient models for a cipher.

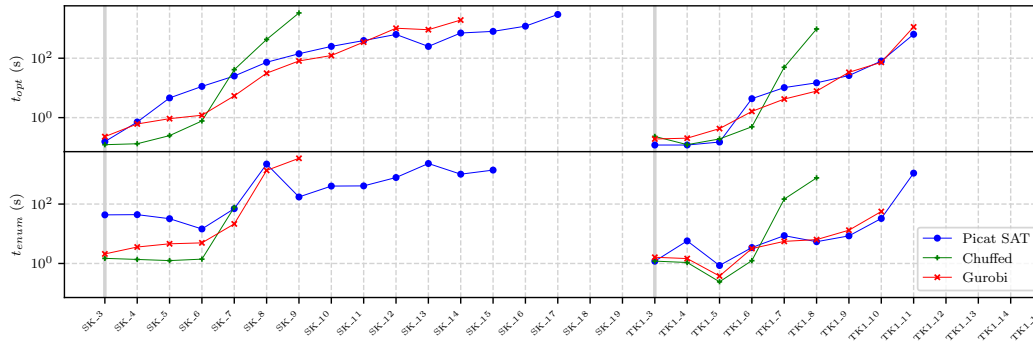
516 In Fig. 3 to 6, we display results, on a per-instance basis, and for three different kinds of
 517 solvers, *i.e.* Picat-SAT [27] (that generates a SAT instance from the MiniZinc model and
 518 uses Lingeling to solve it), Gurobi [22] (which is an ILP solver), and Chuffed [9] (which is
 519 a CP solver with lazy clause generation). For these figures, we report results for the best
 520 parameter setting for each cipher, *i.e.*, $n_{max} = 4$ and MC is selected for Midori, $n_{max} = 5$
 521 and MC is selected for the AES, $n_{max} = 0$ and MC is not selected for Skinny and Craft.

522 Picat-SAT is usually more efficient than Chuffed and Gurobi. However, Chuffed is often
 523 faster on small instances, and Gurobi is the best performing solver on many Craft instances.

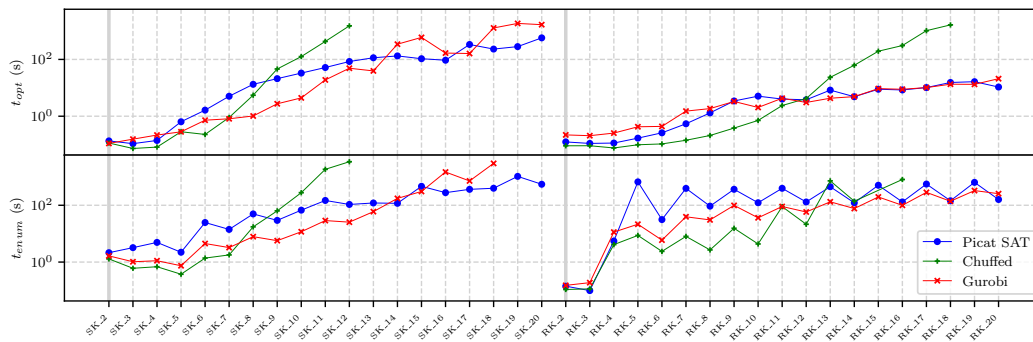
524 The MiniZinc models for the AES and Midori described in [16] and [14] are publicly
 525 available, and we compare our automatically generated models with these handcrafted models
 526 (we only report results with Picat-SAT in this case as this is the best performing solver).
 527 However, for instances of AES-192 we do not report results obtained with the model of [16]
 528 because it does not solve the same problem: for these instances, the model of [16] does not
 529 integrate in the objective function the S-boxes of the last round, which is an error of this



■ **Figure 4** CPU time of Picat-SAT, Chuffed and Gurobi on the model generated by TAGADA for AES instances when $n_{max} = 5$ and MC is selected (top plot for Step1-opt and bottom plot for Step1-enum). State-of-the art is the handcrafted model of [16] run with Picat-SAT.



■ **Figure 5** CPU time of Picat-SAT, Chuffed and Gurobi on the model generated by TAGADA for Skinny when $n_{max} = 0$ and MC is not selected (top plot for Step1-opt and bottom plot for Step1-enum).



■ **Figure 6** CPU time of Picat-SAT, Chuffed and Gurobi on the model generated by TAGADA for Craft when $n_{max} = 0$ and MC is not selected (top plot for Step1-opt and bottom plot for Step1-enum).

530 model for this particular case. For both Midori and the AES, models automatically generated
 531 with TAGADA are competitive with state-of-the-art handcrafted models. The largest Midori
 532 instances (when the key has 128 bits and the number of rounds is greater than 17) cannot be
 533 solved within one hour by the model of [14] whereas the TAGADA model solves them. This
 534 is remarkable because it takes weeks/months for a researcher to design these handcrafted
 535 models. Moreover, with TAGADA we can check that the description of the cipher is correct
 536 (as explained in Section 4), and the model is automatically generated from this description
 537 without any human action (except parameter selection).

538 For Skinny, the most efficient approach is a dedicated dynamic program [11]. However,
 539 this approach consumes huge amounts of memory (more than 700 GB of RAM). In [11], a
 540 MiniZinc model is also described, and results obtained with Picat-SAT are reported. The
 541 number of instances solved by this approach within one hour on a server composed of $2 \times$ AMD
 542 EPYC7742 64-Core is the same as with our TAGADA model when using Picat-SAT, *i.e.*, 22.

543 Finally, for Craft, [5] only reports optimal solutions of Step1-opt and does not report
 544 CPU times. Our TAGADA model has found the same solutions as those of [5].

545 **7 Conclusion**

546 In this article, we present TAGADA, a tool for automatically generating MiniZinc models for
 547 solving differential cryptanalysis problems given the description of a symmetric block cipher.
 548 The description is based on a unifying framework, *i.e.*, a DAG that describes how operators
 549 are combined and black-boxes that give an operational definition of operators.

550 This description allows us to perform a correctness verification using initialization vectors
 551 and comparing the behavior of our implementation with reference implementations found in
 552 the literature, limiting the possible bugs.

553 Then, for each black box operator, we perform an exhaustive search of its input and output
 554 values to infer a relation that represents a provably optimal abstraction for this operator.
 555 The DAG is further modified by removing some parts that are not useful for differential
 556 attacks, and by adding new operators that tighten the model. Finally, the MiniZinc model is
 557 generated from the relations and the DAG.

558 We experimentally compare automatically generated models with state-of-the-art ap-
 559 proaches on four ciphers (Midori, AES, Skinny, Craft) and on two types of attacks (Single-Key
 560 and Related-Key). For all scenarios, our models find the same solutions as hand-crafted
 561 models, and they have similar running times.

562 While the models generated by TAGADA have the same tightness and performance as
 563 state-of-the-art hand-crafted models, MIP/CP/SAT solvers still struggle to solve the largest
 564 instances. Recently, some ad-hoc dynamic programming algorithms have been proposed (for
 565 instance, on Skinny [11]), and show better scale-up properties though they have high space
 566 complexities. Hence, we plan to study the possibility of integrating dynamic programming
 567 approaches within TAGADA.

568 Also, we plan to integrate other differential attacks than single-key and related-key (*i.e.*,
 569 related-tweak, related-tweakey and boomerang attacks), and to extend TAGADA so that it
 570 also generates models for computing MDCs given TDCs. Of course, we will use TAGADA to
 571 analyze the recent ten finalists of NIST's competition, as there is a need to provide quickly
 572 differential attacks (or prove the robustness of the cipher against these attacks).

References

- 573 ———
- 574 1 S. Banik, A. Bogdanov, T. Isobe, K. Shibutani, H. Hiwatari, T. Akishita, and F. Regazzoni.
575 Midori: A block cipher for low energy. In *ASIACRYPT*, volume 9453 of *LNCS*, pages 411–436.
576 Springer, 2015.
- 577 2 Subhadeep Banik, Sumit Kumar Pandey, Thomas Peyrin, Yu Sasaki, Siang Meng Sim, and
578 Yosuke Todo. Gift: a small present. In *International Conference on Cryptographic Hardware
579 and Embedded Systems*, pages 321–345. Springer, 2017.
- 580 3 Ray Beaulieu, Stefan Treatman-Clark, Douglas Shors, Bryan Weeks, Jason Smith, and Louis
581 Wingers. The simon and speck lightweight block ciphers. In *2015 52nd ACM/EDAC/IEEE
582 Design Automation Conference (DAC)*, pages 1–6. IEEE, 2015.
- 583 4 Christof Beierle, Jérémy Jean, Stefan Kölbl, Gregor Leander, Amir Moradi, Thomas Peyrin,
584 Yu Sasaki, Pascal Sasdrich, and Siang Meng Sim. The SKINNY family of block ciphers and
585 its low-latency variant MANTIS. In *CRYPTO 2016 - 36th Annual International Cryptology
586 Conference.*, volume 9815 of *Lecture Notes in Computer Science*, pages 123–153. Springer,
587 2016.
- 588 5 Christof Beierle, Gregor Leander, Amir Moradi, and Shahram Rasoolzadeh. Craft: Lightweight
589 tweakable block cipher with efficient protection against dfa attacks. *IACR Transactions on
590 Symmetric Cryptology*, 2019(1):5–45, 2019.
- 591 6 E. Biham and A. Shamir. Differential cryptanalysis of feal and n-hash. In *EUROCRYPT*,
592 volume 547 of *LNCS*, pages 1–16. Springer, 1991.
- 593 7 A. Biryukov and I. Nikolic. Automatic search for related-key differential characteristics in
594 byte-oriented block ciphers: Application to AES, camellia, khazad and others. In *Advances in
595 Cryptology, LNCS 6110*, pages 322–344. Springer, 2010.
- 596 8 Andrey Bogdanov, Lars R Knudsen, Gregor Leander, Christof Paar, Axel Poschmann,
597 Matthew JB Robshaw, Yannick Seurin, and Charlotte Vikkelse. Present: An ultra-lightweight
598 block cipher. In *International workshop on cryptographic hardware and embedded systems*,
599 pages 450–466. Springer, 2007.
- 600 9 Geoffrey Chu and Peter J. Stuckey. Chuffed solver description, 2014. Available at http://www.minizinc.org/challenge2014/description_chuffed.txt.
601
- 602 10 Carlos Cid, Tao Huang, Thomas Peyrin, Yu Sasaki, and Ling Song. A security analysis of deoxys
603 and its internal tweakable block ciphers. *IACR Trans. Symmetric Cryptol.*, 2017(3):73–107,
604 2017.
- 605 11 Stéphanie Delaune, Patrick Derbez, Paul Huynh, Marine Minier, Victor Mollimard, and
606 Charles Prud’homme. SKINNY with scalpel - comparing tools for differential analysis. *IACR
607 Cryptol. ePrint Arch.*, 2020:1402, 2020.
- 608 12 FIPS 197. Advanced Encryption Standard. Federal Information Processing Standards Publi-
609 cation 197, 2001. U.S. Department of Commerce/N.I.S.T.
- 610 13 P. Fouque, J. Jean, and T. Peyrin. Structural evaluation of AES and chosen-key distinguisher
611 of 9-round AES-128. In *Advances in Cryptology - CRYPTO 2013 - Part I*, volume 8042 of
612 *LNCS*, pages 183–203. Springer, 2013.
- 613 14 D. Gérard. *Security Analysis of Contactless Communication Protocols*. PhD thesis, Université
614 Clermont Auvergne, 2018.
- 615 15 D. Gérard and P. Lafourcade. Related-key cryptanalysis of midori. In *INDOCRYPT*, volume
616 10095 of *LNCS*, pages 287–304, 2016.
- 617 16 D. Gerault, P. Lafourcade, M. Minier, and C. Solnon. Computing AES related-key differential
618 characteristics with constraint programming. *Artif. Intell.*, 278, 2020.
- 619 17 D. Gérard, M. Minier, and C. Solnon. Constraint programming models for chosen key
620 differential cryptanalysis. In *CP*, volume 9892 of *LNCS*, pages 584–601. Springer, 2016.
- 621 18 Hosein Hadipour, Sadegh Sadeghi, Majid M Niknam, Ling Song, and Nasour Bagheri. Com-
622 prehensive security analysis of craft. *IACR Transactions on Symmetric Cryptology*, pages
623 290–317, 2019.

- 624 **19** Jérémy Jean, Ivica Nikolic, Thomas Peyrin, and Yannick Seurin. Deoxys v1. 41. *Submitted to*
625 *CAESAR*, 2016.
- 626 **20** L. Knudsen. Truncated and higher order differentials. In *Fast Software Encryption*, pages
627 196–211. Springer, 1995.
- 628 **21** Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and
629 Guido Tack. Minizinc: Towards a standard CP modelling language. In *Principles and Practice*
630 *of Constraint Programming - CP 2007*, volume 4741 of *LNCS*, pages 529–543. Springer, 2007.
- 631 **22** Gurobi Optimization. Gurobi optimizer reference manual, 2018. URL: <http://www.gurobi.com>.
632
- 633 **23** L. Rouquette and C. Solnon. abstractXOR: A global constraint dedicated to differential
634 cryptanalysis. In *26th International Conference on Principles and Practice of Constraint*
635 *Programming*, volume 12333 of *LNCS*, pages 566–584, Louvain-la-Neuve, Belgium, September
636 2020. Springer.
- 637 **24** CE Shannon. Communication theory of secrecy systems, bell systems tech. *Bell System*
638 *Technical Journal*, 28:656–715, 1949.
- 639 **25** R. Singleton. Maximum distance q-nary codes. *IEEE Trans. Inf. Theor.*, 10(2):116–118,
640 September 2006. URL: <http://dx.doi.org/10.1109/TIT.1964.1053661>, doi:10.1109/TIT.
641 1964.1053661.
- 642 **26** Boxin Zhao, Xiaoyang Dong, and Keting Jia. New related-tweakey boomerang and rectangle
643 attacks on deoxys-bc including bdt effect. *IACR Transactions on Symmetric Cryptology*, pages
644 121–151, 2019.
- 645 **27** N.-F. Zhou, H. Kjellerstrand, and J. Fruhman. *Constraint Solving and Planning with Picat*.
646 Springer, 2015.