# Automatic Search for Bit-based Division Property

Shibam Ghosh, Orr Dunkelman

Department of Computer Science, University Of Haifa, Haifa, Israel
{sghosh03@campus,orrd@cs}.haifa.ac.il

**Abstract.** Division properties, introduced by Todo at Eurocrypt 2015, are extremely useful in cryptanalysis, are an extension of square attack (also called saturation attack or integral cryptanalysis). Given their importance, a large number of works tried to offer automatic tools to find division properties, primarily based on MILP or SAT/SMT. This paper studies better modeling techniques for finding division properties using the Constraint Programming and SAT/SMT-based automatic tools. We use the fact that the Quine-McCluskey algorithm produces a concise CNF representation corresponding to the division trail table of an Sbox. As a result, we can offer significantly more compact models, which allow SAT and Constraint Programming tools to outperform previous results. To show the strength of our new approach, we look at the NIST lightweight candidate KNOT and Ascon. We show several new distinguishers with a lower data complexity for 17-round KNOT-256, KNOT-384 and 19-round KNOT-512. In addition, for the 5-round Ascon, we get a lower data distinguisher than the previous division-based results.
Finally, we revisit the method to extend the integral distinguisher by composing linear layers at the input and output. We provide a formulation to find the optimal number of linear combinations that need to be considered. As a result of this new formulation, we prove that 18-round KNOT-256 and KNOT-384 have no integral distinguisher using conventional division property and we show this more efficiently than the previous methods.

**Keywords:** Constraint programming, division property, integral cryptanalysis, KNOT, Ascon.

## 1 Introduction

The *Square attack* was introduced by Daemen et al. in [7] to attack the SQUARE block cipher. A variant of this attack was applied to the Twofish cipher by Lucks in [19] and named the *Saturation attack*. These were formalized by Knudsen and Wagner in [16], under the name *Integral cryptanalysis*. The main idea behind the integral attack is to find different properties of a set of ciphertexts corresponding to a set of plaintexts with a certain structure. These properties propagate through different operations of the cipher. Let us consider a set of plaintexts $\mathcal{P}$ from $(\mathbb{F}_2^m)^n$ and then any element of $\mathcal{P}$ can be seen as $(p_1, p_2, \cdots, p_n)$ where

$p_i \in \mathbb{F}_2^m$, i.e., vector of $m$-bit words. Integral distinguisher exploits the propagation of some simple properties of the words from plaintext to ciphertext. The integral distinguisher considers the following properties: ALL (If the word position considers all possible values exactly once), BALANCED (If the word position is zero in the XOR sum of all elements), CONSTANT (If the word position is identical for all vectors). Based on these properties, an attacker can distinguish a cryptographic function from a random function. For example, the well-known 6-round integral distinguisher used in [12], to attack AES [8]. The integral distinguishers have since been also applied to a variety of ciphers [18,35,37].

The Division property was proposed as a generalization of the integral property by Todo at Eurocrypt 2015 [30] and was used in [31] to offer the first attack on the full MISTY1. The division property proposed by Todo was word-based division property, i.e., the propagation of the division property captures information only from the word level. In FSE 2016, Todo and Morii first introduced the bit-based division property [32]. In such bit-based division properties, the propagation captures information at the bit level which naturally captures more information than word-based division properties. The idea of the division property is the same as the integral property: consider an affine subspace of plaintexts and then check if the resulting set of ciphertexts has some balanced bits, i.e., their XOR sum is zero. To detect these balanced bits we consider the algebraic normal form (ANF) of a vectorial Boolean function. Suppose that $f : \mathbb{F}_2^n \to \mathbb{F}_2^n$ is a vectorial Boolean function that maps $x = (x_0, x_1, \cdots, x_{n-1})$ to $y = (f_0(x), f_1(x), \cdots, f_{n-1}(x))$. Let $X \subset \mathbb{F}_2^n$ be an input set and $Y = \{f(x) : x \in X\}$. The bit-based division property exploits the fact that, for some $i \in \{0, 1, \cdots, n-1\}$, $\bigoplus_{y \in Y} y_i = \bigoplus_{x \in X} f_i(x) = 0$ is predictable or not.

### 1.1 Related Work

The bit-based division property is an important tool for integral cryptanalysis. However, finding the bit-based division property is a tedious job. Direct programming approach was used in [31] to find bit-based division properties of SIMON-32 and SIMECK-32. Both ciphers have a block size of 32 bits. Unfortunately, this direct approach fails for larger block sizes used in modern ciphers.

In this case, automatic tools play a significant role. The main idea is to transform this bit-based property search problem into some mathematical problem and use an automatic tool to solve it. In this direction, Xiang et al. first proposed to use Mixed Integer Linear Programming (MILP) based tool in [36]. This approach has been used to attack many ciphers in the last few years [25,27,34]. A different approach suggested by Sun et al. [26] is to use SAT/SMT modeling [6]. Based on this method, Eskandari et al. studied many block ciphers in [11] and built a tool called SOLVATORE. In [14], the authors studied the bit-based division property for the ciphers with complex linear layers and modeled using SAT/SMT tool. Another approach is the use of Constraint Programming (CP) based tools. This approach was proposed in [28] to find the integral distinguisher of the PRESENT [4] block cipher. An extension of the integral cryptanalysis was

proposed by Lambin et al. in [17], where they proposed to compose linear layers in the input and output to extend the distinguisher. With this approach, they found a 10-round distinguisher for the RECTANGLE [38] block cipher.

### 1.2 Our Contribution

Our work aims at providing compact modeling of Sbox to improve the automatic search of bit-based division properties. We use the SAT/SMT and the CP-based automatic approach to find bit-based division properties of all the variants of the KNOT family [33] and Ascon [10]. we also test again some of the previous results on GIFT, Rectangle and PRESENT. While we check our approach for consistency, the comparison allows us to determine that for all tested models, our approach signficantly reduces the running times of the tools. We express the propagation of bit-based division properties using Boolean logical formulas. We observe that modeling a formula in the Conjunctive Normal Form (CNF) instead of the table-based approach used in [14], gives a significant advantage in performance. We also provide a comparative analysis of these two methods. The above-mentioned tool SOLVATORE [11] was also modeled using the CNF, where the authors used the trivial approach to find the CNF of a function. Here we propose to use the Quine-McCluskey algorithm [20,22,23] to find the minimum size CNF. The Quine-McCluskey algorithm was previously used in the context of differential cryptanalysis in [1]. From our result, we can observe that for the KNOT and Ascon, the CP-based approach outperforms the SAT-based approach.

We also provide a concrete algorithm for finding lower data distinguishers. This algorithm is a formalization of two previous works in [11,26]. We used our algorithm on the KNOT and Ascon and found many distinguishers, which are more efficient. Table 1 compares the known results and our results.

We studied the direction provided in [17] to extend distinguishers by composing linear layers at the input and output. For the output layer, we used linear combinations instead of linear maps to reduce the search space, as suggested in [9]. Here we provide a formal way to find the optimal number of linear combinations that need to be considered, using Depth First Search (DFS) to find these. As an application of this theory, we found a new result that 18-round KNOT-256 and KNOT-384 have no integral distinguisher using conventional division property and we proved that more efficiently than the usual method.

## 2 Preliminaries

### 2.1 Notations

The Hamming weight of $a \in \mathbb{F}_2^n$ is $wt(a) = \sum_{i=1}^{i=n} a_i$ and for any vector $\boldsymbol{a} = (a_0, a_1, \cdots, a_{m-1}) \in \mathbb{F}_2^{l_0} \times \mathbb{F}_2^{l_1} \times \cdots \times \mathbb{F}_2^{l_{m-1}}$, the vectorial Hamming weight of $\mathbf{a}$ is $W(\boldsymbol{a}) = (wt(a_0), wt(a_1), \cdots, wt(a_{m-1})) \in \mathbb{Z}^m$. For any $\boldsymbol{k} \in \mathbb{Z}^m$ and $\boldsymbol{k'} \in \mathbb{Z}^m$, we define $\boldsymbol{k} \succeq \boldsymbol{k'}$ if $k_i \geq k'_i$ for all $i$. For any integer $k \in \{0, 1, .., n\}$ we define

| Primitive | #Rounds | Data | #Balanced bits | Source |
|---|---|---|---|---|
| KNOT-256 | 17 | $2^{255}$ | 1 | [33] |
| | 17 | $2^{254}$ | 7 | Sec 5 |
| | 18 | Does not exist | | Sec 7 |
| KNOT-384 | 17 | $2^{383}$ | 1 | [33] |
| | 17 | $2^{380}$ | 19 | Sec 5 |
| | 18 | Does not exist | | Sec 7 |
| KNOT-512 | 19 | $2^{511}$ | 1 | [33] |
| | 19 | $2^{508}$ | 139 | Sec 5 |
| Ascon | 5 | $2^{16}$ | 320 | [11] |
| | 5 | $2^{12}$ | 2 | Sec 6 |

Table 1: Summary of previous results and our best results

the set $\mathbb{S}_k^n = \{a \in \mathbb{F}_2^n : k \leq wt(a)\}$ and for any vector $\boldsymbol{k} \in (\{0, 1, .., n\})^m$ we define the set $\mathbb{S}_{\boldsymbol{k}}^{m,n} = \{\boldsymbol{a} = (a_1, a_2, ..., a_m) \in (\mathbb{F}_2^n)^m : \boldsymbol{k} \preceq W(\boldsymbol{a})\}$. For any vector $u \in \mathbb{F}_2^n$ and $x \in \mathbb{F}_2^n$, we define the *bit product* $\pi_u : \mathbb{F}_2^n \to \mathbb{F}_2$ as $\pi_u(x) = \prod_{i=1}^n {x_i}^{u_i}$ and for any vector $\boldsymbol{u} \in (\mathbb{F}_2^n)^m$ and $\boldsymbol{x} \in (\mathbb{F}_2^n)^m$, we define the *vectorial bit product* $\pi_{\boldsymbol{u}} : (\mathbb{F}_2^n)^m \to \mathbb{F}_2^n$ as $\pi_{\boldsymbol{u}}(\boldsymbol{x}) = \prod_{i=1}^m \pi_{u_i}(x_i) = \prod_{i=1}^m \left( \prod_{j=1}^n x_{i_j}^{u_{i_j}} \right)$. The Algebraic Normal Form (ANF) of a function $f : \mathbb{F}_2^n \to \mathbb{F}_2$ can be defined as $f(x) = \bigoplus_{u \in \mathbb{F}_2^n} a_u^f \pi_u(x)$ and the degree of a function $f : \mathbb{F}_2^n \to \mathbb{F}_2$ is $d$ if $d$ is the degree of the largest monomial in the ANF of $f$, i.e., $d = \max_{u \in \mathbb{F}_2^n, a_u^f \neq 0} wt(u)$. We define $in_{i_1,i_2,...,i_p}$ is the vector in $\mathbb{F}_2^n$ with all coordinates 1 expect for the positions $i_1, i_2, ..., i_p$ and we define $out_{j_1,j_2,...,j_p}$ is the vector in $\mathbb{F}_2^n$ with all coordinates 0 expect for the positions $j_1, j_2, ..., j_p$. Similarly, we use $in_{(k,\ell)}$ to denote the binary matrix with all the elements are 1 except for $(k, \ell)$ position and $out_{(k,\ell)}$ to denote the binary matrix with all the elements are 0 except for $(k, \ell)$ position.

### 2.2 Definitions

**Definition 1.** *(Division Property [30]) A multi-set $X \subseteq \mathbb{F}_2^n$ is said to have the division property of order $k$, $\mathcal{D}_k^n$ for some $1 \leq k \leq n$, if the sum over all vectors $x \in X$ of the product $x^u = 0$, for all vectors $u$ with hamming weight less than $k$, i.e.,*

$$\bigoplus_{x \in X} \pi_u(x) = 0, \forall u \in \mathbb{F}_2^n \text{ with } wt(u) < k.$$

**Definition 2.** *(Vectorial Division Property [30]) A multi-set $X \subseteq \mathbb{F}_2^{l_0} \times \mathbb{F}_2^{l_1} \times \cdots \times \mathbb{F}_2^{l_{m-1}}$ is said to have the division property $\mathcal{D}_{\mathbb{K}}^{l_0,l_1,\cdots l_{m-1}}$ for some set of $m$-dimensional vectors $\mathbb{K}$ whose $i$-th element takes a value between 0 to $l_i$, it fulfills the following conditions:*

$$\bigoplus_{\boldsymbol{x} \in X} \pi_{\boldsymbol{u}}(\boldsymbol{x}) = \begin{cases} unknown, & \text{if there is } \boldsymbol{k} \in \mathbb{K} \text{ s.t } W(\boldsymbol{u}) \succeq \boldsymbol{k} \\ 0, & otherwise \end{cases}$$

*Moreover, if each $l_i$ is restricted to 1, we will say bit-based division property and we will denote it by $\mathcal{D}_{\mathbb{K}}^{1,n}$.*

**Definition 3.** *(Balanced Position [30]) Let $Y \subseteq \mathbb{F}_2^n$ be a multi-set of vectors. A coordinate position $0 \leq i < n$ is called balanced position if $\bigoplus_{y \in Y} y_i = 0$.*

**Definition 4.** *(Even Polynomial) Let $f$ be a polynomial in the ring*

$$\mathbb{F}_2[x_0, x_1, ....., x_{m-1}]/(x_0^2 + x_0, x_1^2 + x_1, ....., x_{n-1}^2 + x_{m-1})$$

*with the algebraic normal form (ANF) $f(x_{m-1}, \cdots, x_0) = \bigoplus_{u \in \mathbb{F}_2^m} a_u^f \pi_u(x)$. Then $f$ is called a even polynomial over a multiset $X$ if the following holds $\forall u \in \mathbb{F}_2^m$:*

$$a_u^f \bigoplus_{x \in X} \pi_u(x) = 0.$$

## 3 Propagation of Bit-based Division Property

Let us consider a function $F : \mathbb{F}_2^n \to \mathbb{F}_2^n$. This function can be an Sbox, linear function or even a round function. We are interested in how the division property can propagate through this function. We consider the input set $X$ as an affine subspace. Suppose $X$ has division property $\mathcal{D}_{\{k_0\}}^{1,n}$ and after propagation through $F$, we get a division property $\mathcal{D}_{\{k_1\}}^{1,n}$. If $k_0 = (k_0^0, k_1^0, \cdots, k_{n-1}^0)$ and $k_1 = (k_0^1, k_1^1, \cdots, k_{n-1}^1)$, then we call $(k_0^0, k_1^0, \cdots, k_{n-1}^0, k_0^1, k_1^1, \cdots, k_{n-1}^1)$ a valid *division trail* through $F$. A formal definition of *division trail* was given in [36]. Here we recall the definition.

**Definition 5.** *Let $f_r$ denote the round function of an $r$ round iterative primitive. Suppose the initial division property is $\mathcal{D}_{\{k_0\}}^{1,n}$ and after $(i-1)$-round propagation, the division property is $\mathcal{D}_{\mathbb{K}_i}^{1,n}$. Then we have the following chain of division property propagations:*

$$\{k_0\} := \mathbb{K}_0 \xrightarrow{f_0} \mathbb{K}_1 \xrightarrow{f_1} \mathbb{K}_1 \xrightarrow{f_2} \cdots$$

*Moreover, for any vector $k_i \in \mathbb{K}_i$ $(i \geq 1)$, there must exist a vector $k_{i-1} \in \mathbb{K}_{i-1}$ such that $k_{i-1}$ can propagate to $k_i$ by division property propagation rules. For $(k_0, k_1, \cdots, k_{r-1})$, if $k_{i-1}$ can propagate to $k_i$ for all $i \in \{1, 2, \cdots, r\}$, we call $(k_0, k_1, \cdots, k_r)$ an $r$-round division trail.*

Our main motivation is to model such a search problem that each solution of it be a valid division trail. Then we can solve that problem using various tools like constraint programming, SAT solver, etc. Suppose we are given an $r$-round primitive $E_r : \mathbb{F}_2^n \to \mathbb{F}_2^n$. Let $(a_0, a_1, \cdots, a_{n-1})$ be the variables denoting the input division property vectors and $(b_0, b_1, \cdots, b_{n-1})$ be the variables denoting the output division property vectors. We set values to the input variables $(a_0, a_1, \cdots, a_{n-1})$ of the first round by a vector $k \in \{0,1\}^n$ of our choice and find the balanced positions in the output vector from the last round. Once we

have a set of balanced positions corresponding to an input division property $k$, we can distinguish $E_r$ from a random function. For this, we take a set $X \subset \mathbb{F}_2^n$ of plaintexts and get an output set $Y$ such that $Y = \{y = E_r(x) \mid x \in X\}$. The set $X$ is an affine subspace, constructed corresponding to the input division property $k$. For each vector $x = (x_0, \cdots, x_{n-1}) \in X$, if the $i$-th coordinate of $k$ is 1 then $x_i$ can accepts all possible values from $\{0, 1\}$ and if the $i$-th coordinate of $k$ is 0 then $x_i$ is set to a fixed constant $c_i \in \{0, 1\}$. From the division property, we can guarantee that the balanced positions of the vectors of $Y$ are balanced, which can distinguish $E_r$ from a random function. As the size of the set $X$ is $2^{wt(k)}$, the data complexity is $2^{wt(k)}$.

### 3.1 Modeling the Sbox

From the above discussion, we can observe that modeling a primitive is the main tool of the attack. We now discuss how to model an Sbox. Let $x = (x_{n-1}, \cdots, x_1, x_0)$ be the input and $y = (y_{n-1}, \cdots, y_1, y_0)$ be the corresponding output of an Sbox. Then Algorithm 1 implements the approach to find division trails of the Sbox [36]. This algorithm takes the input division property vector $k = (k_0, k_1, \cdots, k_{n-1})$ as the input and outputs a set of vectors $\mathbb{K}_k$ such that the output multi-set has division property $\mathcal{D}^{1,n}_{\mathbb{K}_k}$. Here we have denoted the output set as $\mathbb{K}_k$ to attach it with the input property $k$.

**Algorithm 1** SboxDivisionTrail [36]

**Require:** $k = (k_0, k_1, \cdots, k_{n-1})$
**Ensure:** A set $\mathbb{K}$ of vectors
$\mathbb{S}_k = \{a \mid a \succeq k\}$
$F(X) = \{\pi_a(x) \mid a \in \mathbb{S}_k\}$
$\mathbb{K} = \phi$
**for** $u \in \mathbb{F}_2^n$ **do**
  **if** $\pi_u(x) \cap F(X) \neq \phi$ **then**
    $Flag = True$
    $R = \phi$
    **for** $v \in \mathbb{K}$ **do**
      **if** $v \succeq u$ **then**
        $Flag = False$
      **else if** $u \succeq v$ **then**
        $R = R \cup \{v\}$
      **end if**
    **end for**
    **if** $Flag = True$ **then**
      $\mathbb{K} = \mathbb{K} \backslash R$
      $\mathbb{K} = \mathbb{K} \cup \{u\}$
    **end if**
  **end if**
**end for**
**return** $\mathbb{K}$

Here let us explain Algorithm 1. Note that $\pi_u(y)$ is an $n$-variable Boolean function. Also let us consider a set $\mathbb{S}_k = \{a \mid a \succeq k\}$ (see $\mathbb{S}_{\mathbf{k}}^{m,n}$ in notation). If the ANF of $\pi_u(y)$ contains any monomial from $F(X) = \{\pi_a(x) \mid a \in \mathbb{S}_k\}$ then $\bigoplus_{x \in X} \pi_{u'}(y)$ is unknown for any $u' \succeq u$, follows from the definition of division property. So we have to add $u$ in the set $\mathbb{K}_k$. But before that, we check if there is any vector $v \in \mathbb{K}_k$ such that $u \succeq v$. If such $v$ is there, then $u$ is redundant, and there is no need to add $u$. On the other hand, if there is some vector $v$ such that $v \succeq u$, then by adding $u$, $v$ is redundant. We store all such vectors in the redundant vector's set $R$ and we remove $R$ after adding $u$. Thus Algorithm 1 finds the output division property vectors set as well as reduces its size by removing redundant vectors.

Given an Sbox and an initial division property vector $k = (k_0, k_1, \cdots, k_{n-1})$, Algorithm 1 returns the set $\mathbb{K}_k$. Then for any $u \in \mathbb{K}_k$, $(k, u)$ is a valid division trail. In other words, any $(k, u) \in \mathbb{F}_2^n \times \mathbb{F}_2^n$ is a valid division trail if and only if $u \in \mathbb{K}_k$, where $\mathbb{K}_k$ is the output of Algorithm 1 on input $k$.

### 3.2 CNF from Division Trail

As discussed in the previous part we can form a division trail table $T$, such that

$$T = \{(a, b) \in \mathbb{F}_2^{2n} \mid b \in \mathbb{K}_a\}.$$

For example let us consider the Sbox of the KNOT permutation [33] $S : \mathbb{F}_2^4 \to \mathbb{F}_2^4$ such that $S((x_3, x_2, x_1, x_0)) = (y_3, y_2, y_1, y_0)$. The algebraic normal form (ANF) of its 4 coordinates are given in Equation 1 and the table corresponding to $T$ is given in Table 2.

$$\begin{aligned} y_0(x) &= x_0x_1x_3 + x_0x_1 + x_0x_2 + x_1x_3 + x_2x_3 + x_2 + x_3 \\ y_1(x) &= x_0x_3 + x_1x_2x_3 + x_1 + x_2x_3 + x_2 \\ y_2(x) &= x_0 + x_1x_2 + x_1 + x_2 + x_3 + 1 \\ y_3(x) &= x_0x_1 + x_1 + x_2 + x_3 \end{aligned} \tag{1}$$

From the construction of $T$ it is clear that $T^c = \mathbb{F}_2^{2n} \backslash T$ contains all the invalid division trails. We can consider a Boolean function $F_S$ from $\mathbb{F}_2^{2n}$, corresponding to a given Sbox $S$ with the following property:

$$F_S(a, b) = \begin{cases} 1, \text{ if } (a, b) \in T \\ 0, \text{ if } (a, b) \in T^c \end{cases}$$

Here we are interested in modeling this function to SAT/CP formula. One idea of this kind of modeling is the table-based approach, used in [14]. This table-based approach is the same as considering the disjunctive normal form (DNF) of $F_S$. However, we observe that the performance of this model is very low. Hence, we propose modeling using the conjunctive normal form (CNF) of the function $F_s$. The difference between the time requirements of these two methods suggests that we can significantly improve the performance by using the CNF instead

| $k$ | $\mathbb{K}_k$ |
|---|---|
| 0000 | {0000} |
| 0001 | {0001, 0010, 0100, 1000} |
| 0010 | {0001, 0010, 0100, 1000} |
| 0011 | {0001, 0110, 1000} |
| 0100 | {0001, 0010, 0100, 1000} |
| 0101 | {0001, 0110, 1010, 1100} |
| 0110 | {0010, 0100, 1001} |
| 0111 | {0111, 1010, 1101} |
| 1000 | {0001, 0010, 0100, 1000} |
| 1001 | {0001, 0010, 1100} |
| 1010 | {0001, 0010, 1100} |
| 1011 | {0001, 0110, 1100} |
| 1100 | {0001, 0010, 1100} |
| 1101 | {0111, 1001, 1010} |
| 1110 | {0010, 0101, 1100} |
| 1111 | {1111} |

Table 2: Valid division trail table of the KNOT Sbox

of the DNF. Let us discuss how to compute the CNF of a given function $F_S$. We do so by first computing another function $G(x) = \overline{F_s(x)}$, i.e., $G(a, b) = 1$ if $(a, b) \in T^c$. The disjunctive normal form (DNF) of the function $G$ can be trivially found, as used in [11]. Then we can again convert $G(x)$ to $\overline{G(x)}$ to get the CNF of $F_s(x)$ using De Morgan's laws. However, this approach results in a huge CNF representation. The number of terms in the CNF is the same as the size of $T^c$. To counter this effect, we propose to use the Quine-McCluskey algorithm [20,22] to find a minimum size CNF. We provide a note about the Quine-McCluskey algorithm and the CNF corresponding to the KNOT Sbox resulting from this algorithm in Appendix C.

We modeled the division property propagation problem using two different tools: as an SMT problem and as a Constraint programming problem. This was done to identify which of these two approaches are better for the CNF clauses. As we report in Section 5, CP based approach on the CNF, is more efficient. There are many public solvers to solve SAT and SMT problems. Here we construct our model using the CVC [3] language and give it to an SMT solver. The SMT solver solves the satisfiability problem with the help of an SAT solver. We used the STP 2.3.3[1] [13] as the SMT solver and the Cryptominisat 5.7.1 [24] as the SAT solver. We modeled our CP problem in MiniZinc [21], which is a solver-independent open-source language that can be used to express CP models readable by multiple solvers. Then the model is given to the publicly available solver Chuffed[2] 0.10.4 [5]. We provided a small tutorial on MiniZinc in Appendix B.

[1] https://stp.github.io/.
[2] https://github.com/chuffed/chuffed.

## 4 Input Division Property and Output Division Property

Consider an SPN structure block cipher or a permutation. Each round function of this structure consists of parallel applications of a certain number of Sboxes, followed by a linear layer. We construct a model for each layer and we repeat the procedure $r$ times for an $r$-round primitive. Then we set values to the input variables $a = (a_0, a_1, \cdots, a_{n-1})$ and the output variables $b = (b_0, b_1, \cdots, b_{n-1})$ and solve the model. We now have to choose those values for $a$ and $b$.

### 4.1 Input Division Property

Our initial division property is selected on the basis of the *embedded property*, introduced in [26], whose definition is recalled as follows.

**Proposition 1 (Embedded Property [26]).** *Let $E_r$ be an $r$-round iterated encryption algorithm, $R$ be the round function, which only composes of Substitution, Copy, XOR, Split and Concatenation operations. Suppose that the input and the output take values from $\mathbb{F}_2^n$ and $k_0$, $k_1$ are two initial division properties with $k_0 \succeq k_1$. If the output multi-set under $k_0$ does not have integral property, then the output multi-set under $k_1$ also has no integral property.*

Thus we consider $n$ vectors with Hamming weight $n-1$ as $in_i = (1, ..., 1, 0, 1, ..., 1)$ (0 is in the $i$-th position) for $0 \leq i \leq n-1$. If we start from each of these $in_i$, i.e., set $a = in_i$ and cannot find any integral distinguisher then we can conclude that there is no integral distinguisher. We do not to check with the other initial properties according to the above Proposition 1.

### 4.2 Output Division Property

The choice of the output division property depends on when we need to stop the search, i.e., when we get a set without an integral property. This is described in the following proposition from [36].

**Proposition 2 ([36]).** *Let $\mathbb{X}$ be a multi-set with bit-based division property $\mathcal{D}_{\mathbb{K}}^n$, then $\mathbb{X}$ does not have integral property iff $\mathbb{K}$ contains all vectors of weight 1.*

Thus here, if the output multi-set (set of ciphertexts) has the division property $\mathbb{K}$ where $\mathbb{K}$ contains each $out_j$, then the output multi-set has no integral distinguishers. So we set the output variable with each $out_j$ and solve the model.

### 4.3 Automatic Algorithm for Finding Division Property

Now we recall an algorithm to find the maximum round $r$ for which we can find an integral distinguisher from [26]. If for $r$-round, the system is consistent (satisfiable) for all $in_i$ and $out_j$ then we have no distinguisher, because from any input division property, the set of output division property contains all $out_j$. In that case, we set $r-1$ as the highest possible round. On the other hand, if for some $in_i$ and $out_j$, the model is not consistent with $a = in_i$ and $b = out_j$, then we can conclude that the $j$-th bit of the output multi-set is balanced. In that case, we try the search by increasing another round.

### 4.4 Reduction of Data Complexity

Till now, we have discussed how to find the maximal number of rounds with a division property. Suppose that for some $in_i$ and $out_j$, the model is inconsistent with the input variable $a = in_i$ and output variable $b = out_j$, i.e., there is an integral distinguisher. This distinguisher uses a set of plaintext vectors $\mathbb{X} \subset \mathbb{F}_2^n$ such that the $i$-th bit is constant and the other bits take all possible ($\{0, 1\}$) values. The data complexity of this distinguisher is $2^{n-1}$ plaintexts. We now discuss an idea from [26] to reduce the data complexity.

We first find an index set $\mathbb{S}$ such that for each $i \in \mathbb{S}$, if we set the initial division property as $in_i$, we have at least one $j$ such that the $j$-th bit of the output multi-set is balanced. The set $\overline{\mathbb{S}} = \{0, 1, \cdots, n-1\} \backslash \mathbb{S}$ is called the *necessary set* [26]. This name *necessary set* follows from the fact that we can get balanced bit at the output only if we set $a_i = 1$ for all $i \in \overline{\mathbb{S}}$. The set $\mathbb{S}$ also called as *sufficient set* [26]. To choose an index $i$ such that $a = in_i$, the set $\mathbb{S}$ is sufficient. Now if $\mid \mathbb{S} \mid > 1$, then we may set more than one $a_i = 0$ where $i \in \mathbb{S}$ and still have some balanced bits. Suppose we choose $m$ indices $\{i_0, i_1, \cdots, i_{m-1}\}$ from $\mathbb{S}$ and set $a_i = 0$ for all $i \in \{i_0, i_1, \cdots, i_{m-1}\}$ and still have some balanced bit. In that case, we have an integral distinguisher with data complexity of $2^{n-m}$.

Given the sufficient set $\mathbb{S}$, we can try with all possible subsets from $\mathbb{S}$, as suggested in [26], to see which offers the best data complexity. If we get a balanced bit for some subset, we stop this search; otherwise, we continue. This strategy brute forces all the subsets, i.e., its worst-case time complexity is $\binom{|\mathbb{S}|}{t}$ calls to the solver.

An improved idea proposed in [11], is to test only those combinations of indices from $\mathbb{S}$ which already have common balanced bits. We justify here the idea with a small example. Consider a function $f : \mathbb{F}_2^4 \to \mathbb{F}_2^4$ which maps $x_3, \cdots, x_0$ to $y_3, \cdots, y_0$ with degree at most 3. We can write each $y_i$ as a polynomial in $x_3, \cdots, x_0$. The division property depends on the presence of monomials in the polynomial representations (ANF) of $y_i$ as we discussed in the case of Sbox. Suppose for both $a = in_0$ and $a = in_1$ we get $y_0$ is balanced. Then it is clear that the monomials $x_2x_1x_0$ and $x_3x_1x_0$ are not present in the polynomial representation of $y_0$. Here, we observe that $x_1x_0$ can be involved only in the following monomials:

$$x_1x_0, x_2x_1x_0, x_3x_1x_0.$$

As we already have that the monomials $x_2x_1x_0$ and $x_3x_1x_0$ absent in the polynomial representation of $y_0$, if $x_1x_0$ is also absent in the polynomial of $y_0$, then we can set the initial division property $a = \{0, 0, 1, 1\}$ and still have balanced $y_0$. This integral distinguisher has a lower data complexity. We alert the reader that if $x_1x_0$ is not absent, then $y_0$ is not balanced with the initial division property $a = \{0, 0, 1, 1\}$, but this heuristics the chance of $y_0$ to be balanced.

Now we discuss the idea from [11] more formally and we provide an algorithm to use this search process. Let us consider a set $OUT_i$ of all indices $j$ such that the $j$-th bit is balanced when the input division property is $in_i$. We find

the following set

$$IN_2 = \{\{i_0, i_1\} : (i_0 \neq i_1) \wedge (OUT_{i_0} \cap OUT_{i_1} \neq \phi), \forall i_0, i_1 \in \mathbb{S}\}.$$

We test if there are some balanced bits with initial division property $in_{i_0,i_1}$ for each $\{i_0, i_1\} \in IN_2$. If we can find some balanced bit for some $\{i_0, i_1\}$ then we have a lower data distinguisher. Note that in this stage we are checking only with those $out_j$ such that $j \in (OUT_{i_0} \cap OUT_{i_1})$, i.e., which are already balanced in the previous stage. This gives a significant advantage over searching for all $out_j$. The idea can be trivially generalized to $IN_m$ where $IN_m = \{\{i_0, i_1, \cdots, i_{m-1}\} : (i_0 \neq i_1 \neq \cdots \neq i_{m-1}) \wedge ((\cap_{j=0}^{m-1} OUT_{i_j}) \neq \phi), \forall i_0, \cdots, i_{m-1} \in \mathbb{S}\}$. According to this, we can take $IN_1 = \mathbb{S}$.

This search starts from lowest value of $m$, i.e., $m = 2$ and increases by 1 if there is some balanced bits from an element of $IN_m$. To justify this we are proposing the following new Proposition 3.

**Proposition 3.** *Let $m_0$ and $m_1$ be two non-zero integers with $m_0 < m_1$. If we cannot get any integral distinguisher by setting the initial division property $a = in_{i_0,i_1,\cdots,i_{m_0-1}}$ for each $\{i_0, i_1, \cdots, i_{m_0-1}\} \in IN_{m_0}$ then there is no integral property from any index set of $IN_{m_1}$.*

*Proof.* If $m_1 > m_0$, then for any element $\{j_0, j_1, \cdots, j_{m_1-1}\} \in IN_{m_1}$ there is some element $\{i_0, i_1, \cdots, i_{m_0-1}\} \in IN_{m_0}$ such that $in_{i_0,i_1,\cdots,i_{m_0-1}} \succeq in_{j_0,j_1,\cdots,j_{m_1-1}}$. So the proof follows from the embedded property in Proposition 1.

It was also suggested in [11] to continue the process until we find some $m$ such that $IN_m$ is empty. But from the above Proposition 3 it also follows that we can stop our search when $m$ is such that no element of $IN_m$ leads to a balanced bit.

Algorithm 2 captures the above proposition. This algorithm outputs the size of the maximum possible combination from $\mathbb{S}$ for which we can get an integral distinguisher with data complexity $2^{n-t}$. This algorithm also outputs a set $Z = \{i_0, i_1, \cdots, i_{t-1}\}$ of such a combination. Note that if the algorithm cannot find any larger combination, we can take any element from $IN_1$, so $Z$ can be initialized with any one element from $IN_1$. In Algorithm 2 we initialized $Z$ with an $i \in IN_1$ such that it gives a maximum number of balanced bits. Finally, once we have such a set $Z$ we can get balanced bits corresponding to the initial division property $in_{i_0,i_1,\cdots,i_{n-1}}$. Here if we keep track of the list $OUT$ from Algorithm 2, then we can use that to find balanced bits efficiently. In that case we search only on $\cap_{i \in \{i_0,i_1,\cdots,i_{t-1}\}} OUT[i]$ for balanced bits.

*Remark 1.* One important remark is that the time complexity of Algorithm 2 may not be feasible if the size of $\mathbb{S}$ is very large. The main work is needed here to compute the sets $OUT_i$ for all $i \in \mathbb{S}$. For every element in $\mathbb{S}$ we need to call our model $n$ times. If this is computationally infeasible, one can sample a smaller subset of $\mathbb{S}$ as we show in the next section.

**Algorithm 2** OptimalDistinguisher

**Require:** SAT/CP model for the primitive, Max round $r > 1$, Sufficient set $\mathbb{S}$
**Ensure:** $t$ and $Z$ such that $t$ is the max possible size and $Z$ is one combination of size $t$

$IN_1 = \mathbb{S}$
$OUT$ is empty list of sets
**for** $i \in IN_1$ **do**
    $OUT_i = \phi$
    $a = (a_0, \cdots, a_{n-1}) = in_i$
    **for** $0 \leq j < n$ **do**
        $b = (b_0, \cdots, b_{n-1}) = out_j$
        {solve the $r$-round model with $a$ and $b$
        as first round and last round variable, respectively}
        **if** not satisfiable **then**
            $OUT_i = OUT_i \cup \{j\}$
        **end if**
    **end for**
    $OUT[i] = OUT_i$
**end for**
$Flag = True$
$t = 1$
$Z = \max_{i \in IN_1} \mid OUT[i] \mid$
**while** $Flag = True$ **do**
    $FLAG = False$
    $t = t + 1$
    compute $IN_t$
    **for** $\{i_0, i_1, \cdots, i_{t-1}\} \in IN_t$ **do**
        $a = in_{i_0, i_1, \cdots, i_{t-1}}$
        $B = \cap_{i \in \{i_0, i_1, \cdots, i_{t-1}\}} OUT[i]$
        **for** $(b_0, \cdots, b_{n-1}) \in B$ **do**
            $b = (b_0, \cdots, b_{n-1}) = out_j$
            {solve the $r$-round model with $a$ and $b$
            as first round and last round variable, respectively}
            **if** not satisfiable **then**
                $Flag = True$
                $Z = \{i_0, i_1, \cdots, i_{t-1}\}$
                break
            **end if**
        **end for**
        **if** $Flag = True$ **then**
            break
        **end if**
    **end for**
**end while**
$t = t - 1$
**return** $t, Z$

## 5 Application to the KNOT permutation

KNOT is a family of bit-slice lightweight authenticated encryption algorithms and hash functions [33], submitted to the NIST lightweight crypto competition [29]. The KNOT permutation is the main primitive used in the KNOT family and comprises three variants with different sizes: 256 bits, 384 bits and 512 bits (denoted by KNOT-b).

### 5.1 Specification

The underlying permutations iteratively applies an SP-network round transformation. Each round is the composition of three operations: Add round constants, Sub Column, Shift Row. The round constants do not affect the division property. So we do not describe them here and refer the interested reader to [33]. Each $b$-bit state of the KNOT-b can be seen as a $4 \times \frac{b}{4}$ matrix, where $b = 256, 384$ or 512.

The operation of Sub Column is a parallel application of $\frac{b}{4}$ similar Sboxes to the 4 bits in the same column. The Sbox $S : \mathbb{F}_2^4 \to \mathbb{F}_2^4$ is given in Table 3. The Shift Row transformation left rotate each row by 0, $c_1$, $c_2$ and $c_3$ bits, respectively. The offsets $c_1$, $c_2$ and $c_3$ are different for different state size, given in Table 4.

| $x$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | $A$ | $B$ | $C$ | $D$ | $E$ | $F$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $S(x)$ | 4 | 0 | $A$ | 7 | $B$ | $E$ | 1 | $D$ | 9 | $F$ | 6 | 8 | 5 | 2 | $C$ | 3 |

Table 3: KNOT's Sbox

| $b$ | $c_1$ | $c_2$ | $c_3$ |
|---|---|---|---|
| 256 | 1 | 8 | 25 |
| 384 | 1 | 8 | 55 |
| 512 | 1 | 16 | 25 |

Table 4: Shift Row offsets for the KNOT permutation

### 5.2 Application of Our Model

We applied both SAT and CP models to KNOT. When the state size is $b$, we have in total $\frac{b}{4}$ many 4-bit Sboxes for each round. Thus we have a total $\frac{b}{4} \times r$ many constraints for $r$ rounds. To implement $r$-round KNOT we have the variable

matrices $a^0, a^1, a^2, ..., a^r$ of the form

$$a^i = \begin{bmatrix} a^i_{0,0}\ a^i_{0,1}\ a^i_{0,2} \cdots a^i_{0,\frac{b}{4}-1} \\ a^i_{1,0}\ a^i_{1,1}\ a^i_{1,2} \cdots a^i_{1,\frac{b}{4}-1} \\ a^i_{2,0}\ a^i_{2,1}\ a^i_{2,2} \cdots a^i_{2,\frac{b}{4}-1} \\ a^i_{3,0}\ a^i_{3,1}\ a^i_{3,2} \cdots a^i_{3,\frac{b}{4}-1} \end{bmatrix} \forall i \in \{0, 1, ..., r\},$$

where each $a^i_{j,k} \in \{0, 1\}$. In our model, the variables are related with some constraints. Each column of $a^i$ and $a^{(i+1)}$ are related with parallel application of $\frac{b}{4}$ Sboxes. Then $a^i$ is rotated according to the shift row and we get $b^i$. Note that as this is only a permutation of variables we do not need to introduce new variables for $b^i$, instead we can just connect $a^i$ and $b^i$ according to shift row. The chain of propagation is as follows (omitted last shift row):

$$a^0 \xrightarrow{\text{Sbox}} a^1 \xrightarrow{\text{rotation}} b^1 \xrightarrow{\text{Sbox}} a^2 \xrightarrow{\text{rotation}} \cdots b^{r-1} \xrightarrow{\text{Sbox}} a^r.$$

### 5.3 Finding the Longest Division Properties

The authors of the KNOT adopted the MILP-based search strategy to analyze KNOT's integral properties. They found $17, 17, 19$-round integral distinguishers for the sizes $b = 256$, 384 and 512, respectively. All of the distinguishers have data complexity of $2^{b-1}$ and in all cases, they found one balanced bit at the $(3, 0)$ position. To the best of our knowledge, this is the only available result on the KNOT in the context of the integral attacks. Also, the authors did not provide any time requirements for these findings.

We obtained several new results on the KNOT-256, KNOT-384 and KNOT-512. All of our experiments are conducted on the following 64-bit Linux platform: Intel Core i7-3520M CPU @ 2.90GHz, 8.00G RAM. We used the model proposed in Section 5.2 and solved it using SAT/SMT and CP based tools. All the source codes are avaiable in public domain at `https://github.com/ShibamCrS/AutomaticSearchforBBDP`.

First, for all versions of KNOT, we considered the initial division property $in_{(k,\ell)}$ (all coordinates 1 except for the $(k, \ell)$-th position) for each $0 \leq k < 4$ and $0 \leq \ell < \frac{b}{4}$. We are getting at least two balanced bits on the output states after 17-rounds KNOT-256 and KNOT-384 and 19-round KNOT-512. From this result, we can say that the *sufficient index set* $\mathbb{S}$ defined in Section 4.4 contains all of the $b$ bits positions for $b = 256, 384, 512$. Secondly, if we set the input variables as

$$a^0 = in_{(3,0)} = \begin{cases} 0, \text{ if } (k, \ell) = (3, 0) \\ 1, \text{ otherwise} \end{cases}$$

then we can get many balanced bits after 17 rounds for the KNOT-256 and KNOT-384 and after 19 rounds for the KNOT-512, which spread over all the four rows, given in Appendix A. These results outperform the previous result in [33], where the authors found only one balanced bit in each version. The time requirements with the number of balanced positions are given in Table 5.

| KNOT-b | #Rounds | #Data | #Balanced-bits | SAT | SAT/CNF | CP | CP/CNF |
|---|---|---|---|---|---|---|---|
| KNOT-256 | 17 | $2^{255}$ | 89 | 13 hr | 19 min | >15 hr | 12 min |
| KNOT-384 | 17 | $2^{383}$ | 140 | >15 hr | 45 min | >15 hr | 17 min |
| KNOT-512 | 19 | $2^{511}$ | 269 | >15 hr | 2.1 hr | >15 hr | 70 min |

Table 5: Distinguishers for the KNOT with input property $a = in_{(3,0)}$

### 5.4 More Efficient Distinguisher

From the previous result, as the sufficient index set $\mathbb{S}$ is huge, here we cannot use the whole set on the data complexity reduction algorithm. Instead we used a subset $\mathbb{S}'$ of $\mathbb{S}$. If we consider only the first column, i.e., we set constant at positions $\mathbb{S}' = \{(0,0),(1,0),(2,0),(3,0)\}$.

**KNOT-256:** For the KNOT-256 we calculated the balanced bits for each constant positions in $\{(0,0),(1,0),(2,0),(3,0)\}$. Then we applied Algorithm 2 with $\mathbb{S}' = \{(0,0),(1,0),(2,0),(3,0)\}$ on 17 round KNOT-256. It gives the output $t = 2$ and $Z = \{(0,0),(1,0)\}$. Now we can get balanced positions with input $Z$. We get two balanced positions $(2,16)$ and $(2,56)$ after 17 rounds. By a similar approach we can get several distinguishers, a few of them are given in Table 12. In fact for KNOT-256 we can get two balanced positions in the third row by setting constant in the first two elements of each column with data complexity $2^{254}$.

**KNOT-384:** For the KNOT-384 we took $\mathbb{S}' = \{(0,0),(1,0),(2,0),(3,0)\}$ and we apply Algorithm 2 for 17 rounds. It gives the outputs $t = 4$ and $Z = \{(0,0),(1,0),(2,0),(3,0)\}$. We get in total 19 many balanced bits from $Z$, spread over the last two rows given in Table 6.

| Constant positions | (0,0),(1,0),(2,0),(3,0) |
|---|---|
| Balanced positions at the third row | [24, 31, 38, 39, 40, 77, 78, 79, 80, 86, 87, 89] |
| Balanced positions at the fourth row | [40, 77, 79, 80, 86, 87, 89] |

Table 6: Distinguisher on the 17-round KNOT-384 with the data complexity $2^{380}$

**KNOT-512** For the KNOT-512 also we found a lower complexity distinguisher on 19 rounds. Here also we took $\mathbb{S}' = \{(0,0),(1,0),(2,0),(3,0)\}$ and we apply Algorithm 2 which outputs $t = 4$ and $Z = \{(0,0),(1,0),(2,0),(3,0)\}$. We get in total 139 many balanced bits from $Z$, spread over all the four rows given in Table 7.

| | |
|---|---|
| *constant* | (0,0),(1,0),(2,0),(3,0) |
| 1st row | {0, 7, 8, 15, 16, 30, 31, 32, 40, 47, 55, 56, 64, 79, 80, 88, 112, 127} |
| 2nd row | {0, 7, 8, 14, 15, 16, 23, 30, 31, 32, 39, 40, 47, 48, 55, 56, 63, 64, 79, 80, 87, 88, 95, 96, 103, 111, 112, 119, 127} |
| 3rd row | {0, 5, 6, 7, 8, 14, 15, 16, 22, 23, 24, 29, 30, 31, 32, 38, 39, 40, 41, 47, 48, 55, 56, 57, 63, 64, 65, 70, 71, 79}, {80, 81, 86, 87, 88, 89, 94, 95, 96, 102, 103, 104, 109, 110, 111, 112, 118, 119, 120, 126, 127} |
| 4th row | {0, 5, 6, 7, 8, 14, 15, 16, 23, 29, 30, 31, 32, 39, 40, 41, 47, 48, 55, 56, 57, 63, 64, 65, 71, 79} {80, 81, 87, 88, 89, 95, 96, 103, 104, 109, 111, 112, 118, 119, 127} |

Table 7: Distinguisher on the 19-round KNOT-512 with the data complexity $2^{508}$

## 6 Other Results

To show the strength of our approach, we considered some well-known ciphers RECTANGLE [38], GIFT [2], PRESENT [4]. While we obtained results that do not improve the previously known results, they demonstrate the clear advantage of using CNF models. The difference between the time requirements of two methods is given in the Table 8. We also implement our model to NIST lightweight candidate Ascon [10] and we obtain a distinguisher with data complexity $2^{12}$. This result improves the previous result in [11], where data complexity was $2^{16}$. The 12 active bit positions in the Ascon state matrix are $(0,0)$,$(1,0)$,$(2,0)$,$(3,0)$,$(4,0)$, $(0,1)$,$(1,1)$,$(2,1)$,$(3,1)$,$(4,1)$,$(0,2)$,$(1,2)$.

| Primitives | #Round | #Data | #Balanced Bits | SAT | SAT(CNF) | CP | CP(CNF) |
|---|---|---|---|---|---|---|---|
| Ascon-320 | 5 | $2^{12}$ | 2 | >15 hr | 20 min | >15 hr | 10 min |
| RECTANGLE-64 | 9 | $2^{60}$ | 16 | 70 s | 35 s | 50 s | 40 s |
| RECTANGLE-64 | 10 | | No Distinguisher* | 1.48 hr | 41 min | 1.44 hr | 39 min |
| PRESENT-64 | 9 | $2^{60}$ | 1 | 76 s | 21 s | 17 min | 45 s |
| PRESENT-64 | 10 | | No Distinguisher* | 1.5 hr | 24 min | 1.7 hr | 38 min |
| GIFT-64 | 9 | $2^{61}$ | 5 | 103 s | 60 s | 58 min | 62 s |
| GIFT-64 | 10 | | No Distinguisher* | 1.49 hr | 42 min | 1.19 hr | 43 min |

Table 8: Division property results
*Time required for exhaustive search with all possible $in_i$ and $out_j$

## 7 Extended Integral Attack

In a very recent work in [17], Lambin et al. proposed a new way to extend the integral distinguisher. The main motivation of these types of extensions is that the division properties are not linearly invariant. If we consider a linear map $L$ and an Sbox $S$ then the division properties of $L \circ S$ and $S \circ L$ may be different from those of $S$ and consequently, the division trail table may differ. Thus by choosing a proper $L$, we may get new distinguishers which were impossible when modeling $S$ alone. Thus, for a given r-round primitive $E_r$, the authors of [17]

proposed to consider $L_{out} \circ E_r \circ L_{in}$ instead of $E_r$ where both $L_{in}$ and $L_{out}$ are linear mappings. This way, we may get some distinguisher on $L_{out} \circ E_r \circ L_{in}$ which is not possible on $E_r$. Lambinet al. also described some ideas to choose proper $L_{in}$ and $L_{out}$. Note that the search space of these linear combinations is huge and this space needs to be reduced. On this direction, Lambinet al.'s first proposed the following Proposition 4.

**Proposition 4.** *Let $S$ be an invertible $m$-bit Sbox and $P$ be an $m$-bit permutation. Let $S_1 = S \circ P$ and $S_2 = P \circ S$ and $k \xrightarrow{S} k'$ be any valid division property propagation through $S$. Then both of the propagations*

$$P^{-1}(k) \xrightarrow{S_1} k' \text{ and } k \xrightarrow{S_2} P(k')$$

*are also valid.*

Proof of the above proposition is obvious. As we consider the bit-based division property, bit-permutation just permutes the division property vector. But this plays a crucial role in reducing the search space.

We focus on linear mapping which are block diagonals, each block corresponds to an $m$-bit Sbox in the Sbox layer (an $m \times m$ matrix). So we can write $L_{in} = (L_{in}^0, L_{in}^1, \cdots, L_{in}^{s-1})$ and $L_{out} = (L_{out}^0, L_{out}^1, \cdots, L_{out}^{s-1})$ where each $L_{in}^j$ and $L_{out}^j$ is an $m \times m$ matrix. To find the $i$-th block $L_{in}^i$ Lambinet al. considered the following permutation equivalence classes

$$\mathcal{E}_{in}(L) = \{L' \in GL_m(\mathbb{F}_2) | \exists \text{ permutation } P \text{ s.t. } L' = L \circ P\}.$$

Similarly to find $L_{out}^i$ we have

$$\mathcal{E}_{out}(L) = \{L' \in GL_m(\mathbb{F}_2) | \exists \text{ permutation } P \text{ s.t. } L' = P \circ L\}.$$

The number of these classes can be deduced from $\frac{\prod_{i=0}^{m-1} 2^m - 2^i}{m!}$. This number is much lower than the total number of $m \times m$ invertible matrices, which is $\prod_{i=0}^{m-1} 2^m - 2^i$. Now if we can find each $\mathcal{E}_{in}(L)$ and $\mathcal{E}_{out}(L)$, we consider only **one** linear map from each of the classes. For example, if $m = 4$, there are in total 840 of such classes.

### 7.1 Further Reduction of the Search Space for $L_{out}$

We now discuss the method to find an optimal number of $L_{out}$. It was suggested in [9] that finding proper linear combinations of the output bits is enough to find an integral distinguisher instead of a linear map. Indeed, here our main motivation is that multiplying the output vectors with a matrix is the same as taking linear combinations. If we cannot get any balanced bit by linear combinations, we cannot find by matrix multiplication. Thus, we check all possible nonzero linear combinations. This reduces the search space from $\frac{\prod_{i=0}^{m-1} 2^m - 2^i}{m!}$ to $2^m - 1$. If $m = 4$, it reduced from 840 to 15. However, checking for all those 15 linear

combinations is also a huge task. We now discuss how to further reduce this formally.

Let us consider an $r$-round primitive $E_r$ as described at the starting of this section. Here we want to take linear combinations of each Sbox output, not the whole state. For an $m \times m$ Sbox, there are $2^m - 1$ linear combinations. If the Sbox $S : \mathbb{F}_2^m \to \mathbb{F}_2^m$ maps $(x_{m-1}, \cdots, x_0)$ to $(y_{m-1}, \cdots, y_0)$ then any linear combination of $(y_{m-1}, \cdots, y_0)$ is also a Boolean function. Its ANF can be determined from the ANF of $y_i$'s. If $c$ is an integer with binary representation $(c_{m-1}, \cdots, c_0)$ then $P_c$ is the linear combination corresponding to $c$, i.e.,

$$P_c = (c_{m-1}, \cdots, c_0) \cdot (y_{m-1}, \cdots, y_0) = \sum_{i=0}^{m-1} c_i y_i.$$

So there are $2^m - 1$ polynomials $P_1, P_2, \cdots, P_{2^m-1}$ corresponding to each Sbox. Each of these can be written as $P_c(x_{m-1}, \cdots, x_0) = \bigoplus_{u \in \mathbb{F}_2^m} a_u^{P_c} \pi_u(x)$. A high-level overview of our process is as follows:

1. For a given $r$ round primitive first consider the propagation of some initial property $k_0$ through
$$R_{r-2} \circ \cdots \circ R_1 \circ R_0.$$
and the propagation forms the following chain:
$$k_0 \xrightarrow{R_0} \mathbb{K}^1 \xrightarrow{R_1} \mathbb{K}^2 \xrightarrow{R_2} \cdots \xrightarrow{R_{r-2}} \mathbb{K}^{r-1}.$$
2. After this each vector $k^{r-1} \in \mathbb{K}^{r-1}$ propagate through the last Sbox layer $S^{r-1} = (S_0^{r-1}, \cdots, S_{s-1}^{r-1})$.
3. At this stage let us consider the state after $R_{r-2}$. let
$$x = (x_{m-1}^0, \cdots, x_0^0 | x_{m-1}^1, \cdots, x_0^1 | \cdots | x_{m-1}^{s-1}, \cdots, x_0^{s-1})$$
be the state after $R_{r-2}$ where the block $(x_{m-1}^i, \cdots, x_0^i)$ is the input to the $i$-th Sbox and let $S^{r-1}(x) = y$. We can write the following equations
$$\begin{gathered} S_0^{r-1}(x_{m-1}^0, \cdots, x_0^0) = (y_{m-1}^0, \cdots, y_0^0) \\ \vdots \\ S_i^{r-1}(x_{m-1}^i, \cdots, x_0^i) = (y_{m-1}^i, \cdots, y_0^i) \\ \vdots \\ S_{s-1}^{r-1}(x_{m-1}^{s-1}, \cdots, x_0^{s-1}) = (y_{m-1}^{s-1}, \cdots, y_0^{s-1}) \end{gathered}$$
4. Now if we consider the linear combinations of the output from any Sbox, say the $i$-th Sbox, then the ANF of any linear combination will be a polynomial in $(x_{m-1}^i, \cdots, x_0^i)$. To check any linear combination is balanced or not we have to check the polynomial corresponding to this is even or not.

### 7.2 Checking Polynomials

We now discuss the process to check a polynomial is even polynomial or not. To do so, we check each monomial present in the polynomial by our SAT/CP model. Suppose that we want to check a monomial $x^i_{j_0}x^i_{j_1}\cdots x^i_{j_k}$, where $(x^i_{m-1},\cdots,x^i_0)$ are input to the $i$-th Sbox of the last round. To check the monomial $x^i_{j_0}x^i_{j_1}\cdots x^i_{j_k}$ we set the output property as $out_{j_0,j_1,\cdots,j_k}$, where $out_{j_0,j_1,\cdots,j_k}$ is a vector with all zero except for positions $j_0,\cdots,j_k$. If we get the system unsatisfiable for some initial property, then we can say that the monomial is even. Consequently, if all the monomials in the ANF are even, then the polynomial is even polynomial and there is an integral distinguisher. One important thing is that we solve the model for $r-1$ rounds to decide balanced bit after $r$ rounds.

Let us consider an Sbox of size $m$. There are in total $2^m-1$ linear combinations polynomial $P_1,\cdots,P_{2^m-1}$. Each $P_i$ corresponds to the linear combination obtained from the binary representation of $i$. The 15 linear combinations polynomial corresponding to KNOT sbox 1 are given in Table 9.

| $c$ | $P_c$ |
|---|---|
| 0001 | $x_3x_1x_0+x_1x_0+x_2x_0+x_3x_1+x_3x_2+x_2+x_3$ |
| 0010 | $x_3x_0+x_3x_2x_1+x_1+x_3x_2+x_2$ |
| 0011 | $x_3x_1x_0+x_1x_0+x_2x_0+x_3x_0+x_3x_2x_1+x_3x_1+x_1+x_3$ |
| 0100 | $x_0+x_2x_1+x_1+x_2+x_3+1$ |
| 0101 | $x_3x_1x_0+x_1x_0+x_2x_0+x_0+x_2x_1+x_3x_1+x_1+x_3x_2+1$ |
| 0110 | $x_3x_0+x_0+x_3x_2x_1+x_2x_1+x_3x_2+x_3+1$ |
| 0111 | $x_3x_1x_0+x_1x_0+x_2x_0+x_3x_0+x_0+x_3x_2x_1+x_2x_1+x_3x_1+x_2+1$ |
| 1000 | $x_1x_0+x_1+x_2+x_3$ |
| 1001 | $x_3x_1x_0+x_2x_0+x_3x_1+x_1+x_3x_2$ |
| 1010 | $x_1x_0+x_3x_0+x_3x_2x_1+x_3x_2+x_3$ |
| 1011 | $x_3x_1x_0+x_2x_0+x_3x_0+x_3x_2x_1+x_3x_1+x_2$ |
| 1100 | $x_1x_0+x_0+x_2x_1+1$ |
| 1101 | $x_3x_1x_0+x_2x_0+x_0+x_2x_1+x_3x_1+x_3x_2+x_2+x_3+1$ |
| 1110 | $x_1x_0+x_3x_0+x_0+x_3x_2x_1+x_2x_1+x_1+x_3x_2+x_2+1$ |
| 1111 | $x_3x_1x_0+x_2x_0+x_3x_0+x_0+x_3x_2x_1+x_2x_1+x_3x_1+x_1+x_3+1$ |

Table 9: Linear combinations from outputs of the KNOT Sbox

However, here no need to check each of the $2^m-1$ polynomials. We identify a subset of polynomials which are sufficient, i.e., if these polynomials are not even, then there is no division property. We define an order ($\sqsubseteq$) among the polynomials such that if $P_i \sqsubseteq P_j$ then we check if the polynomial $P_i$ is even polynomial or not. Suppose $P_i$ is not even, then no need to check for $P_j$. If $P_i$ is even, we have a distinguisher and for more distinguishers, we can check $P_j$'s such that $P_i \sqsubseteq P_j$. The definition of this order is as follows.

**Definition 6.** *Let us consider two polynomials $P$ and $Q$ from the ring*

$$\mathbb{F}_2[x_0,x_1,.....,x_{m-1}]/(x_0^2+x_0,x_1^2+x_1,.....,x_{n-1}^2+x_{m-1}).$$

*We say that $Q$ is Dependent on $P$, denoted by $P \sqsubseteq Q$ if any monomial (term) in $P$ divides at least one monomial (term) of $Q$.*

**Proposition 5.** *If we have two linear combination polynomials $P$ and $Q$ with $P \sqsubseteq Q$ and we can find that $P$ is not even, then $Q$ is not even.*

*Proof.* If $P$ is not even then according to definition 4 of the even polynomial we can get some monomials in $P$ which is not even, i.e., there is some $u \in \mathbb{F}_2^m$ such that $a_u^P = 1$ and $\bigoplus_{x \in \mathbb{F}_2^m} \pi_u(x)$ is unknown. As $P \sqsubseteq Q$, then that unknown term must divide some term of $Q$ and that term of $Q$ must be unknown according to the definition of division property. So $Q$ can not be an even polynomial.

For example there are 15 polynomials for the KNOT Sbox. Let us consider $P_4$ and $P_2$, where terms of $P_4$ are $\{x_0, x_2x_1, x_1, x_2, x_3, 1\}$ and terms of $P_2$ are $\{x_3x_0, x_3x_2x_1, x_1, x_3x_2, x_2\}$. We can observe that $x_0, x_3$ divides $x_3x_0$ and $x_1, x_2, x_1x_2$ divides $x_3x_2x_1$. So we can say $P_4 \sqsubseteq P_2$.

**Minimal Number of Polynomials:** Finally, we want to decide how many polynomials we need to check. According to the dependency relation defined above, we form a few clusters of polynomials so that all the polynomials present in a cluster depend on a single polynomial and then by checking that single polynomial, we can decide about all other polynomials in the cluster.

To form this cluster we consider a dependency graph $G = (V, E)$, where each vertex $v_P$ in the vertex set $V$ corresponds to a polynomial $P$ and there is a directed edge from $v_P$ to $v_Q$ if and only if $P \sqsubseteq Q$. Then we choose a starting vertex $v_P$ and try to construct a trail. It is clear from transitivity that all the vertices $v_Q$ fall in this trail must satisfy $P \sqsubseteq Q$. Also, We have to find a trail of size as large as possible. The reason is that if the size of the trail is large, we can check a large number of polynomials at once. To do this, we use Depth First Search (DFS) from each vertex one by one, which can find a trail starting from that vertex.

| *Length* | *c* |
|---|---|
| 11 | [4, 2, 3, 7, 11, 15, 6, 10, 14, 5, 13] |
| 11 | [8, 1, 3, 7, 11, 15, 5, 13, 9, 10, 14] |
| 9 | [12, 3, 7, 11, 15, 5, 13, 10, 14] |
| 8 | [1, 3, 7, 11, 15, 5, 13, 9] |
| 8 | [2, 3, 7, 11, 15, 6, 10, 14] |
| 8 | [6, 2, 3, 7, 11, 15, 10, 14] |
| 8 | [9, 1, 3, 7, 11, 15, 5, 13] |
| 6 | [5, 3, 7, 11, 15, 13] |
| 6 | [10, 3, 7, 11, 15, 14] |
| 6 | [13, 3, 7, 11, 15, 5] |
| 6 | [14, 3, 7, 11, 15, 10] |
| 4 | [3, 7, 11, 15] |
| 4 | [7, 3, 11, 15] |
| 4 | [11, 3, 7, 15] |
| 4 | [15, 3, 7, 11] |

Table 10: Trails from each vertex

### 7.3 Application on the KNOT

The trails for the KNOT Sbox are given in Table 10 where the first element is the starting linear combination $c$. From Table 10 we can see that one of the maximum length trails can be formed from the polynomial $P_4$ (i.e, from the vertex $v_{P_4}$). First we remove all the vertices that are reachable from $v_{P_4}$, which are $\{v_{P_4}, v_{P_2}, v_{P_3}, v_{P_7}, v_{P_{11}}, v_{P_{15}}, v_{P_6}, v_{P_{10}}, v_{P_{14}}, v_{P_5}, v_{P_{13}}\}$ and then we move to the next cluster. As the next cluster starts with $v_{P_8}$ and $v_{P_8}$ does not belong to the previously removed cluster we can start a search from $v_{P_8}$ and construct a cluster with remaining vertices. If $v_{P_8}$ was already removed, just consider starting vertex of the next cluster on the list and so on. Finally, we have found the following trails given in Table 11.

| *starting* | *trails* |
|---|---|
| 4 | [4, 2, 3, 7, 11, 15, 6, 10, 14, 5, 13] |
| 8 | [8, 1, 9] |
| 12 | [12] |

Table 11: Final trails of considerations

According to this result first we need to check the following three polynomials given in Equation 2. If this set of polynomials are not even then we need not check further.

$$\begin{aligned} y_2(x) = x_0 + x_2x_1 + x_1 + x_2 + x_3 + 1, \text{linear combination for } 4 = (0,1,0,0) \\ y_3(x) = x_1x_0 + x_1 + x_2 + x_3, \text{linear combination for } 8 = (1,0,0,0) \\ y_3(x) + y_2(x) = x_1x_0 + x_0 + x_2x_1 + 1, \text{linear combination for } 12 = (1,1,0,0) \end{aligned} \tag{2}$$

Also, we can see that there are only 6 different monomials $\{x_0, x_1, x_2, x_3, x_0x_1, x_1x_2\}$ in all those three polynomials that we need to check. Among these 6, we start from $x_0x_1$ and $x_1x_2$ because none of the polynomials are even if these two are not even.

Let $b^{r-1}$ be the input variables to the $r$th round Sbox $S^{r-1}$ where $a^{r-1} \xrightarrow{\text{rotation}} b^{r-1}$. Let us take linear combination of the output of the first Sbox. Then to check $x_0x_1$ is even or not we set 1 to the positions $b^{r-1}_{0,0}$ and $b^{r-1}_{1,0}$ and to check $x_1x_2$ we set 1 to $b^{r-1}_{1,0}$ and $b^{r-1}_{2,0}$ as follows:

$$b^{r-1} = \begin{bmatrix} 1\,0\,0\cdots 0 \\ 1\,0\,0\cdots 0 \\ 0\,0\,0\cdots 0 \\ 0\,0\,0\cdots 0 \end{bmatrix} \text{ and } b^{r-1} = \begin{bmatrix} 0\,0\,0\cdots 0 \\ 1\,0\,0\cdots 0 \\ 1\,0\,0\cdots 0 \\ 0\,0\,0\cdots 0 \end{bmatrix}.$$

Finally, we solve the model in both cases with all possible input vectors $in_{(k,\ell)}$ of weight $b-1$.

**Result on the KNOT** For the KNOT-256 and KNOT-384, we searched for all initial vectors for all Sboxes after 17 rounds and we did not get any even monomial. The implication of this result is two-fold. First of all, this proves that we can not extend the integral distinguisher to 18 rounds by this method. Also, with this, we can get the most important result here that the 18-round KNOT-256 and KNOT-384 has no integral distinguisher using conventional division property. This follows from the fact that $y_0, y_1, y_2, y_3$ are also belong to the set of all linear combinations as we have $P_1 = y_0, P_2 = y_1, P_4 = y_2$ and $P_8 = y_3$. However, note that here for each column, we need to solve a 17-round SAT/CP model twice per Sbox, whereas in the usual method, we need to solve the 18-round model four times per Sbox. In other words, not only we proved a strong result about KNOT-256 and KNOT-384, we did so more efficiently. Our CP model takes 20 hours for KNOT-256 and 50 hours for KNOT-384 to complete this search, which is significantly more efficient. The usual method did not complete the search even after several days. We terminated the process after three days.

**Adding $L_{in}$** We also tried to add a linear layer $L_{in}$ at the input. But we could not find any useful information for any versions of the KNOT and Ascon.

## 8 Conclusion

In this paper we provided several new distinguishers for the KNOT permutation and Ascon using the SAT and CP-based automatic tools. To model the division trail table of an Sbox we used the Quine-McCluskey method, which gives the minimal size CNF. We provided a compact algorithm to find the optimal distinguishers. Our model is much more efficient and accurate than the previous result on KNOT and Ascon. Finally, we provided a way to get the optimal number of linear combinations for extended integral attack and using this, we have shown that 18-round KNOT-256 and KNOT-384 have no integral distinguisher.

## References

1. Abdelkhalek, A., Sasaki, Y., Todo, Y., Tolba, M.F., Youssef, A.: MILP modeling for (large) S-boxes to optimize probability of differential characteristics. IACR Trans. Symmetric Cryptol. **2017**, 99–129 (2017)
2. Banik, S., Pandey, S.K., Peyrin, T., Sasaki, Y., Sim, S.M., Todo, Y.: GIFT: A small present. In: Proceedings of CHES 2017. LNCS, vol. 10529, pp. 321–345. Springer (2017)
3. Barrett, C., Tinelli, C.: CVC3. In: Proceedings of CAV'07. LNCS, vol. 4590, p. 298–302. Springer (2007)
4. Bogdanov, A., Knudsen, L.R., Leander, G., Paar, C., Poschmann, A., Robshaw, M.J.B., Seurin, Y., Vikkelsoe, C.: PRESENT: An ultra-lightweight block cipher. In: Proceedings of the CHES 2007. LNCS, vol. 4727, pp. 450–466. Springer (2007)
5. Chu, G.: Improving combinatorial optimization: Extended abstract. In: Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence. p. 3116–3120. IJCAI '13, AAAI Press (2013)

6. Cook, S.A.: The complexity of theorem-proving procedures. In: Proceedings of STOC '71. p. 151–158. ACM, New York, USA (1971)
7. Daemen, J., Knudsen, L., Rijmen, V.: The block cipher SQUARE. In: Proceedings of FSE 1997. LNCS, vol. 1267, pp. 149–165. Springer (1997)
8. Daemen, J., Rijmen, V.: AES and the wide trail design strategy. In: Advances in Cryptology – EUROCRYPT 2002. LNCS, vol. 2332, pp. 108–109. Springer (2002)
9. Derbez, P., Fouque, P.A.: Increasing precision of division property. IACR Cryptol. ePrint Arch. **2021**, 22 (2020)
10. Dobraunig, C., Eichlseder, M., Mendel, F., Schläffer, M.: Ascon v1.2 - lightweight authenticated encryption and hashing. Journal of Cryptology (2020)
11. Eskandari, Z., Kidmose, A., Kölbl, S., Tiessen, T.: Finding integral distinguishers with ease. In: Proceedings of SAC 2018. LNCS, vol. 11349, pp. 115–138. Springer (2019)
12. Ferguson, N., Kelsey, J., Lucks, S., Schneier, B., Stay, M., Wagner, D., Whiting, D.: Improved cryptanalysis of Rijndael. In: Proceedings of FSE 2000. LNCS, vol. 1978, pp. 213–230. Springer
13. Ganesh, V., Dill, D.L.: A decision procedure for bit-vectors and arrays. In: Proceedings of CAV 2007. LNCS, vol. 4590, pp. 519–531. Springer (2007)
14. Hu, K., Wang, Q., Wang, M.: Finding bit-based division property for ciphers with complex linear layers. IACR Transactions on Symmetric Cryptology **2020**(1), 396–424 (2020)
15. Huang, J.: Programing implementation of the quine-mccluskey method for minimization of boolean expression (2014)
16. Knudsen, L., Wagner, D.: Integral cryptanalysis. In: Proceedings of FSE 2002. LNCS, vol. 2365, pp. 112–127. Springer
17. Lambin, B., Derbez, P., Fouque, P.A.: Linearly equivalent S-boxes and the division property. Designs, Codes and Cryptography **88** (2020)
18. Li, Y., Wu, W., Zhang, L.: Improved integral attacks on reduced-round CLEFIA block cipher. In: Proceedings of WISA 2011. LNCS, vol. 7115, pp. 28–39. Springer (2012)
19. Lucks, S.: The saturation attack — a bait for Twofish. In: Proceedings of FSE 2001. LNCS, vol. 2355, pp. 1–15. Springer (2002)
20. McCluskey, E.J.: Minimization of boolean functions. The Bell System Technical Journal **35**(6), 1417–1444 (1956)
21. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: MiniZinc: towards a standard cp modelling language. In: Bessière, C. (ed.) Proceedings of CP 2007. LNCS, vol. 4741, pp. 529–543. Springer (2007)
22. Quine, W.V.: The problem of simplifying truth functions. The American Mathematical Monthly **59**(8), 521–531 (1952), `http://www.jstor.org/stable/2308219`
23. Quine, W.V.: A way to simplify truth functions. The American Mathematical Monthly **62**(9), 627–631 (1955), `http://www.jstor.org/stable/2307285`
24. Soos, M., Nohl, K., Castelluccia, C.: Extending SAT solvers to cryptographic problems. In: Proceedings of SAT 2009. LNCS, vol. 5584, pp. 244–257. Springer (2009)
25. Sun, L., Wang, W., Liu, R., Wang, M.: MILP-aided bit-based division property for arx ciphers. Science China Information Sciences **61** (2018)
26. Sun, L., Wang, W., Wang, M.: Automatic search of bit-based division property for ARX ciphers and word-based division property. In: Advances in Cryptology – ASIACRYPT 2017. LNCS, vol. 10624, pp. 128–157. Springer (2017)
27. Sun, L., Wang, W., Wang, M.: MILP-aided bit-based division property for primitives with non-bit-permutation linear layers. IET Information Security **14** (2019)

28. Sun, S., Gerault, D., Lafourcade, P., Yang, Q., Todo, Y., Qiao, K., Hu, L.: Analysis of aes, skinny, and others with constraint programming. IACR Transactions on Symmetric Cryptology **2017**(1), 281–306 (2017)
29. Technology, N.: Report on Lightweight Cryptography: NiSTIR 8114. CreateSpace Independent Publishing Platform (2017)
30. Todo, Y.: Structural evaluation by generalized integral property. In: Advances in Cryptology – EUROCRYPT 2015. LNCS, vol. 9056, pp. 287–314. Springer (2015)
31. Todo, Y.: Integral cryptanalysis on full MISTY1. Journal of Cryptology **30**, 920–959 (2016)
32. Todo, Y., Morii, M.: Bit-based division property and application to SIMON family. In: Fast Software Encryption. LNCS, vol. 9783, pp. 357–377. Springer (2016)
33. W. Zhang, T. Ding, B. Yang, Z. Bao, Z. Xiang, F. Ji, and X. Zhao: KNOT: Algorithm specifications and supporting document. IACR Cryptol. ePrint Arch. (2020)
34. Wang, Q., Grassi, L., Rechberger, C.: Zero-Sum Partitions of PHOTON Permutations, LNCS, vol. 10808, pp. 279–299. Springer (2018)
35. Wu, W., Zhang, L.: LBlock: a lightweight block cipher. In: Proceedings of ACNS 2011. LNCS, vol. 6715, pp. 327–344. Springer (2011)
36. 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: Advances in Cryptology – ASIACRYPT 2016. LNCS, vol. 10031, pp. 648–678 (2016)
37. Z'aba, M.R., Raddum, H., Henricksen, M., Dawson, E.: Bit-pattern based integral attack. In: Proceedings of FSE 2008. LNCS, vol. 5086, pp. 363–381. Springer (2008)
38. Zhang, W., Bao, Z., Lin, D., Rijmen, V., Yang, B., Verbauwhede, I.: RECTANGLE: a bit-slice lightweight block cipher suitable for multiple platforms. Science China Information Sciences **58** (2015)

# A Results on KNOT

## A.1 Results on KNOT-256

Results on the 17-round KNOT-256 with $\mathbb{S}' = \{(0,0),(1,0),(2,0),(3,0)\}$ and obtained the following results for $OUT[(k,\ell)]$.

- $a^0 = in_{(0,0)}$ gives the following balanced bits:
  Row 0 ⇒ [ ]
  Row 1 ⇒ [ ]
  Row 2 ⇒ [16, 56]
  Row 3 ⇒ [ ]
- $a^0 = in_{(1,0)}$ gives the following balanced bits:
  Row 0 ⇒ [0, 8, 16, 24]
  Row 1 ⇒ [0, 1, 8, 9, 16, 17, 24, 25, 32, 33, 40, 48, 56]
  Row 2 ⇒ [0, 1, 8, 9, 16, 17, 24, 25, 32, 33, 40, 41, 48, 49, 56, 57]
  Row 3 ⇒ [0, 1, 8, 9, 16, 17, 24, 25, 32, 33, 40, 48, 49, 56, 57]
- $a^0 = in_{(2,0)}$ and gives the following balanced bits:
  Row 0 ⇒ [ ]
  Row 1 ⇒ [ ]
  Row 2 ⇒ [8, 16, 24, 32, 40, 48, 56]
  Row 3 ⇒ [ ]
- $a^0 = in_{(3,0)}$ gives the following balanced bits:
  Row 0 ⇒ [0, 1, 8, 9, 10, 16, 17, 24, 25, 32, 33, 40, 41, 48, 49, 50, 56, 57, 58]
  Row 1 ⇒ [0, 1, 8, 9, 10, 16, 17, 18, 24, 25, 26, 32, 33, 34, 40, 41, 48, 49, 50, 56, 57, 58]
  Row 2 ⇒ [0, 1, 2, 8, 9, 10, 16, 17, 18, 24, 25, 26, 32, 33, 34, 40, 41, 42, 48, 49, 50, 56, 57, 58]
  Row 3 ⇒ [0, 1, 2, 8, 9, 10, 16, 17, 18, 24, 25, 26, 32, 33, 34, 40, 41, 42, 48, 49, 50, 56, 57, 58]

| Constant positions | Balanced positions at third row |
|---|---|
| $(0,0),(1,0)$ | [16, 56] |
| $(0,1),(1,1)$ | [17, 57] |
| $(0,2),(1,2)$ | [18, 58] |
| $(0,3),(1,3)$ | [19, 59] |
| $(0,4),(1,4)$ | [20, 60] |
| $(0,5),(1,5)$ | [21, 61] |
| $(0,6),(1,6)$ | [22, 62] |
| $(0,7),(1,7)$ | [23, 63] |
| $(0,8),(1,8)$ | [0, 24] |
| $(0,9),(1,9)$ | [1, 25] |
| $(0,10),(1,10)$ | [2, 26] |
| $(0,11),(1,11)$ | [3, 27] |
| $(0,12),(1,12)$ | [4, 28] |
| $(0,13),(1,13)$ | [5, 29] |
| $(0,14),(1,14)$ | [6, 30] |
| $(0,15),(1,15)$ | [7, 31] |

Table 12: Distinguishers on the 17-round KNOT-256 with the data complexity $2^{254}$

### A.2 Results on KNOT-384

Results on the 17-round KNOT-384 with $\mathbb{S}' = \{(0,0),(1,0),(2,0),(3,0)\}$ and obtained the following results for $OUT[(k,\ell)]$.

- $a^0 = in_{(0,0)}$ gives the following balanced bits:
  Row 0 ⇒ [ ]
  Row 1 ⇒ [79]
  Row 2 ⇒ [24, 25, 30, 31, 32, 37, 38, 39, 40, 42, 45, 50, 71, 76, 77, 78, 79, 80, 81, 85, 86, 87, 88, 89, 91, 92, 94]
  Row 3 ⇒ [1, 30, 31, 32, 38, 39, 40, 41, 77, 78, 79, 80, 81, 86, 87, 88, 89, 90]
- $a^0 = in_{(1,0)}$ gives the following balanced bits:
  Row 0 ⇒ [32, 38, 40, 77, 79, 80, 81, 87]
  Row 1 ⇒ [30, 31, 32, 38, 40, 77, 78, 79, 80, 81, 82, 87]
  Row 2 ⇒ [0, 1, 2, 3, 7, 9, 10, 14, 17, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 54, 55, 57, 64, 69, 70, 71, 72, 73, 75, 76, 77, 78, 79, 80, 81, 82, 84, 85, 86, 87, 88, 89, 90, 91, 92, 94, 95]
  Row 3 ⇒ [0, 1, 2, 9, 23, 24, 25, 28, 29, 30, 31, 32, 33, 35, 36, 37, 38, 39, 40, 41, 42, 43, 46, 47, 48, 49, 50, 54, 55, 56, 57, 71, 72, 73, 77, 78, 79, 80, 81, 82, 83, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95]
- $a^0 = in_{(2,0)}$ gives the following balanced bits:
  Row 0 ⇒ [ ]
  Row 1 ⇒ [79]
  Row 2 ⇒ [1, 30, 31, 32, 38, 39, 40, 41, 42, 48, 50, 77, 78, 79, 80, 81, 86, 87, 88, 89, 90]
  Row 3 ⇒ [10, 24, 25, 30, 31, 32, 36, 37, 38, 39, 40, 42, 45, 46, 50, 71, 76, 77, 78, 79, 80, 81, 82, 85, 86, 87, 88, 89, 90, 91, 92, 94]
- $a^0 = in_{(3,0)}$ gives the following balanced bits:
  Row 0 ⇒ [32, 38, 40, 77, 79, 80, 81, 87]
  Row 1 ⇒ [30, 31, 32, 38, 40, 77, 78, 79, 80, 81, 82, 87]
  Row 2 ⇒ [0, 1, 2, 3, 7, 8, 9, 10, 14, 17, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 54, 55, 57, 64, 65, 69, 70, 71, 72, 73, 75, 76, 77, 78, 79, 80, 81, 82, 84, 85, 86, 87, 88, 89, 90, 91, 92, 94, 95]
  Row 3 ⇒ [0, 1, 2, 9, 23, 24, 25, 28, 29, 30, 31, 32, 33, 35, 36, 37, 38, 39, 40, 41, 42, 43, 46, 47, 48, 49, 50, 54, 55, 56, 57, 64, 71, 72, 73, 77, 78, 79, 80, 81, 82, 83, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95]

### A.3 Results on KNOT-512

Results on the 19-round KNOT-512 with $\mathbb{S}' = \{(0,0),(1,0),(2,0),(3,0)\}$ and obtained the following results for $OUT[(k,\ell)]$.

- $a^0 = in_{(0,0)}$ gives the following balanced bits:
  Row 0 ⇒ [0, 7, 8, 14, 15, 16, 23, 30, 31, 32, 39, 40, 41, 47, 48, 55, 56, 63, 64, 65, 79, 80, 87, 88, 89, 95, 96, 103, 104, 111, 112, 119, 120, 127]
  Row 1 ⇒ [0, 7, 8, 14, 15, 16, 23, 30, 31, 32, 39, 40, 41, 47, 48, 55, 56, 63, 64, 65,

71, 79, 80, 87, 88, 89, 95, 96, 103, 104, 111, 112, 118, 119, 120, 127]
Row 2 ⇒ [0, 5, 6, 7, 8, 9, 14, 15, 16, 17, 22, 23, 24, 29, 30, 31, 32, 33, 38, 39, 40, 41, 46, 47, 48, 54, 55, 56, 57, 62, 63, 64, 65, 70, 71, 72, 78, 79, 80, 81, 86, 87, 88, 89, 94, 95, 96, 102, 103, 104, 105, 109, 110, 111, 112, 113, 118, 119, 120, 121, 126, 127]
Row 3 ⇒ [0, 5, 6, 7, 8, 14, 15, 16, 17, 23, 24, 29, 30, 31, 32, 33, 38, 39, 40, 41, 47, 48, 55, 56, 57, 63, 64, 65, 71, 72, 79, 80, 81, 86, 87, 88, 89, 94, 95, 96, 103, 104, 105, 109, 110, 111, 112, 113, 118, 119, 120, 121, 127]

- $a^0 = in_{(1,0)}$ gives the following balanced bits:
  Row 0 ⇒ [0, 1, 5, 6, 7, 8, 9, 14, 15, 16, 17, 22, 23, 24, 25, 29, 30, 31, 32, 33, 38, 39, 40, 41, 47, 48, 49, 55, 56, 57, 62, 63, 64, 65, 71, 72, 73, 79, 80, 87, 88, 89, 94, 95, 96, 97, 102, 103, 104, 105, 110, 111, 112, 113, 118, 119, 120, 121, 126, 127]
  Row 1 ⇒ [0, 1, 5, 6, 7, 8, 9, 14, 15, 16, 17, 22, 23, 24, 25, 29, 30, 31, 32, 33, 38, 39, 40, 41, 47, 48, 49, 55, 56, 57, 62, 63, 64, 65, 70, 71, 72, 73, 78, 79, 80, 81, 87, 88, 89, 94, 95, 96, 97, 102, 103, 104, 105, 110, 111, 112, 113, 118, 119, 120, 121, 126, 127]
  Row 2 ⇒ [0, 1, 5, 6, 7, 8, 9, 14, 15, 16, 17, 22, 23, 24, 25, 29, 30, 31, 32, 33, 38, 39, 40, 41, 46, 47, 48, 49, 54, 55, 56, 57, 62, 63, 64, 65, 70, 71, 72, 73, 78, 79, 80, 81, 86, 87, 88, 89, 93, 94, 95, 96, 97, 102, 103, 104, 105, 109, 110, 111, 112, 113, 118, 119, 120, 121, 126, 127]
  Row 3 ⇒ [0, 1, 5, 6, 7, 8, 9, 14, 15, 16, 17, 22, 23, 24, 25, 29, 30, 31, 32, 33, 38, 39, 40, 41, 46, 47, 48, 49, 54, 55, 56, 57, 62, 63, 64, 65, 70, 71, 72, 73, 78, 79, 80, 81, 86, 87, 88, 89, 94, 95, 96, 97, 102, 103, 104, 105, 109, 110, 111, 112, 113, 117, 118, 119, 120, 121, 126, 127]
- $a^0 = in_{(2,0)}$ the following balanced bits:
  Row 0 ⇒ [0, 7, 8, 14, 15, 16, 23, 24, 30, 31, 32, 39, 40, 41, 47, 48, 55, 56, 63, 64, 65, 71, 79, 80, 87, 88, 89, 95, 96, 103, 104, 111, 112, 119, 120, 127]
  Row 1 ⇒ [0, 6, 7, 8, 14, 15, 16, 23, 24, 30, 31, 32, 39, 40, 41, 47, 48, 55, 56, 63, 64, 65, 71, 79, 80, 87, 88, 89, 95, 96, 103, 104, 111, 112, 118, 119, 120, 127]
  Row 2 ⇒ [0, 5, 6, 7, 8, 9, 14, 15, 16, 17, 22, 23, 24, 29, 30, 31, 32, 38, 39, 40, 41, 46, 47, 48, 54, 55, 56, 57, 62, 63, 64, 65, 70, 71, 72, 78, 79, 80, 81, 86, 87, 88, 89, 94, 95, 96, 102, 103, 104, 109, 110, 111, 112, 113, 118, 119, 120, 121, 126, 127]
  Row 3 ⇒ [0, 5, 6, 7, 8, 14, 15, 16, 17, 22, 23, 24, 29, 30, 31, 32, 38, 39, 40, 41, 47, 48, 55, 56, 57, 63, 64, 65, 71, 72, 79, 80, 81, 87, 88, 89, 94, 95, 96, 103, 104, 109, 110, 111, 112, 113, 118, 119, 120, 121, 127]
- $a^0 = in_{(3,0)}$ gives the following balanced bits:
  Row 0 ⇒ [0, 1, 5, 6, 7, 8, 9, 14, 15, 16, 17, 22, 23, 24, 25, 29, 30, 31, 32, 33, 38, 39, 40, 41, 46, 47, 48, 49, 54, 55, 56, 57, 62, 63, 64, 65, 70, 71, 72, 73, 78, 79, 80, 81, 86, 87, 88, 89, 94, 95, 96, 97, 102, 103, 104, 105, 110, 111, 112, 113, 118, 119, 120, 121, 126, 127]
  Row 1 ⇒ [0, 1, 5, 6, 7, 8, 9, 14, 15, 16, 17, 22, 23, 24, 25, 29, 30, 31, 32, 33, 38, 39, 40, 41, 46, 47, 48, 49, 54, 55, 56, 57, 62, 63, 64, 65, 70, 71, 72, 73, 78, 79, 80, 81, 86, 87, 88, 89, 94, 95, 96, 97, 102, 103, 104, 105, 110, 111, 112, 113, 118, 119, 120, 121, 126, 127]
  Row 2 ⇒ [0, 1, 5, 6, 7, 8, 9, 14, 15, 16, 17, 22, 23, 24, 25, 29, 30, 31, 32, 33, 38, 39, 40, 41, 46, 47, 48, 49, 54, 55, 56, 57, 62, 63, 64, 65, 70, 71, 72, 73, 74, 78, 79,

80, 81, 86, 87, 88, 89, 93, 94, 95, 96, 97, 102, 103, 104, 105, 109, 110, 111, 112, 113, 118, 119, 120, 121, 126, 127]
Row 3 ⇒ [0, 1, 5, 6, 7, 8, 9, 14, 15, 16, 17, 22, 23, 24, 25, 29, 30, 31, 32, 33, 38, 39, 40, 41, 46, 47, 48, 49, 54, 55, 56, 57, 62, 63, 64, 65, 70, 71, 72, 73, 78, 79, 80, 81, 86, 87, 88, 89, 94, 95, 96, 97, 102, 103, 104, 105, 109, 110, 111, 112, 113, 117, 118, 119, 120, 121, 126, 127]

## B Tutorial on MiniZinc

MiniZinc [21] is a solver-independent open-source language which can be used for modeling the CSP. A complete tutorial on MiniZinc can be found in `https://www.minizinc.org/doc-2.5.3/en/part_2_tutorial.html#`. Here we provide a small tutorial on the modeling of the division property propagation. To demonstrate this we consider a small SPN structure as same as the KNOT with $b = 32$. The Sbox is given in Table 13 and the shift row offsets are given in Table 14. We can view each state of this as a two-dimensional matrix of size $4 \times 8$

| $x$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $S(x)$ | 6 | 0 | 11 | 5 | 9 | 4 | 2 | 7 | 14 | 8 | 15 | 1 | 12 | 13 | 3 | 10 |

Table 13: Sbox

| $b$ | $c_1$ | $c_2$ | $c_3$ |
|---|---|---|---|
| 32 | 1 | 5 | 7 |

Table 14: Shift Row offsets

and consequently the variables are denoting round properties, are also matrices of the same size. As we have explained for the KNOT, we have the following chain of propagation:

$$a^0 \xrightarrow{\text{Sbox}} a^1 \xrightarrow{\text{rotation}} b^1 \xrightarrow{\text{Sbox}} a^2 \xrightarrow{\text{rotation}} \cdots b^{r-1} \xrightarrow{\text{Sbox}} a^r.$$

To denote a two-dimensional variable, MiniZinc has *array* structure and we can declare the input variable $a^i$ as:

array[0..3,0..7] of var bool : ai;

The next task is to find the logical formula corresponding to the Sbox division trail table. We can declare this formula in MiniZinc using the "predicate". Finally to call the predicate we just have to add a constraint. For example, the first

column of $a^0$ and $a^1$ are connected with the predicate formed by the above Sbox, so we add :

constraint sbox($a0[3,0], a0[2,0], a0[1,0], a0[0,0], a1[3,0], a1[2,0], a1[1,0], a1[0,0]$);

A complete MiniZinc code is given below in Table 15 for 2-round of our small cipher. We named the file as **smallKNOT.mzn**.

Suppose we want to check with the input property

$$in_{(0,0)} = \begin{bmatrix} 0\,1\,1\,1\,1\,1\,1\,1 \\ 1\,1\,1\,1\,1\,1\,1\,1 \\ 1\,1\,1\,1\,1\,1\,1\,1 \\ 1\,1\,1\,1\,1\,1\,1\,1 \end{bmatrix}$$

and the output property

$$out_{(3,0)} = \begin{bmatrix} 0\,0\,0\,0\,0\,0\,0\,0 \\ 0\,0\,0\,0\,0\,0\,0\,0 \\ 0\,0\,0\,0\,0\,0\,0\,0 \\ 1\,0\,0\,0\,0\,0\,0\,0 \end{bmatrix}.$$

Then we need to write one more file for this data. The data file **data.dzn** is given in Table 16.

```
a0 = array2d(0..3, 0..7,
        [|false, true, true, true, true, true, true, true|
        true, true, true, true, true, true, true, true|
        true, true, true, true, true, true, true, true|
        true, true, true, true, true, true, true, true|]);

a2 = array2d(0..3, 0..7,
        [|false, false, false, false, false, false, false, false|
        false, false, false, false, false, false, false, false|
        false, false, false, false, false, false, false, false|
        true, false, false, false, false, false, false, false|]);
```

Table 16: data.dzn

To run this code with the data file we can use the following command:

```
> minizinc --solver chuffed smallKNOT.mzn data.dzn
```

## C Note on the Quine-McCluskey (QM) Algorithm

If a Boolean function has a greater number of variables, simplification using Boolean algebra is a hard task. The Quine-McCluskey (QM) algorithm [20,22,23]

```
array[0..3,0..7] of var bool : a0;
array[0..3,0..7] of var bool : a1;
array[0..3,0..7] of var bool : a2;
predicate sbox(var bool:x0, var bool:x1, var bool:x2, var bool:x3,
var bool:y0, var bool:y1, var bool:y2, var bool:y3 ) =
((not x1)\/(not x2)\/y1\/y2\/(not y3)) /\
((not x0)\/y0\/y1\/y3) /\
(x2\/(not y1)\/(not y2)) /\
(x1\/(not y1)\/(not y2)\/(not y3)) /\
(x3\/(not y0)\/(not y3)) /\
((not y0)\/y1\/(not y3)) /\
(x2\/(not y0)\/(not y2)) /\
((not y0)\/(not y1)\/y3) /\
(x3\/(not y1)\/(not y2)\/(not y3)) /\
((not x3)\/y0\/y1\/y2\/y3) /\
(x1\/x3\/(not y1)\/(not y2)) /\
(x0\/x1\/x2\/x3\/(not y0)) /\
((not x1)\/(not x2)\/(not x3)\/y1\/(not y3)) /\
((not x1)\/y0\/y1\/y2\/y3) /\
(x0\/x1\/(not y0)\/(not y2)) /\
((not x2)\/(not x3)\/(not y1)\/y2\/y3) /\
((not x1)\/(not x2)\/(not x3)\/y2) /\
(x2\/(not y1)\/(not y3)) /\
(x1\/x3\/(not y1)\/(not y3)) /\
(x0\/x3\/(not y2)\/(not y3)) /\
((not x1)\/(not x3)\/(not y1)\/(not y2)\/y3) /\
(x3\/(not y0)\/(not y2)) /\
((not x2)\/(not x3)\/y1\/y2\/(not y3)) /\
(x0\/(not y0)\/(not y1)) /\
((not x2)\/(not x3)\/y0\/y1\/y3) /\
(x0\/x3\/(not y1)\/(not y2)) /\
(x1\/x2\/x3\/(not y2)) /\
((not x0)\/(not x1)\/(not x3)\/y1\/(not y2)) /\
((not x0)\/(not x2)\/(not x3)\/y2\/y3) /\
(x0\/x1\/x2\/x3\/(not y1)) /\
((not x0)\/(not x2)\/y1\/y2\/(not y3)) /\
(x2\/(not y2)\/(not y3)) /\
((not x0)\/(not x1)\/(not x2)\/(not y1)\/y2\/y3) /\
(x0\/x1\/x2\/x3\/(not y3)) /\
((not x0)\/(not x3)\/y0\/(not y3)) /\
((not x2)\/y0\/y1\/y2\/y3) /\
(x0\/x3\/(not y1)\/(not y3));

constraint sbox(a0[3,0], a0[2,0], a0[1,0], a0[0,0],  a1[3,0], a1[2,0], a1[1,0], a1[0,0]);
constraint sbox(a0[3,1], a0[2,1], a0[1,1], a0[0,1],  a1[3,1], a1[2,1], a1[1,1], a1[0,1]);
constraint sbox(a0[3,2], a0[2,2], a0[1,2], a0[0,2],  a1[3,2], a1[2,2], a1[1,2], a1[0,2]);
constraint sbox(a0[3,3], a0[2,3], a0[1,3], a0[0,3],  a1[3,3], a1[2,3], a1[1,3], a1[0,3]);
constraint sbox(a0[3,4], a0[2,4], a0[1,4], a0[0,4],  a1[3,4], a1[2,4], a1[1,4], a1[0,4]);
constraint sbox(a0[3,5], a0[2,5], a0[1,5], a0[0,5],  a1[3,5], a1[2,5], a1[1,5], a1[0,5]);
constraint sbox(a0[3,6], a0[2,6], a0[1,6], a0[0,6],  a1[3,6], a1[2,6], a1[1,6], a1[0,6]);
constraint sbox(a0[3,7], a0[2,7], a0[1,7], a0[0,7],  a1[3,7], a1[2,7], a1[1,7], a1[0,7]);
constraint sbox(a1[3,7], a1[2,5], a1[1,1], a1[0,0],  a2[3,0], a2[2,0], a2[1,0], a2[0,0]);
constraint sbox(a1[3,8], a1[2,6], a1[1,2], a1[0,1],  a2[3,1], a2[2,1], a2[1,1], a2[0,1]);
constraint sbox(a1[3,9], a1[2,7], a1[1,3], a1[0,2],  a2[3,2], a2[2,2], a2[1,2], a2[0,2]);
constraint sbox(a1[3,10], a1[2,8], a1[1,4], a1[0,3],  a2[3,3], a2[2,3], a2[1,3], a2[0,3]);
constraint sbox(a1[3,11], a1[2,9], a1[1,5], a1[0,4],  a2[3,4], a2[2,4], a2[1,4], a2[0,4]);
constraint sbox(a1[3,12], a1[2,10], a1[1,6], a1[0,5],  a2[3,5], a2[2,5], a2[1,5], a2[0,5]);
constraint sbox(a1[3,13], a1[2,11], a1[1,7], a1[0,6],  a2[3,6], a2[2,6], a2[1,6], a2[0,6]);
constraint sbox(a1[3,14], a1[2,12], a1[1,8], a1[0,7],  a2[3,7], a2[2,7], a2[1,7], a2[0,7]);

solve satisfy;
```

Table 15: smallKNOT.mzn

is a systematic approach to find a minimum size CNF from the truth table of a function. This method is based on the reduction principle. Consider the expression $(X \wedge y) \vee (X \wedge \overline{y})$ where $X$ can be a variable or product of variables and $y$ is a variable. Then by the reduction principle we have:

$$(X \wedge y) \vee (X \wedge \overline{y}) = X.$$

All the terms in a formula can be tested for possible reduction. The irreducible terms are called *Prime Implicant* (PI). All the PIs are not necessary to represent a function. Some PIs can be covered by others. The PIs which are necessary to cover all the original terms, called *Essential Prime Implicant* (EPI).

Let us consider a function as we got for the valid division trail table. Let the function $F : \mathbb{F}_2^n \to \mathbb{F}_2$ be defined as

$$F(x) = \begin{cases} 1, \text{ if } x \in T \\ 0, \text{ if } x \in T^c \end{cases}$$

To use the QM method, the function needs to be given as a sum of minterms. These minterms can be found by the following mapping from $\{0, 1\}$ to $\{x, \overline{x}\}$:

$$m(b) = \begin{cases} x, \text{ if } b = 1 \\ \overline{x}, \text{ if } b = 0 \end{cases}$$

For each point in $(x_{n-1}, \cdots, x_0) \in T$, we add the minterm $m(x_{n-1}) \wedge m(x_{n-2}) \wedge \cdots \wedge m(x_0)$. For example if $(0, 1, 0, 1) \in T$ then we add the minterm $m(0) \wedge m(1) \wedge m(0) \wedge m(1) = \overline{x}_3 \wedge x_2 \wedge \overline{x}_1 \wedge x_0$.

The QM algorithm first finds all the PIs. To do this it combines all the minterms using reduction principle. For example the minterm corresponding to 0110 and 0100 can be combined to $01 - 0$, where '-' represents the canceled bit. This is justified as:

$$\begin{aligned} (\overline{x}_4 \wedge x_2 \wedge x_1 \wedge \overline{x_0}) \vee (\overline{x}_4 \wedge x_2 \wedge \overline{x}_1 \wedge \overline{x_0}) &= \overline{x}_4 \wedge x_2 \wedge \overline{x_0} \wedge (x_1 \vee \overline{x}_1) \\ &= \overline{x}_4 \wedge x_2 \wedge \overline{x_0} \end{aligned}$$

This combination can be formed with the terms which differ in one variable. These procedure continues until we get all the PIs. Finally, it selects EPIs. These EPIs are necessary to cover the function but may not be sufficient. In that case, the next step is to choose some more PIs to represent the Boolean function. This step is done heuristically. So the reader may get different result for the KNOT Sbox than we have given in Table 17. Moreover, this method can also Handel 'Don't-Care' conditions i.e., there may be some input to a function such that we don't care about the corresponding output. Performance of QM algorithm significantly depends on the data structures used in implementation. For this direction we refer the interested readers to [15].

The CNF corresponding to the division trail table of the KNOT Sbox is 'AND' of the following clauses given in Table 17.

| $x_3$ | $x_2$ | $x_1$ | $x_0$ | $y_3$ | $y_2$ | $y_1$ | $y_0$ |
|---|---|---|---|---|---|---|---|
| $\bar{x_3}\vee$ | $\bar{x_2}\vee$ | | $\bar{x_0}\vee$ | | $\bar{y_2}\vee$ | $y_1\vee$ | |
| $x_3\vee$ | $x_2\vee$ | $x_1\vee$ | $x_0\vee$ | $\bar{y_3}\vee$ | | | |
| $\bar{x_3}\vee$ | | | | $y_3\vee$ | $y_2\vee$ | $y_1\vee$ | $y_0\vee$ |
| | $\bar{x_2}\vee$ | | | $y_3\vee$ | $y_2\vee$ | $y_1\vee$ | $y_0\vee$ |
| $x_3\vee$ | $x_2\vee$ | $x_1\vee$ | $x_0\vee$ | | $\bar{y_2}\vee$ | | |
| $x_3\vee$ | $x_2\vee$ | $x_1\vee$ | $x_0\vee$ | | | | $\bar{y_0}\vee$ |
| $\bar{x_3}\vee$ | | $\bar{x_1}\vee$ | | $\bar{y_3}\vee$ | $y_2\vee$ | | |
| | | | $\bar{x_0}\vee$ | $y_3\vee$ | $y_2\vee$ | $y_1\vee$ | $y_0\vee$ |
| | $x_2\vee$ | | | | $\bar{y_2}\vee$ | | $\bar{y_0}\vee$ |
| | $\bar{x_2}\vee$ | $\bar{x_1}\vee$ | $\bar{x_0}\vee$ | | $\bar{y_2}\vee$ | | $y_0\vee$ |
| $x_3\vee$ | | $x_1\vee$ | | | $\bar{y_2}\vee$ | | $\bar{y_0}\vee$ |
| | | $x_1\vee$ | | | $\bar{y_2}\vee$ | $y_1\vee$ | $\bar{y_0}\vee$ |
| $\bar{x_3}\vee$ | $\bar{x_2}\vee$ | $\bar{x_1}\vee$ | $\bar{x_0}\vee$ | $y_3\vee$ | | | |
| | $\bar{x_2}\vee$ | $\bar{x_1}\vee$ | | | $y_2\vee$ | $y_1\vee$ | $y_0\vee$ |
| $\bar{x_3}\vee$ | $x_2\vee$ | | $\bar{x_0}\vee$ | $\bar{y_3}\vee$ | $y_2\vee$ | | |
| | | $\bar{x_1}\vee$ | | $y_3\vee$ | $y_2\vee$ | $y_1\vee$ | $y_0\vee$ |
| | $\bar{x_2}\vee$ | $\bar{x_1}\vee$ | | $y_3\vee$ | $y_2\vee$ | | $\bar{y_0}\vee$ |
| | | | | | $y_2\vee$ | $\bar{y_1}\vee$ | $\bar{y_0}\vee$ |
| $\bar{x_3}\vee$ | | | $x_0\vee$ | $\bar{y_3}\vee$ | | | $\bar{y_0}\vee$ |
| | | $\bar{x_1}\vee$ | $\bar{x_0}\vee$ | $y_3\vee$ | $\bar{y_2}\vee$ | $y_1\vee$ | |
| | | $x_1\vee$ | | $\bar{y_3}\vee$ | $\bar{y_2}\vee$ | $\bar{y_1}\vee$ | |
| | | | $x_0\vee$ | | $\bar{y_2}\vee$ | $\bar{y_1}\vee$ | |
| | $\bar{x_2}\vee$ | | $\bar{x_0}\vee$ | $y_3\vee$ | $y_2\vee$ | $\bar{y_1}\vee$ | |
| $\bar{x_3}\vee$ | $\bar{x_2}\vee$ | | | $y_3\vee$ | $\bar{y_2}\vee$ | | $y_0\vee$ |
| $\bar{x_3}\vee$ | | $\bar{x_1}\vee$ | | $y_3\vee$ | | $y_1\vee$ | $y_0\vee$ |
| | $x_2\vee$ | $x_1\vee$ | | | $\bar{y_2}\vee$ | $\bar{y_1}\vee$ | |
| | $x_2\vee$ | $x_1\vee$ | $x_0\vee$ | $\bar{y_3}\vee$ | $\bar{y_2}\vee$ | | |
| | $\bar{x_2}\vee$ | | $\bar{x_0}\vee$ | $y_3\vee$ | $\bar{y_2}\vee$ | $y_1\vee$ | |
| | $\bar{x_2}\vee$ | | $\bar{x_0}\vee$ | | $y_2\vee$ | $y_1\vee$ | $y_0\vee$ |
| $\bar{x_3}\vee$ | $\bar{x_2}\vee$ | | $x_0\vee$ | $\bar{y_3}\vee$ | $y_2\vee$ | | |
| $x_3\vee$ | | | $x_0\vee$ | | $\bar{y_2}\vee$ | | $\bar{y_0}\vee$ |
| | $\bar{x_2}\vee$ | $\bar{x_1}\vee$ | $\bar{x_0}\vee$ | | $y_2\vee$ | | $\bar{y_0}\vee$ |
| $\bar{x_3}\vee$ | $\bar{x_2}\vee$ | | $\bar{x_0}\vee$ | $y_3\vee$ | | $y_1\vee$ | |
| | | $\bar{x_1}\vee$ | $\bar{x_0}\vee$ | $y_3\vee$ | $y_2\vee$ | $\bar{y_1}\vee$ | |
| $\bar{x_3}\vee$ | | | $\bar{x_0}\vee$ | $y_3\vee$ | $\bar{y_2}\vee$ | $y_1\vee$ | |
| $x_3\vee$ | | | $x_0\vee$ | $\bar{y_3}\vee$ | $\bar{y_2}\vee$ | | |
| | $x_2\vee$ | | | $\bar{y_3}\vee$ | | $\bar{y_1}\vee$ | |
| | $x_2\vee$ | | | $\bar{y_3}\vee$ | | | $\bar{y_0}\vee$ |
| $x_3\vee$ | | $x_1\vee$ | | $\bar{y_3}\vee$ | | | $\bar{y_0}\vee$ |
| $x_3\vee$ | $x_2\vee$ | $x_1\vee$ | $x_0\vee$ | | | $\bar{y_1}\vee$ | |