# Automatic Generation of Declarative Models for Differential Cryptanalysis


**Luc Libralesso**
LIMOS, CNRS UMR 6158, University Clermont Auvergne, France

**François Delobel**
LIMOS, CNRS UMR 6158, University Clermont Auvergne, France

**Pascal Lafourcade**
LIMOS, CNRS UMR 6158, University Clermont Auvergne, France

**Christine Solnon**
INSA Lyon, CITI, INRIA CHROMA, F-69621 Villeurbanne, France



## Abstract

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

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



**2012 ACM Subject Classification** Computing methodologies

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

**Digital Object Identifier** 10.4230/LIPIcs.CVIT.2016.23



## Acknowledgement

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




42nd Conference on Very Important Topics (CVIT 2016).
Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:18
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

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

### Notations

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

## 2 Unifying Description of Symmetric Block Ciphers

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

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

### 2.1 Ciphering Operators

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

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

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

More generally, there are two main categories of operators that ensure two main concepts identified by Shannon in [24]: Non-linear operators that ensure confusion, and linear operators that ensure diffusion. Non-linear operators are either S-Boxes (like the AES SubBytes) or non-linear arithmetic operations (like in ARX[1] structures). The most common linear operations used in symmetric ciphers are: multiplication by a MDS (Maximum Distance Separable) matrix (like the AES MixColumns), bit permutations, XOR and rotation (like the AES ShiftRows). Every linear operator $o$ satisfies the following property: $\forall t, t' \in [0,1]^{k*\#t_{in}(o)}, o(t) \oplus o(t') = o(t \oplus t')$.

## 2.2 Description of a Cipher with a DAG

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

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

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

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

# 3 Optimization and Enumeration of TDCs

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

## 3.1 Maximum Differential Characteristics

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

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

▶ **Definition 3** (Differential probability of an operator)**.** *The probability that an operator $o$ transforms an input difference $\delta t_{in}$ into an output difference $\delta t_{out}$ is*

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

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

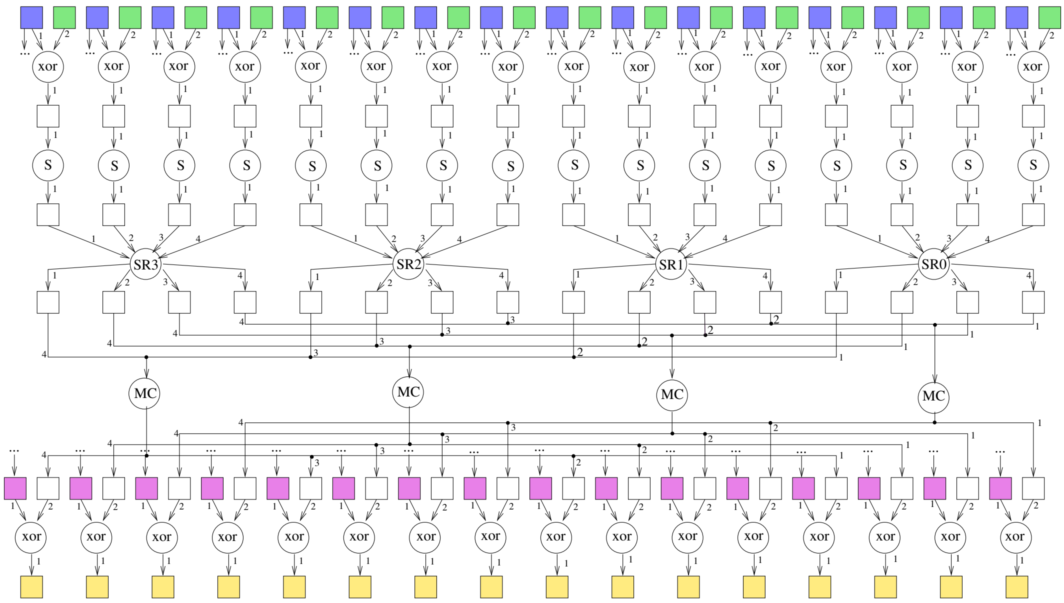


**Figure 1** DAG of the first round of the AES for 128-bit keys. Bytes are represented with squares, and operators with circles. The input key and plaintext have 128 bits and are split into 16 bytes colored in blue and green, respectively. Yellow squares correspond to the text state after one encryption 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).

This probability is equal to 0 or 1 for linear operators. More precisely, for any linear 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 comes from the fact that for any linear operator $o$ and any input parameters $t$ and $t'$, $o(t) \oplus o(t') = o(t \oplus t')$.

When an operator $o$ is not linear, $p_o$ may be different from 0 and 1 and the only case 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.

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

Let us now formally define what is an MDC.

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

*An MDC is a differential characteristic with maximum probability.*

## 3.2 Truncated Differential Characteristics

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

A TDC is a solution to an abstract problem. More precisely, the abstraction of a k-bit differential sequence $\delta x$ is a Boolean value denoted $\Delta X$ such that $\Delta X = 1$ iff $\delta x$ contains a difference, *i.e.*, $\delta x \neq 0$. Similarly, the abstraction of a differential tuple $\delta t = (\delta x_1, \ldots, \delta x_i)$ is the Boolean tuple $\Delta t = (\Delta x_1, \ldots, \Delta x_i)$ such that $\Delta x_j$ is the abstraction of $\delta x_j$ for each $j \in [1, i]$.

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

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

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

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

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

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

▶ **Example 10** ($S$). $S$ is not linear, and we cannot deterministically compute the output difference $\delta y_1$ given the input difference $\delta x_1$. However, as the look-up table is a bijection, we 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 $S$ operator.

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

As the probability $p_o(\delta t_{out} | \delta t_{in})$ associated with a non-linear operator $o$ is equal to 1 whenever $\delta t_{out} = \delta t_{in} = 0$ whereas it is very small otherwise (*e.g.*, smaller than or equal to $2^{-6}$ for the AES Sbox), we can compute an upper bound on an MDC by computing a lower bound on the number of *active* non-linear operators in a TDC, where an operator is said to be active whenever its input/output differential tuples are different from 0. More precisely, let $s(\Delta)$ be the number of active non-linear operators in a TDC $\Delta$ (*i.e.*, $s(\Delta) = \#\{o \in O : o \text{ is not linear} \wedge \delta pred(o) \neq 0\}$), and let $s^*$ be the minimal value of $s(\Delta)$ for all possible TDCs $\Delta$. If the maximal probability of an active non-linear operator is equal to $p$, then the probability of an MDC is upper bounded by $p^{s^*}$. For example, for the AES this upper bound is $2^{-6 \cdot s^*}$. In some cases, this upper bound is small enough to ensure the security of the cipher with respect to differential attacks, and it is not necessary to actually compute MDCs. Most papers that introduce new ciphering algorithms demonstrate the security of their cipher with respect to differential attacks only by computing this upper bound (*e.g.*, [5]). When the upper bound $p^{s^*}$ is large enough to allow mounting differential attacks, we have to enumerate all possible TDCs that have a given number of active non-linear operators, and we have to search for an MDC for each of these TDCs.

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

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

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

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

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

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

## 4 Description of a Symmetric Block Cipher with Tagada

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

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

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

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

Some patterns may be repeated in the DAG. For example, let us consider the DAG describing the first round of the AES displayed in Fig. 1. At the top level of this DAG, there are 16 XORs which correspond to the *AddRoundKey (ARK)* step, where each byte of the text (in blue) is XORed with the corresponding byte of the key (in green). As it is tedious to write 16 times the JSON representation of one XOR operation, TAGADA provides functions corresponding to meta-operators, where a meta-operator is a classical combination of operators.

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

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

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

## 5 Automatic Generation of Models with Tagada

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

### 5.1 Automatic Generation of Constraints

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

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

▶ **Definition 13** (Relation $\mathcal{R}_o$ associated with an operator $o$)**.** $\mathcal{R}_o = \{(\Delta(x_1), \ldots, \Delta(x_{\#t_{in}(o)}),\ \Delta(y_1), \ldots, \Delta(y_{\#t_{out}(o)})) : \exists (x_1, \ldots, x_{\#t_{in}(o)}) \in [0,1]^{k*\#t_{in}(o)},$ $(y_1, \ldots, y_{t_{out}(o)}) = o(x_1, \ldots, x_{\#t_{in}(o)})\}$ *where* $\forall x \in [0,1]^k$, $\Delta(x)$ *denotes the Boolean abstraction of* $x$, i.e., $\Delta(x) = 0 \Leftrightarrow x = 0$.

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

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

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

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

## 5.2 Simplification of the DAG

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

### Rule 1: Merging Equal Parameters

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

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

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

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

## Rule 2: Suppressing Constant Parameters

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

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

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

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

## Rule 3: Suppressing Free Parameters

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

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

More formally, Rule 3 is defined as follows.

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

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

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

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

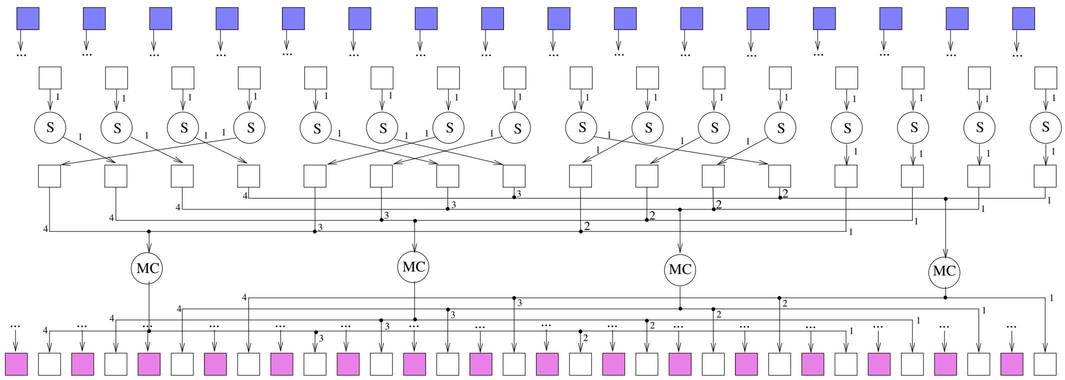

**Figure 2** Shaved DAG obtained from the DAG of Fig. 1 after applying Rules 1, 2, and 3.

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

## 5.3 Extension of the DAG

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

### 5.3.1 Generation of New Vertices and Edges from Existing Operators

In [17, 16, 23], Step1-opt and Step1-enum models are tightened by exploiting the fact that, 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 $t_1 \oplus t_3 = MC(t_2 \oplus t_4)$. As a consequence, the MDS property also holds on $t_1 \oplus t_3$ and $t_2 \oplus t_4$, *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 equal to 0 or strictly greater than 4. Hence, a new variable (called *diff* variable in [16]) is added for each parameter of each couple of $MC$ operators. These *diff* variables are related with initial parameters by adding XOR constraints. Finally, constraints that ensure the MDS property are added for these new *diff* variables.

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

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

- For each $j \in [1, \#t_{out}(u)]$, we add a new parameter vertex $y_{3,j}$ corresponding to the result of XORing $y_{1,j}$ and $y_{2,j}$, *i.e.*, we add a new XOR vertex whose predecessors are $x_{1,i}$ and $x_{2,i}$ and whose successor is $x_{3,i}$;
- We add a new operator vertex $o_3$ such that the UID of $o_3$ is $u$, the predecessors of $o_3$ are $x_{3,1}, \ldots, x_{3,\#t_{in}(u)}$, and the successors of $o_3$ are $y_{3,1}, \ldots, y_{3,\#t_{out}(u)}$.

This may be done for each kind of linear operator except XOR (as this is useless in this case).

As this step may drastically increase the size of the DAG, it is optional, and the user can choose the kind of linear operator that should be considered for this step.

### 5.3.2 Generation of New XORs

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

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

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

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

## 5.4 Generation of the MiniZinc Model from the DAG

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

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

| model | Midori (35) #d | #o | #e | AES (25) #d | #o | #e | SKINNY (56) #d | #o | #e | CRAFT (38) #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.

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

# 6 Experimental Results

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

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

TAGADA has a parameter $n_{max}$ that is used to control the maximum size of new generated XOR equations (see Section 5.3.2). It is also possible to select the linear operators for which we infer new vertices and edges as explained in Section 5.3.1. In the four considered ciphers, the only linear operator that can be selected is $MC$ as $SR$ is removed during the DAG 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`

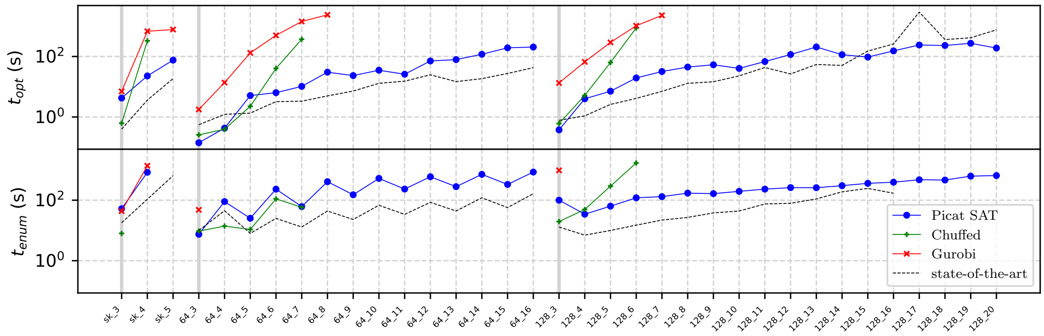


**Figure 3** CPU time of Picat-SAT, Chuffed and Gurobi on the model generated by TAGADA for Midori instances when $n_{max} = 4$ and $MC$ is selected (top plot for Step1-opt and bottom plot for Step1-enum). State-of-the art is the handcrafted model of [14] run with Picat-SAT.

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

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

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

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

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

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

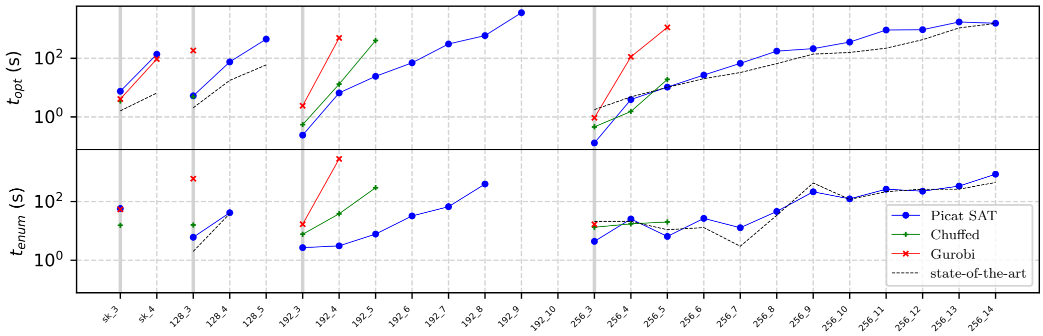


**Figure 4** CPU time of Picat-SAT, Chuffed and Gurobi on the model generated by TAGADA for AES instances when $n_{max} = 5$ and $MC$ is selected (top plot for Step1-opt and bottom plot for Step1-enum). State-of-the art is the handcrafted model of [16] run with Picat-SAT.

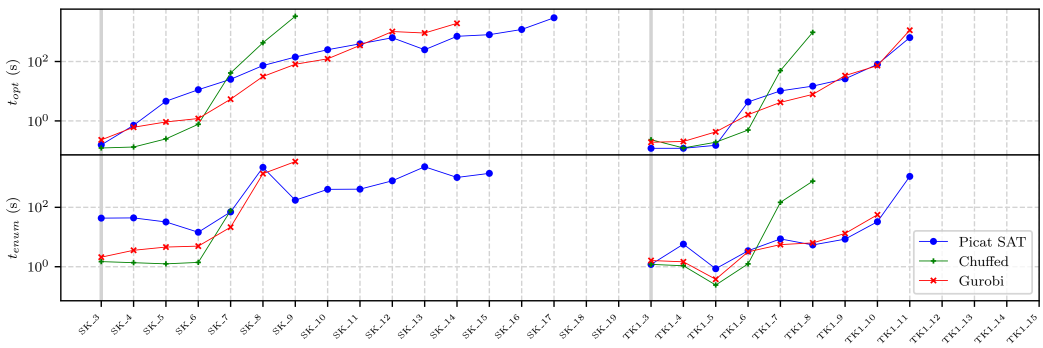


**Figure 5** CPU time of Picat-SAT, Chuffed and Gurobi on the model generated by TAGADA for Skinny when $n_{max} = 0$ and $MC$ is not selected (top plot for Step1-opt and bottom plot for Step1-enum).

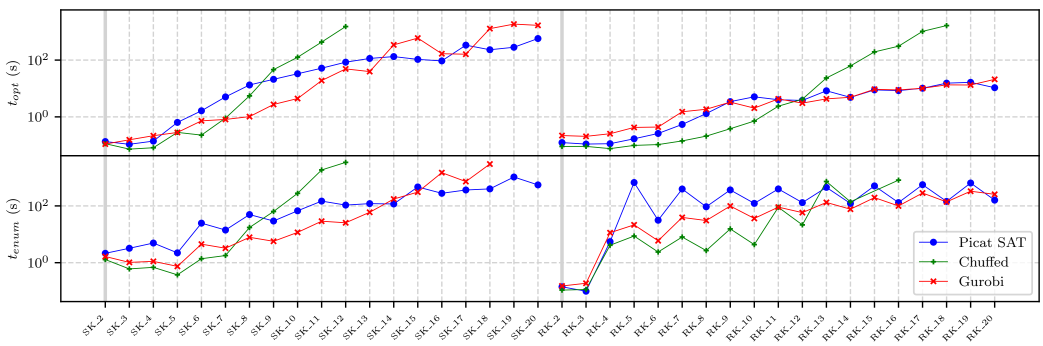


**Figure 6** CPU time of Picat-SAT, Chuffed and Gurobi on the model generated by TAGADA for Craft when $n_{max} = 0$ and $MC$ is not selected (top plot for Step1-opt and bottom plot for Step1-enum).

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

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

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

## 7 Conclusion

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

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

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

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

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

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

## References


1 S. Banik, A. Bogdanov, T. Isobe, K. Shibutani, H. Hiwatari, T. Akishita, and F. Regazzoni. Midori: A block cipher for low energy. In *ASIACRYPT*, volume 9453 of *LNCS*, pages 411–436. Springer, 2015.

2 Subhadeep Banik, Sumit Kumar Pandey, Thomas Peyrin, Yu Sasaki, Siang Meng Sim, and Yosuke Todo. Gift: a small present. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 321–345. Springer, 2017.

3 Ray Beaulieu, Stefan Treatman-Clark, Douglas Shors, Bryan Weeks, Jason Smith, and Louis Wingers. The simon and speck lightweight block ciphers. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2015.

4 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 *CRYPTO 2016 - 36th Annual International Cryptology Conference,*, volume 9815 of *Lecture Notes in Computer Science*, pages 123–153. Springer, 2016.

5 Christof Beierle, Gregor Leander, Amir Moradi, and Shahram Rasoolzadeh. Craft: Lightweight tweakable block cipher with efficient protection against dfa attacks. *IACR Transactions on Symmetric Cryptology*, 2019(1):5–45, 2019.

6 E. Biham and A. Shamir. Differential cryptoanalysis of feal and n-hash. In *EUROCRYPT*, volume 547 of *LNCS*, pages 1–16. Springer, 1991.

7 A. Biryukov and I. Nikolic. Automatic search for related-key differential characteristics in byte-oriented block ciphers: Application to AES, camellia, khazad and others. In *Advances in Cryptology, LNCS 6110*, pages 322–344. Springer, 2010.

8 Andrey Bogdanov, Lars R Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew JB Robshaw, Yannick Seurin, and Charlotte Vikkelsoe. Present: An ultra-lightweight block cipher. In *International workshop on cryptographic hardware and embedded systems*, pages 450–466. Springer, 2007.

9 Geoffrey Chu and Peter J. Stuckey. Chuffed solver description, 2014. Available at `http://www.minizinc.org/challenge2014/description_chuffed.txt`.

10 Carlos Cid, Tao Huang, Thomas Peyrin, Yu Sasaki, and Ling Song. A security analysis of deoxys and its internal tweakable block ciphers. *IACR Trans. Symmetric Cryptol.*, 2017(3):73–107, 2017.

11 Stéphanie Delaune, Patrick Derbez, Paul Huynh, Marine Minier, Victor Mollimard, and Charles Prud'homme. SKINNY with scalpel - comparing tools for differential analysis. *IACR Cryptol. ePrint Arch.*, 2020:1402, 2020.

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

13 P. Fouque, J. Jean, and T. 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.

14 D. Gérault. *Security Analysis of Contactless Communication Protocols.* PhD thesis, Université Clermont Auvergne, 2018.

15 D. Gérault and P. Lafourcade. Related-key cryptanalysis of midori. In *INDOCRYPT*, volume 10095 of *LNCS*, pages 287–304, 2016.

16 D. Gerault, P. Lafourcade, M. Minier, and C. Solnon. Computing AES related-key differential characteristics with constraint programming. *Artif. Intell.*, 278, 2020.

17 D. Gérault, M. Minier, and C. Solnon. Constraint programming models for chosen key differential cryptanalysis. In *CP*, volume 9892 of *LNCS*, pages 584–601. Springer, 2016.

18 Hosein Hadipour, Sadegh Sadeghi, Majid M Niknam, Ling Song, and Nasour Bagheri. Comprehensive security analysis of craft. *IACR Transactions on Symmetric Cryptology*, pages 290–317, 2019.

**19** Jérémy Jean, Ivica Nikolic, Thomas Peyrin, and Yannick Seurin. Deoxys v1. 41. *Submitted to CAESAR*, 2016.

**20** L. Knudsen. Truncated and higher order differentials. In *Fast Software Encryption*, pages 196–211. Springer, 1995.

**21** 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.

**22** Gurobi Optimization. Gurobi optimizer reference manual, 2018. URL: `http://www.gurobi.com`.

**23** L. Rouquette and C. Solnon. abstractXOR: A global constraint dedicated to differential cryptanalysis. In *26th International Conference on Principles and Practice of Constraint Programming*, volume 12333 of *LNCS*, pages 566–584, Louvain-la-Neuve, Belgium, September 2020. Springer.

**24** CE Shannon. Communication theory of secrecy systems, bell systems tech. *Bell System Technical Journal*, 28:656–715, 1949.

**25** R. Singleton. Maximum distance q-nary codes. *IEEE Trans. Inf. Theor.*, 10(2):116–118, September 2006. URL: `http://dx.doi.org/10.1109/TIT.1964.1053661`, `doi:10.1109/TIT.1964.1053661`.

**26** Boxin Zhao, Xiaoyang Dong, and Keting Jia. New related-tweakey boomerang and rectangle attacks on deoxys-bc including bdt effect. *IACR Transactions on Symmetric Cryptology*, pages 121–151, 2019.

**27** N.-F. Zhou, H. Kjellerstrand, and J. Fruhman. *Constraint Solving and Planning with Picat*. Springer, 2015.