A Simpler Model for Recovering Superpoly on Trivium

Stéphanie Delaune, Patrick Derbez, Arthur Gontier, Charles Prud'Homme

▶ **To cite this version:**

Stéphanie Delaune, Patrick Derbez, Arthur Gontier, Charles Prud'Homme. A Simpler Model for Recovering Superpoly on Trivium. SAC 2021: International Conference on Selected Areas in Cryptography, Sep 2021, Victoria, British-Columbia, Canada. pp.266-285, 10.1007/978-3-030-99277-4_13 . hal-03534492

HAL Id: hal-03534492
https://hal.science/hal-03534492

Submitted on 19 Jan 2022

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.

A Simpler Model for Recovering Superpoly on Trivium

Stéphanie Delaune[1], Patrick Derbez[1], Arthur Gontier[1], and Charles Prud'homme[2]

[1] Univ Rennes, CNRS, IRISA, Rennes, France
{stephanie.delaune,patrick.derbez,arthur.gontier}@irisa.fr
[2] TASC, IMT-Atlantique, LS2N-CNRS, F-44307 Nantes
charles.prudhomme@imt-atlantique.fr

Abstract. The cube attack is a powerful cryptanalysis technique against symmetric primitives, especially for stream ciphers. One of the key step in a cube attack is recovering the superpoly. The division property has been introduced to cube attacks with the aim first to identify variables/monomials that are *not* involved in the superpoly. Recently, some improved versions of this technique allowing the recovery of the exact superpoly have been developed and applied on various stream ciphers [13,15].

In this paper, we propose a new model to recover the exact superpoly of a stream cipher given a cube. We model the polynomials involved in the stream cipher as a directed graph. It happens that this structure handles some of the monomial cancellations more easily than those based on division property, and this leads to better timing results. We propose two implementations of our model, one in MILP and one in CP, which are up to 10 times faster than the original division property-based model from Hao *et al.* [13], and consistently 30 to 60 times faster than the monomial prediction-based model from Hu *et al.* [15].

Keywords: Stream cipher · Cube Attack · Division Property · Trivium · MILP · CP

1 Introduction

Generic solvers as Gurobi [12] or Choco [17] are nowadays very common tools for cryptographers. They allow cryptographers to describe the problems they want to solve with high-level codes, making implementation faster and results much easier to verify. This approach has been very successful so far as generic solvers were used to find a large variety of attacks and distinguishers. On the one hand, Gurobi was used to search for integral distinguishers based on division property [23], differential characteristics [1], advanced meet-in-the-middle attacks [19] and cube attacks [13] by solving Mixed-Integer Linear Programming (MILP) models. On the other hand, Constraint Programming (CP) solvers as Choco were mainly used for highly non-linear problems as instantiating truncated differential characteristics [11,5]. Furthermore, some works combine both

approaches as in [6] where a MILP model is first used to search for best truncated boomerang characteristics followed by a call to Choco to find the best instantiations and clusters.

Given a cryptographic problem, there are most often many ways to model it in MILP or CP and determining the best one is a hard task. It is commonly assumed that smaller the model is, faster it can be solved and many works were dedicated to decrease the number of constraints required to describe cryptographic problems. For instance, Abdelkhalek *et al.* used both the Quine-McCluskey and Espresso algorithms to reduce the number of inequalities required to model the Difference Distribution Table (DDT) of 8-bit Sboxes [1]. Similarly, Boura *et al.* proposed new techniques to represent any subset of $\{0,1\}^n$ with a small number of inequalities [3]. However this approach has its limits and sometimes adding redundant constraints may improve the running times. For instance, Delaune *et al.* found that adding the constraints related to minimal number of active Sboxes into their model dedicated to boomerang characteristics greatly improves it [6].

Another approach used to decrease the overall running time consists in dividing the problem into two (or more) subproblems much easier to solve. In [25], Zhou *et al.* showed that to search for best truncated differential characteristics against Substitution-Permutation Network (SPN) it was much faster to split the whole model according to the number of active Sboxes on some of the internal states. The intuition is that best truncated differential characteristics will have several internal states with very few active Sboxes. Similarly, in [13], Hao *et al.* proposed MILP models to recover the superpolies of several stream ciphers using division property. To speed-up the resolution and reach reasonable running times they had to compute all the possible monomials at the half of the initialisation process (from the output bit) and then solve the problem for each of them.

Our contributions. In this paper, we propose a new model to recover the superpoly of a stream cipher given a cube. We illustrate it on Trivium but the approach is generic and could be applied to other stream ciphers. Our idea is to model the polynomials involved in the initialisation process as a directed graph for which nodes are the state variables and edges the monomials. The main advantage of our model regarding the one of Hao *et al.* [13] based on division property is that it is much easier to handle some of the monomial cancellations and thus much faster to solve large instances. We then propose two implementations of this model, one in MILP and one in CP, which are up to 10 times faster than the original division property-based model from Hao *et al.* [13]. We also compare our approach to the *monomial prediction* technique introduced by Hu *et al.* in [15] and found our models are consistently 30 to 60 times faster to solve. Finally, regarding the MILP model, we show how to improve the strategy deployed by Gurobi without relying on a divide-and-conquer strategy.

2 Some background

2.1 Cube attacks

Cube attacks, introduced by Dinur and Shamir at EUROCRYPT in 2009 [7], has become a general tool for evaluating the security of cryptographic primitives, and has been successfully applied against various stream ciphers, e.g. [2,7,9].

Roughly, the output bit of a cipher is seen as an unknown Boolean polynomial $f(\boldsymbol{k}, \boldsymbol{v})$ where $\boldsymbol{k}$ is a vector of secret input variables, and $\boldsymbol{v}$ is a vector of public input variables. Given a monomial t_I which is the product of public variables in $I = \{i_1, \ldots, i_d\}$, i.e. $t_I = v_{i_1} \cdot \ldots \cdot v_{i_d}$, the function f can be represented as:

$$f(\boldsymbol{k}, \boldsymbol{v}) = t_I \cdot p_I + q(\boldsymbol{k}, \boldsymbol{v})$$

where the polynomial $q(\boldsymbol{k}, \boldsymbol{v})$ only contains terms which are not supersets of I. The polynomial p_I is called the *superpoly* of I in f. The set $I = \{i_1, \ldots, i_d\}$ determines a specific structure called *cube*, denoted as C_I, containing 2^d values where variables in $\{v_{i_1}, \ldots, v_{i_d}\}$ take all possible combinations of values.

Then, the main idea behind a cube attack (and its variants) is the fact that the sum of the polynomials $f(\boldsymbol{k}, \boldsymbol{v})$ considering all the possible values for the cube, and assuming that the other ones are fixed, is exactly the superpoly p_I.

$$\bigoplus_{C_I} f(\boldsymbol{k}, \boldsymbol{v}) = \bigoplus_{C_I} \big(t_I \cdot p_I(\boldsymbol{k}, \boldsymbol{v}) + q(\boldsymbol{k}, \boldsymbol{v})\big) = p_I(\boldsymbol{k}, \boldsymbol{v})$$

Thus, being able to determine the superpoly allows either to distinguish the cipher from a random function (if $p_I = 0$) or to retrieve some information on the key. The goal of the adversary is then to find the best trade-off between the size of the cube, the number of key bits involved in the superpoly and how far is the superpoly from a balanced function.

The mainly used methods to retrieve the superpoly are currently all based on the division property without unknown subsets.

2.2 Division property

The division property is a generic technique to search for integral distinguishers. It was originally proposed by Todo *et al.* at Eurocrypt 2015 [20] and has been widely applied to many ciphers, e.g. [22]. In [21], division property was used to search for cube attacks against stream ciphers but authors made two assumptions on superpolies which turned out to be wrong [24]. Thus, at Eurocrypt 2020, Hao *et al.* introduced the *exact* division property and showed the technique allows one to fully recover the superpoly of several round-reduced stream ciphers [13]. Basically, this technique allows one to track a monomial through the successive applications of a round function relying on the notion of trails.

We illustrate this on a simple example.

Example 1. Consider the functions f and g defined as follows:

$$\begin{aligned}(y_1, y_2) &= f(x_1, x_2, x_3) = (x_1 + x_3,\ x_1x_2 + x_1)\\ (z_1, z_2) &= \quad g(y_1, y_2) \quad = (y_1y_2,\ y_1 + y_2)\end{aligned}$$

Assume that our cipher is given by $g \circ f$. We can compute the ANF (Algebraic Normal Form) of the entire cipher to determine the monomials that compose this function but performing this computation is not possible on real ciphers. Here, we have that:

$$(z_1, z_2) = (g \circ f)(x_1, x_2, x_3) = (x_1 + x_1x_2 + x_1x_3 + x_1x_2x_3,\ x_3 + x_1x_2)$$

Instead of computing the ANF, we may represent the propagation of the different monomials through a table. The tables below represent respectively the behaviours of the functions f and g.

x_1	x_2	x_3	y_1	y_2
0	0	0	0	0
1	0	0	1	0
0	0	1	1	0
1	0	0	0	1
1	1	0	0	1
1	1	0	1	1
1	0	0	1	1
1	1	1	1	1
1	0	1	1	1

y_1	y_2	z_1	z_2
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0
0	0	1	1

For instance, the 4 last lines of the first table represent the fact x_1x_2, x_1, $x_1x_2x_3$, and x_1x_3 are monomials occurring in y_1y_2. Consulting this first table, we can also see that the monomial x_1 is present in y_1 (2^{nd} line), in y_2 (4^{th} line) and also in y_1y_2 (7^{th} line) .

Now, if we want to study whether the monomial x_1 occurs for instance in the first component of the ANF of the function $g \circ f$, we have to look for the existence of some trails starting from $(1, 0, 0)$ which represents x_1 to $(z_1, z_2) = (1, 0)$. Reading the tables, we have that: $(1, 0, 0) \to (1, 0)$, $(1, 0, 0) \to (0, 1)$ and $(1, 0, 0) \to (1, 1)$ in the table representing f. Each of these trails can be completed with a line from the table representing g, leading to the following trails:

$$\begin{aligned}&(1, 0, 0) \to (1, 0) \to (0, 1)\\ &(1, 0, 0) \to (0, 1) \to (0, 1)\\ &(1, 0, 0) \to (1, 1) \to (1, 0)\end{aligned}$$

The last line represents the fact that the monomial x_1 occurs in y_1y_2, and thus in z_1. Regarding, the two first lines, we can see that we have two trails starting from $(1, 0, 0)$ and ending with $(0, 1)$. The first one indicates that the monomial x_1 occurs in y_1, and thus in z_2 (since y_1 occurs in z_2). The second one indicates that the monomial x_1 occurs in y_2, and thus in z_2 (since y_2 occurs also in z_2). These two occurrences cancel each other, and thus at the end, x_1 does not occur

in z_2. Therefore to accurately decide whether a given monomial occurs in the ANF, it is important to count the number of trails: an even number means that the monomial does not occur, whereas an odd number means that it is indeed present.

Relying on this technique, we are therefore looking for trails starting with a given vector representing a particular monomial and leading to a specific vector indicating the presence of the monomial in the ANF after a number of iterations of the round function. The difficulty of the search procedure depends on many parameters as the round function, the state size, the number of rounds. However, for real ciphers computing the whole propagation table of the round function may be infeasible or requires too many inequalities to be described. Hence, in most of the cases, propagation rules are added for each basic operator of the cipher (xor, and, copy).

Several algorithms have been developed to evaluate the propagation of the division property on ciphers. Some are based on the so-called breadth-first search algorithm [20,22] whereas some others implement this search using the mixed integer linear programming (MILP) method [13].

The downside of all these approaches is that we can hardly add new properties to strengthen the model as a global view on the problem we are trying to solve is missing.

3 A graph-based model for superpoly recovery

In this section, we present a novel and simple graph-based model dedicated to recovering the superpoly of a stream cipher for a given cube. It is fully equivalent to the last version of division property since it can recover the exact superpoly, but has the main advantage of being much easier to understand and manipulate. We represent all the intermediate variables and monomials using a directed graph G. A node of G represents a variable and an edge from x to y indicates that y appears in the ANF of x. The possible transitions from a node to its child nodes are described by the monomials of the round function. This results in an automaton which defines the set of outgoing edge of each node of G.

Example 2. Let get back to Example 1. We have the equations:

$$\begin{aligned} y_1 &= x_1 \oplus x_3 & \qquad z_1 &= y_1 y_2 \\ y_2 &= x_1 x_2 \oplus x_1 & \qquad z_2 &= y_1 \oplus y_2 \end{aligned}$$

This system of equations can be expressed as a graph $G = (V, E)$ where :

- $V = \{x_1, x_2, x_3, y_1, y_2, z_1, z_2\}$ is the set of nodes,
- $E = \{(y_1, x_1), (y_1, x_3), (y_2, x_1), (y_2, x_2), (z_1, y_1), (z_1, y_2), (z_2, y_1), (z_2, y_2)\}$ is the set of edges.

We call *trail* any collection of edges $\mathcal{T}$ representing a monomial together with the process to obtain it by developing the polynomial expressions of the

root nodes. For the above system it means that $\mathcal{T}$ has to satisfy the following extra constraints:

- $(y_1, x_1) \in \mathcal{T} \implies (y_1, x_3) \notin \mathcal{T}$
- $(y_1, x_3) \in \mathcal{T} \implies (y_1, x_1) \notin \mathcal{T}$
- $(y_2, x_2) \in \mathcal{T} \implies (y_2, x_1) \in \mathcal{T}$
- $(z_1, y_1) \in \mathcal{T} \iff (z_1, y_2) \in \mathcal{T}$
- $(z_2, y_1) \in \mathcal{T} \implies (z_2, y_2) \notin \mathcal{T}$
- $(z_2, y_2) \in \mathcal{T} \implies (z_2, y_1) \notin \mathcal{T}$

It is interesting to compare our model to the model based on division property with basic propagation rules. In that case the system would be rewritten as:

$$\begin{aligned}(x_{11}, x_{12}, x_{13}) &= \text{copy}(x_1)\\ y_1 &= \text{xor}(x_{11}, x_3)\\ a &= \text{and}(x_{12}, x_2)\\ y_2 &= \text{xor}(a, x_{13})\\ (y_{11}, y_{12}) &= \text{copy}(y_1)\\ (y_{21}, y_{22}) &= \text{copy}(y_2)\\ z_1 &= \text{and}(y_{11}, y_{21})\\ z_2 &= \text{xor}(y_{12}, y_{22})\end{aligned}$$

It now contains 15 variables, 4 copy-constraints, 3 xor-constraints and 2 and-constraints, which seems much more complex than our model with 8 variables (the edges) and 6 constraints. However we have to add extra constraints to our model to ensure that if an edge $(\cdot, a)$ belongs to $\mathcal{T}$ then either a is a leaf or an edge $(a, \cdot)$ also belongs to $\mathcal{T}$. For complex polynomials this can be tricky and force us to use the same intermediate variables than above to simplify the constraints.

Interestingly, and most importantly, both models have exactly the same solutions. This means that there is a one-to-one mapping between the possible trails $\mathcal{T}$ and the possible solutions of the division property-based model. Hence, the main advantage of our model relies on the ease of adding extra constraints to remove false (even) trails and deploying branch-and-cut strategies.

Example on Trivium

Trivium [4] is an NFSR-based stream cipher. Its internal state is represented by a 288-bit state $(s_1, s_2, \ldots, s_{288})$ distributed on three registers A, B, and C. The 80-bit secret key K is loaded to register A, and the 80-bit initialisation vector IV is loaded to register B. The other state bits are set to 0 except the last three bits in register C. Namely, the initial state bits are represented as:

$$\begin{aligned}s_1, \ldots, s_{80}, s_{81}, \ldots, s_{93} &\leftarrow K[1], \ldots, K[80], 0, \ldots, 0\\ s_{94}, \ldots, s_{174}, s_{175}, s_{176}, s_{177} &\leftarrow IV[1], \ldots, IV[80], 0, 0, 0, 0\\ s_{178}, \ldots, s_{285}, s_{286}, s_{287}, s_{288} &\leftarrow 0, \ldots, 0, 1, 1, 1\end{aligned}$$

At each round, we first compute t_1, t_2, and t_3 as:

$$\begin{aligned}t_1 &\leftarrow s_{66} + s_{91}s_{92} + s_{93} + s_{171}\\ t_2 &\leftarrow s_{162} + s_{175}s_{176} + s_{177} + s_{264}\\ t_3 &\leftarrow s_{243} + s_{286}s_{287} + s_{288} + s_{69}\end{aligned}$$

Then, the three registers are updated as follows:

$$\mathsf{A} \leftarrow t_3, s_1, \ldots, s_{92} \qquad \mathsf{B} \leftarrow t_1, s_{94}, \ldots, s_{176} \qquad \mathsf{C} \leftarrow t_2, s_{178}, \ldots, s_{287}$$

The state is updated 1152 times and then, at each new round, an output bit is produced: $z \leftarrow s_{66} + s_{93} + s_{162} + s_{177} + s_{243} + s_{288}$. Figure 1 depicts graphically transitions of the TRIVIUM stream cipher.

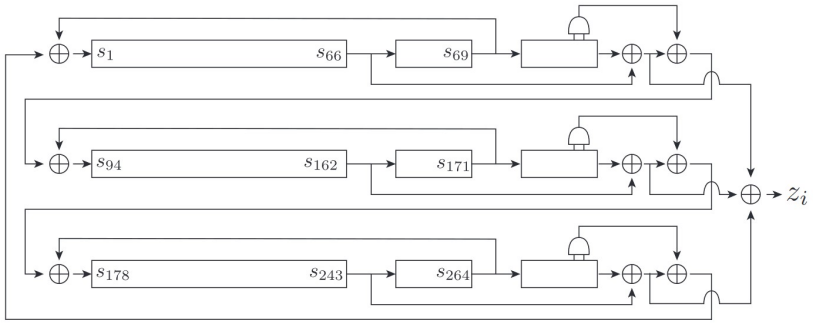

Fig. 1. The Trivium cipher.

Fig. 2 depicts the Deterministic Finite Automaton (DFA) deduced from the description of TRIVIUM. Note that the DFA develops TRIVIUM backwards: from the output bit to first round. There are four possible transitions to go from one register (A, B or C) to its successors: three of them are simple, and one is doubling ($\Longrightarrow$). In the following, the three simple transitions will be named the looping ($\dashrightarrow$), the short ($\dashrightarrow$), and the long ($\longrightarrow$) one.

One may have noticed that the DFA relies only on the first bit of each register. Indeed, for the other bits, the application of the round function simply consists of shifting them to the left. The value of the shifted bits only changes when they turn back to the first position of a register. Moreover, none of these shifted bits are a result of a round function, so they have no use in the DFA. In summary, the DFA already simplifies all the shifted bits at each round on each register to focus on the one produced in the corresponding round function (t_1,t_2, and t_3). The node A (resp. B, C) represents the first bit of register A (resp. B, C). Whenever a transition is taken, the generated bit will have to be shifted k times to be on the first position again. For example, if the bit at the first position of register A is set to 1 at round R, then it can be propagated to register C or A. If A is selected, then the first bit of A will be activated at round $R - 69$, that is $k = 69$. Otherwise, if C is selected, there are two scenarios. Either a simple transition is taken (short or long), which corresponds to the activation of the the first bit of C at either round $R - 66$ or $R - 111$. Or the doubling transition is picked and the first bit of the register C will be activated at round $R - 110$ and $R - 109$.

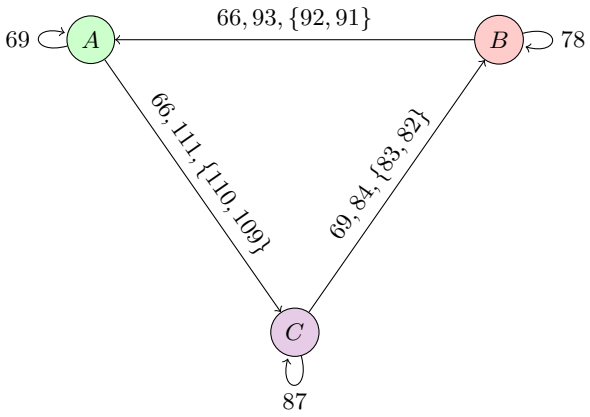

Fig. 2. A DFA that encodes possible transitions for the Trivium cipher.

Now we will present the way to build a graph modelling a division trail based on the DFA (Fig. 2), using a breadth-first search algorithm. The node corresponding to the output bit z at round R is created first and pushed into a queue. This triggers a loop that ends when the queue is empty. A node is popped from the queue and marked as visited. If the node has a positive R value, its child nodes described by the DFA are lazily created, added to the queue and used edges are created. Creating a node requires to know its round R: anytime a transition is visited, the number of shifts that label it is subtracted from the R value of its parent node. If a popped node has a negative R value, which corresponds to the first state of the cipher, no action is performed. When the loop stops, the graph of all possibilities is declared.

Such a graph is not very deep but it is very wide since any node has potentially five child nodes. However, each node in a solution has only one or two outgoing edges. Fig. 3 shows a graph solution for Trivium 672 with the starter node s_{243}, i.e. the 66^{th} bit of register C. Therefore, the source node is labelled by C with $R = 672 - 66 = 606$ since the bit at position 66 in register C has to be shifted 66 times to be on the first position. The blue nodes are the cube bits and the red ones are the key bits. Double-line edges ($\Longrightarrow$) stand for doubling transitions, plain-line edges ($\longrightarrow$) for long transitions, dashed-line edges ($\dashrightarrow$) for short transition, and dotted-line edges ($\cdots\!\!\rightarrow$) for looping transitions. This solution represents one trail for the superpoly monomial x_{16}.

Once we have formalised our problem as a graph problem, we can rely on a MILP solver (e.g. Gurobi) or a CP solver (e.g. Choco) to enumerate all the solutions.

In the following we chose Gurobi [12] as the solver for the MILP model because it already showcased its efficiency on division property and Choco [17] as a constraint programming (CP) solver for comparison but also because it natively supports constraints over graph variables [8].

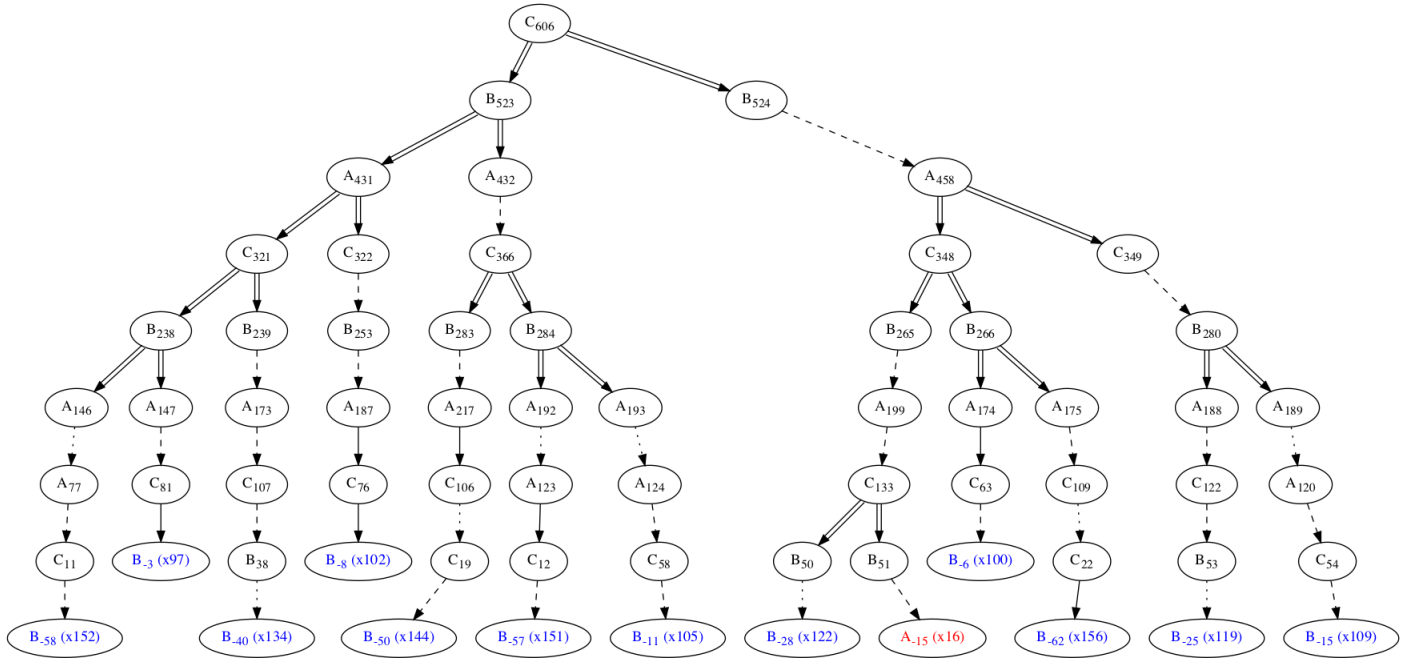

Fig. 3. A solution for TRIVIUM 672 considering s_{243} (66^{th} bit of register C) as starter node. Blue nodes are the cube bits; the red one is the key bit. Double-line edges are for doubling transitions, plain-line edges for long transitions, dashed-line edges for short transition and dotted-line edges for looping transitions.

4 Strengthening the graph-based model

The graph-based model is equivalent to the more classical model based on trails, but its global structure eases new constraints like the following ones.

4.1 Constrain the doubling paths

To retrieve the superpoly we need to enumerate all the trails and count how many trails there are for one given monomial. Indeed, a monomial with an even number of trails will cancel itself in the superpoly.

In the graph-based representation, a doubling path is a pair of distinct sub-graphs connecting a given set of nodes to another given set of nodes (the leaves). If a trail uses one sub-graph of a doubling path, then the trail using the other sub-graph will produce the same monomial. Therefore, preventing doubling paths to exist in the graph will reduce the number of trails that cancel each other out. We identified several doubling paths for Trivium.

Pattern 1 (long-double) Between each register there is the long transition with a given number of shifts p, and the doubling transition with $p-1$ and $p-2$ shifts. Therefore, if the doubling edge is followed by two long edges, we will get the same leaves than taking the long edge first and the doubling edge after, as depicted on Fig. 4.

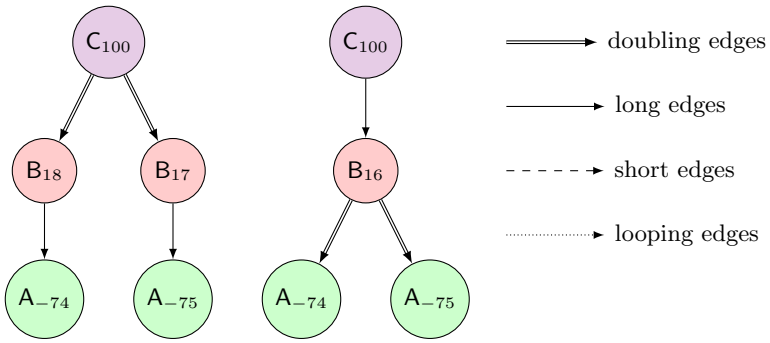

Fig. 4. long-double pattern

To discard this pattern, we have to take care that the intermediate nodes, here B_{16}, B_{17}, and B_{18}, are not used in any other part of the trail.

Pattern 2 (3 consecutive bits) An other doubling pattern is when three bits are at consecutive rounds on the same register as depicted on Fig. 5. For example

on the nodes C_{97}, C_{98}, and C_{99}. If the doubling edges are taken on C_{97} and C_{99}, the long and the doubling edges of the middle bit can then be removed because these two choices lead to the same output nodes (B_{14},B_{15},B_{16},B_{17}).

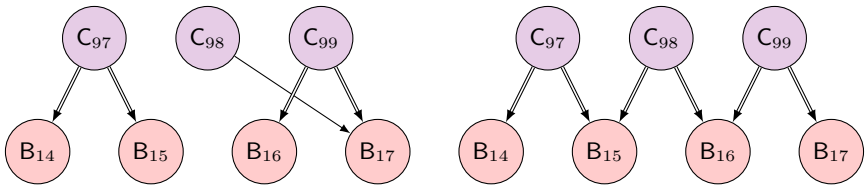

Fig. 5. 3 consecutive bits pattern

Pattern 3 (looping) When a looping transition is taken *i.e.,* the bit stays on the same register, and if all the outgoing edges of the looping register return to the same register at least once in the trail, then a similar result can be obtained by not taking the first looping transition but taking it on each outgoing edges as shown on Fig.6.

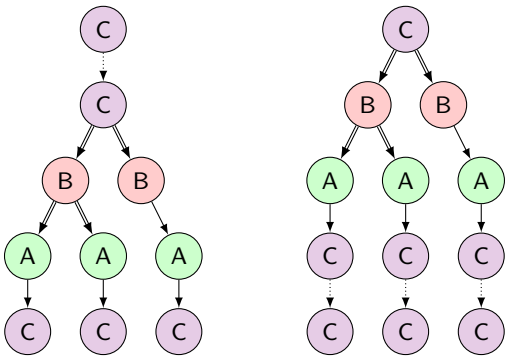

Fig. 6. looping pattern

Pattern 4 (simple cycle) A cycle pattern is completed whenever the path returns to the first register without doubling, then taking any different edge of the cycle after is a doubling pattern. Indeed, any edge after the cycle could be taken before the cycle as shown on Fig. 7.

By considering all these patterns, it seems possible to reduce the number of even solutions and save some useless trail explorations without changing the

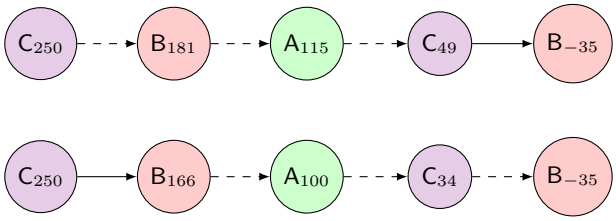

Fig. 7. simple cycle pattern

parity of each solution. However we faced many problems. First adding the constraints for all these patterns slows down the solvers and finding the right trade-off between solution space reduction and time consumption is not easy. Second, we have to ensure a doubling pattern does not interfere with another one. More precisely, let (p_1, p_2) be a doubling pattern. We may have an issue if it is possible to reach a trail containing p_1 while it is impossible to reach p_2 because of another doubling pattern. There are several ways of taking this into account. One option is to apply a constraint if and only if all the nodes involved in a doubling pattern are not reached by other edges than the ones from the pattern. But in practice doing so highly limits the number of times doubling patterns are applied. As a result of our experimentation, we decided to take into account pattern 2 only.

Thus selecting the right patterns to add to the model is still an open and interesting question.

4.2 Use an arity approximation

The idea of approximating the number of cube bits reachable for each bit of the cipher was explored in [16] and we propose to use it to reduce the search space in our graph-based model.

The reasoning on arity is as follows. Starting from the bits of the initialisation vector and going back to the active bit, each intermediate node aggregates an over-approximation of the number of bits of the cube that it would allow to reach if we took it. This value is called arity and is built by consulting all or part of its descendants. Under certain conditions between the arity of a node and that of its predecessors or successors it is possible to deduce whether it may belong to a trail or not.

As shown in [16], an approximation regarding the arity can be computed by recursively taking two consecutive transitions into account, and by propagating the arity from the cube to the output bit.

Example 3. Consider the case where one wants to compute the arity of register C at round 100 and the doubling transitions is selected.

$$ar(\mathsf{C}_{100}) = ar(\mathsf{B}_{18}) + ar(\mathsf{B}_{17}) \tag{1}$$

These terms are developed as follows:

$$ar(\mathsf{B}_{18}) = \max(ar(\mathsf{B}_{-60}), ar(\mathsf{A}_{-48}), ar(\mathsf{A}_{-75}), ar(\mathsf{A}_{-74}) + ar(\mathsf{A}_{-73})) \quad (2)$$
$$ar(\mathsf{B}_{17}) = \max(ar(\mathsf{B}_{-61}), ar(\mathsf{A}_{-49}), ar(\mathsf{A}_{-76}), ar(\mathsf{A}_{-75}) + ar(\mathsf{A}_{-74})) \quad (3)$$

Suppose now that arity of registers with negative round are all equal to the same value, say the value 1. Then (2) and (3) can be simplified to:

$$ar(\mathsf{B}_{18}) = ar(\mathsf{A}_{-74}) + ar(\mathsf{A}_{-73}) \quad (4)$$
$$ar(\mathsf{B}_{17}) = ar(\mathsf{A}_{-75}) + ar(\mathsf{A}_{-74}) \quad (5)$$

By mapping the (4) and (5) in (1), we remark that the arity of A_{-74} is counted twice. In the graph representation, the node labelled A_{-74} will be reachable multiple time from C_{100}. Such an over-approximation would be accumulated along way to the output bit.

For a given node, the approximation of its arity has to take into account all the child cases of the doubling edges and take the maximum of their arity to better approximate the arity of the source. Note that a similar reasoning can also be applied to compute the minimum arity of a node.

This approximation of the arity can then be used as a strategy for a MILP solver or as a constraint for a CP solver. Since the goal is to find the superpoly, it is expected that a significant part of the graph will be cut from the search because of a too low arity.

5 Implementations

In this section we present two implementations of our graph model and discuss our results on Trivium. The first implementation is in Mixed Integer Linear Programming with a relaxed flow problem and the second implementation is in Constraint Programming with a graph variable.

The results presented in this section and in particular the MILP and CP models are publicly available[3].

5.1 MILP

Mixed Integer Linear Programming aims at solving problems described with linear constraints. The MILP graph model is written as a relaxed flow problem. A flow problem is usually defined with the conservation of flows constraint. This constraint states that anything that enter a node must leave it. In our case, this is relaxed because multiple incoming transitions are possible. Having multiple incoming transitions means that a variable is in the monomial multiple times. Regardless of the incoming number of edges, if it is reached, then the out transition is either simple or doubling.

[3] https://gitlab.inria.fr/agontier/trivium-superpoly

In the following, $Pred(i)$ gives all the predecessors of the node i and $Succ(i)$ gives all the linear successors and one of the doubling successor. The functions $brother_1(i)$ and $brother_2(i)$ gives the two doubling sons of i.

First all the edges are declared as Boolean variables:

$$X_{i,j} = \begin{cases} 1 & \text{if the edge } (i,j) \text{ is in the trail} \\ 0 & \text{otherwise} \end{cases}$$

To implement the graph model of TRIVIUM, we add the following constraints:

$$\sum_{j \in Pred(i)} X_{j,i} \geq \sum_{j \in Succ(i)} X_{i,j} \qquad \forall i \in V \tag{6}$$

$$\sum_{j \in Pred(i)} X_{j,i} \leq |Pred(i)| \sum_{j \in Succ(i)} X_{i,j} \qquad \forall i \in V \tag{7}$$

$$X_{i,brother_1(i)} = X_{i,brother_2(i)} \qquad \forall i \in V \tag{8}$$

$$\sum_{j \in Succ(i)} X_{i,j} \leq 1 \qquad \forall i \in V \tag{9}$$

The constraints (6) and (7) are the conservation of flows constraints while (8) and (9) are related to the edges outputting a node (and thus dedicated to TRIVIUM). The cube and the output bit are constrained in the solution by the following:

$$\sum_{j \in Pred(i)} X_{j,i} \geq 1 \qquad \forall i \in cube \tag{10}$$

$$\sum_{j \in Pred(i)} X_{j,i} = 0 \qquad \begin{array}{c} \forall i \in leaves, \\ i \notin cube,\ i \notin key, \\ i \notin non\text{-}zero\text{-}constants \end{array} \tag{11}$$

The key bits are free as well as the non-zero constants because they can also appear in the superpoly.

Constraints for doubling patterns. The MILP model can be strengthened with constraints to discard the doubling patterns. Let P a set of doubling patterns (p_1, p_2) with p_1, p_2 sub-graphs with the same sources and the same leaves.

We used only Pattern 2 for which both the sub-graphs p_1 and p_2 are composed of respectively 5 and 6 edges of the form:

- $p_1 = \{(x_1, y_1), (x_1, y_2), (x_2, y_4), (x_3, y_3), (x_3, y_4)\}$
- $p_2 = \{(x_1, y_1), (x_1, y_2), (x_2, y_2), (x_2, y_3), (x_3, y_3), (x_3, y_4)\}$

Thanks to the equalities of doubling edges we can simplify both p_1 and p_2 such that:

- $p_1 = \{(x_1, y_2), (x_2, y_4), (x_3, y_4)\}$
- $p_2 = \{(x_1, y_2), (x_2, y_2), (x_3, y_4)\}$

Because (x_2, y_4) and (x_2, y_2) cannot be active both at the same time, we can add the inequality $X_{(x_1,y_2)} + X_{(x_2,y_2)} + X_{(x_2,y_4)} + X_{(x_3,y_4)} \leq 2$ to remove both p_1 and p_2. However a problem occurs if the node x_4 consecutive to x_3 is active and reaches y_4. Indeed, the configuration for which the 4 consecutive nodes follow the doubling edges would be removed twice. Thus we modified the inequalitie into:

$$2X_{(x_1,y_2)} + 2X_{(x_2,y_2)} + X_{(x_2,y_4)} + 2X_{(x_3,y_4)} - X_{(x_4,y_4)} \leq 4$$

We verified that adding this inequality to all consecutive nodes leads to the right ANF. In more details, this inequality forbids 3 consecutive nodes to all take the doubling edge and forbids x_2 to take the long edge if x_4 does not take the doubling edge.

Strategy. In both [13] and [15], authors used a divide-and-conquer strategy together with their MILP models. Basically they developed the polynomial of the root node for several hundreds of rounds (between 200 and 400) and then applied their models on each monomial of the polynomial. Without this strategy the solving times are much higher, making unfeasible to retrieve the superpoly in reasonable time. This shows that Gurobi fails to identify the right variables to branch on. While Gurobi does not allow the user to fully control the branch-and-cut strategy, it offers several options to modify it and we mainly used two of them:

- **BranchPriority:** With this option it is possible to give to each variable of the model a priority during the selection of the next variable to branch on. We tried several strategies and it seems that the best choice is to sort the variable according to their arity. More precisely, given an edge (x, y), we chose to:
  1. set a negative priority if $ar(y) \leq 0$, i.e. if the variable y cannot lead to any cube variable;
  2. set the priority to zero for all simple edges, based on the idea that we have to focus on doubling edges;
  3. set the priority to $ar(x)$ for all doubling edges, to focus on the edges which can reach the most cube variables.
- **VarHintVal:** With this option we can tell Gurobi that we think the value of a variable will be in a solution. We chose to set to 0 all simple edges, again to focus on doubling edges.

Using both those options it became unnecessary to use the divide-and-conquer strategy as we reach approximately the same running times with and without it. However we believe there is still room for improvements. First the **BranchPriority** is static while a dynamic approach would be much better. Second, both the options above apply to variables only while we may want to use them on linear combinations of variables. The problem is that if we create a new variable x and add a constraint $x = y + z$, x will be removed from the model during the presolve and it seems Gurobi does not keep its branch priority.

5.2 CP

Constraint programming [18] is a technique for solving combinatorial problems, like MILP. Unlike the latter, it is not necessary to express the rules solely in terms of linear constraints. In addition, CP solves a problem in a way similar to branch-and-bound except that it eliminates, by *filtering*, impossible states or combinations. CP techniques have already been successfully applied to cryptanalytic problems [11,10,5].

The CP graph uses a directed graph variable G. A graph variable G has a domain defined by a graph interval $[\underline{G}, \overline{G}]$. $\underline{G}$ is the lower bound of G and defines nodes and edges that must appear in any solution. In our case, it is declared with the mandatory nodes of the cube. $\overline{G}$ is the upper bound of G and defines nodes and edges that can appear in any solution. In our case, it is the total graph developed from the automaton defined is Section 3. A solution is found when $\overline{G} = \underline{G}$. The solving processes by adding nodes or edges from $\underline{G}$ or by removing nodes or edges from $\overline{G}$. Such modifications are triggered by constraints defining properties on G that need to be satisfied in any solution.

In the following, D is a view of G which only contains doubling edges and the endpoint nodes; L is another view of G wich only contains the long edges and the endpoint nodes; K stores leaf nodes of G. The functions $pred_X(n)$ and $succ_X(n)$ give the predecessors and the successors of a node n in the (sub-)graph X.

The graph model is declared as $(12) \wedge ((13) \vee (14)) \wedge (15) \wedge (16)$ where:

$$|pred_G(n)| > 0 \qquad \forall n \in \underline{G},\ n \neq source \tag{12}$$

$$|succ_G(n)| = 1 \wedge (n, s) \notin D \qquad \forall n \in \underline{G},\ n \notin K, \forall s \in succ_G(n) \tag{13}$$

$$|succ_G(n)| = 2 \wedge (n, s) \in D \qquad \forall n \in \underline{G},\ n \notin K, \forall s \in succ_G(n) \tag{14}$$

$$(n, s_1) \in \underline{G} \iff (n, s_2) \in \underline{G} \qquad \forall n \in \underline{G},\ (n, s_1) \in D, (n, s_2) \in D \tag{15}$$

$$(n, s_1) \notin \overline{G} \iff (n, s_2) \notin \overline{G} \qquad \forall n \in \underline{G},\ (n, s_1) \in D, (n, s_2) \in D \tag{16}$$

The constraint (12) ensures that each node selected in a solution, but the source node, has at least one predecessor. Constraints (13) and (14) maintain the number of successors of each node but the leaf ones. If a given node takes a simple transition then it has exactly one successor; if it takes a doubling transition then it has exactly two successors. The two conditions cannot hold simultaneously. Finally, constraints (15) and (16) ensures that either a single edge or a pair of doubling edges is selected.

The doubling constraints of Section 4.1 are added to the CP model in the form of clauses expressed on the disjoint membership of edges in G and are propagated using a SAT-like constraint. The algorithm for estimating the degree of Trivium-like ciphers [16] can directly be integrated in the graph model as an additional constraint as presented in Section 4.2. Without going into too much details, it imposes to declare RIV (for Reachable Initialisation Vector) additional integer variables. An integer variable v has a domain $[\underline{v}, \overline{v}]$ where $\underline{v}$ (resp. $\overline{v}$) denotes the smallest (resp. the largest) value it can be assigned to. The RIV variables stores, for each node in $\underline{G}$, an approximation of the number of nodes of the cube

it can reach. The algorithm [16] is directly applied dynamically to refine bounds of each RIV_i variable associated to node i, based on RIV_j, $\forall j \in succ_G(i)$. It is important to note that RIV variables are bounded as long as the involved nodes and edges are in $\overline{G}$. When a RIV domain is emptied or is inconsistent with those of its neighbours then the corresponding node is removed from $\overline{G}$.

Strategy Unlike Gurobi, we can fully control the strategy deployed by Choco. However, this solver is inherently sequential and thus we decided to apply the divide-and-conquer strategy to run several instances in parallel. The main issue we faced is that only few instances are hard to solve and thus we regularly need to redivide models in order to maximize the use of available cores.

5.3 Results

We ran our new models together with the ones from both [13] and [15] on our server, limiting the number of available cores to 32. Results are given on Tables 1 and 2 while the cubes used to perform our experiments are detailed on Table 3.

Model	Monomial Prediction [15]	Division Property [13]	MILP Graph	CP Graph
$R = 675$	3m	1m	3s	15s
$R = 735$	4m	2m	10s	31m
$R = 840/1$	472m	269m	10m	> 24h
$R = 840/2$	316m	91m	10m	
$R = 840/3$	351m	108m	6m	
$R = 841$	956m	282m	19m	
$R = 842$	> 24h	990m	182m	

Table 1. Results on Trivium

We see that the graph model of Trivium performs better with the MILP implementation and the Gurobi solver. One explanation might be that Trivium is not highly combinatorial. Indeed, the round function of Trivium has only one nonlinear case and it is a simple product. Our graph model implemented in MILP is consistently much faster than the models from Hu *et al.* based on monomial prediction and from Hao *et al.* based on division property.

Regarding the number of trails outputted by our model, it is reduced by a factor between 2 and 4 which shows how useful are the doubling patterns described in Section 4. But as we explained, we were not able to use all of them. Checking a posteriori the trails for Trivium-842 shows that taking into account all the doubling patterns could remove much more trails. We believe this is an interesting research direction for a future work.

Graph solver	$R = 840/1$	$R = 841$	$R = 842$
without doubling constraints	12 909	30 177	3 188 835
with Pattern 2	5 953	18 929	720 779

Table 2. Number of solutions

Rounds	Cube indices
675	3, 14, 21, 25, 38, 43, 44, 47, 54, 56, 58, 68
735	2, 5, 9, 12, 13, 14, 19, 28, 36, 38, 40, 47, 49, 51, 52, 53, 55, 57, 63, 64, 66, 73, 79
840 /1	$IV \setminus \{34, 47\}$
840 /2	$IV \setminus \{71, 73, 75, 77, 79\}$
840 /3	$IV \setminus \{73, 75, 77, 79\}$
841	$IV \setminus \{9, 79\}$
842	$IV \setminus \{19, 35\}$

Table 3. Cubes used in our experiments for Trivium

6 Conclusion

In this paper, we proposed a graph-based model to recover the exact superpoly of a stream cipher given a cube. Unlike the division property, our graph model is a convenient mathematical object that allows one the use of cipher specific constraints like doubling paths and arity approximation. We show that this graph model can be implemented in MILP and CP. By taking into account some doubling patterns in our model and refining the branch-and-cut strategy, our MILP implementation is faster than existing MILP implementations based on division property [13], or monomial prediction [15] for Trivium and we expect similar results on other stream ciphers.

We opened new research directions and working further on Gurobi strategy may lead to significant improvements of all MILP models used in cryptography as searching for differential characteristics or integral distinguishers. We also believe that our new graph-oriented model can improve the recent work of Hebborn *et al.* [14] regarding lower bounds on the degree of block ciphers.

Acknowledgements The work presented in this article was funded by the French National Research Agency as part of the DeCrypt project (ANR- 18-CE39-0007). The authors would like to express their very great appreciation to Dr Marie Euler from DGA-MI for her valuable and constructive suggestions during the development of this research work.

References

1. Abdelkhalek, A., Sasaki, Y., Todo, Y., Tolba, M., Youssef, A.M.: MILP Modeling for (Large) S-boxes to Optimize Probability of Differential Characteristics. IACR Trans. Symmetric Cryptol. **2017**(4), 99–129 (2017)
2. Aumasson, J., Dinur, I., Meier, W., Shamir, A.: Cube Testers and Key Recovery Attacks on Reduced-Round MD6 and Trivium. In: 16th International Workshop on Fast Software Encryption (FSE'09). Lecture Notes in Computer Science, vol. 5665, pp. 1–22. Springer (2009)
3. Boura, C., Coggia, D.: Efficient MILP modelings for Sboxes and Linear Layers of SPN ciphers. IACR Trans. Symmetric Cryptol. **2020**(3), 327–361 (2020)
4. Cannière, C.D., Preneel, B.: Trivium. In: Robshaw, M.J.B., Billet, O. (eds.) New Stream Cipher Designs - The eSTREAM Finalists, Lecture Notes in Computer Science, vol. 4986, pp. 244–266. Springer (2008)
5. Delaune, S., Derbez, P., Huynh, P., Minier, M., Mollimard, V., Prud'homme, C.: Efficient Methods to Search for Best Differential Characteristics on SKINNY. In: Sako, K., Tippenhauer, N.O. (eds.) 19th International Conference on Applied Cryptography and Network Security, (ACNS'21). Lecture Notes in Computer Science, vol. 12727, pp. 184–207. Springer (2021)
6. Delaune, S., Derbez, P., Vavrille, M.: Catching the Fastest Boomerangs - Application to SKINNY. IACR Trans. Symmetric Cryptol. **2020**(4), 104–129 (2020)
7. Dinur, I., Shamir, A.: Cube Attacks on Tweakable Black Box Polynomials. In: 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT'09). Lecture Notes in Computer Science, vol. 5479, pp. 278–299. Springer (2009)
8. Fages, J.: On the use of graphs within constraint-programming. Constraints An Int. J. **20**(4), 498–499 (2015)
9. Fouque, P., Vannet, T.: Improving Key Recovery to 784 and 799 Rounds of Trivium Using Optimized Cube Attacks. In: 20th International Workshop on Fast Software Encryption (FSE'13). Lecture Notes in Computer Science, vol. 8424, pp. 502–517. Springer (2013)
10. Gérault, D., Lafourcade, P., Minier, M., Solnon, C.: Computing AES related-key differential characteristics with constraint programming. Artif. Intell. **278** (2020)
11. Gérault, D., Minier, M., Solnon, C.: Using Constraint Programming to solve a Cryptanalytic Problem. In: Sierra, C. (ed.) 26th International Joint Conference on Artificial Intelligence (IJCAI'17). pp. 4844–4848. ijcai.org (2017)
12. Gurobi Optimization, LLC: Gurobi Optimizer Reference Manual (2021), https://www.gurobi.com
13. Hao, Y., Leander, G., Meier, W., Todo, Y., Wang, Q.: Modeling for Three-Subset Division Property Without Unknown Subset - Improved Cube Attacks Against Trivium and Grain-128AEAD. In: Canteaut, A., Ishai, Y. (eds.) 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT'20). Lecture Notes in Computer Science, vol. 12105, pp. 466–495. Springer (2020)

14. Hebborn, P., Lambin, B., Leander, G., Todo, Y.: Lower bounds on the degree of block ciphers. In: Moriai, S., Wang, H. (eds.) Advances in Cryptology - ASIACRYPT 2020 - 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7-11, 2020, Proceedings, Part I. Lecture Notes in Computer Science, vol. 12491, pp. 537–566. Springer (2020). https://doi.org/10.1007/978-3-030-64837-4_18, https://doi.org/10.1007/978-3-030-64837-4_18
15. Hu, K., Sun, S., Wang, M., Wang, Q.: An algebraic formulation of the division property: Revisiting degree evaluations, cube attacks, and key-independent sums. In: Advances in Cryptology - ASIACRYPT 2020 - 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7-11, 2020, Proceedings, Part I. Lecture Notes in Computer Science, vol. 12491, pp. 446–476. Springer (2020)
16. Liu, M.: Degree Evaluation of NFSR-Based Cryptosystems. In: 37th Annual International Cryptology Conference (CRYPTO'17). Lecture Notes in Computer Science, vol. 10403, pp. 227–249. Springer (2017)
17. Prud'homme, C., Fages, J., Lorca, X.: Choco solver documentation. TASC, INRIA Rennes, LINA CNRS UMR **6241** (2016)
18. Rossi, F., van Beek, P., Walsh, T. (eds.): Handbook of Constraint Programming, Foundations of Artificial Intelligence, vol. 2. Elsevier (2006)
19. Sun, S., Gérault, D., Lafourcade, P., Yang, Q., Todo, Y., Qiao, K., Hu, L.: Analysis of AES, SKINNY, and Others with Constraint Programming. IACR Trans. Symmetric Cryptol. **2017**(1), 281–306 (2017)
20. Todo, Y.: Structural Evaluation by Generalized Integral Property. In: 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT'15). Lecture Notes in Computer Science, vol. 9056, pp. 287–314. Springer (2015)
21. Todo, Y., Isobe, T., Hao, Y., Meier, W.: Cube Attacks on Non-Blackbox Polynomials Based on Division Property. In: 37th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO'17). Lecture Notes in Computer Science, vol. 10403. Springer (2017)
22. Todo, Y., Morii, M.: Bit-Based Division Property and Application to Simon Family. In: Peyrin, T. (ed.) 23rd International Conference on Fast Software Encryption (FSE'16). Lecture Notes in Computer Science, vol. 9783, pp. 357–377. Springer (2016)
23. Xiang, Z., Zhang, W., Bao, Z., Lin, D.: Applying MILP Method to Searching Integral Distinguishers Based on Division Property for 6 Lightweight Block Ciphers. In: Cheon, J.H., Takagi, T. (eds.) 22nd International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT'16). Lecture Notes in Computer Science, vol. 10031, pp. 648–678 (2016)
24. Ye, C., Tian, T.: Revisit division property based cube attacks: Key-recovery or distinguishing attacks? IACR Trans. Symmetric Cryptol. **2019**(3), 81–102 (2019)
25. Zhou, C., Zhang, W., Ding, T., Xiang, Z.: Improving the MILP-based Security Evaluation Algorithm against Differential/Linear Cryptanalysis Using A Divide-and-Conquer Approach. IACR Trans. Symmetric Cryptol. **2019**(4), 438–469 (2019)