# Trail Search with CRHS Equations

John Petter Indrøy and Håvard Raddum
johnpetter@simula.no, haavardr@simula.no

Simula UiB, Bergen, Norway

**Keywords:** differential cryptanalysis · linear cryptanalysis · CRHS equations

**Abstract.** Evaluating a block cipher's strength against differential or linear cryptanalysis can be a difficult task. Several approaches for finding the best differential or linear trails in a cipher have been proposed, such as using mixed integer linear programming or SAT solvers. Recently a different approach was suggested, modelling the problem as a staged, acyclic graph and exploiting the large number of paths the graph contains.
This paper follows up on the graph-based approach and models the problem via compressed right-hand side equations. The graph we build contains paths which represent differential or linear trails in a cipher with few active S-boxes. Our method incorporates control over the memory usage, and the time complexity scales linearly with the number of rounds of the cipher being analysed.
The proposed method is made available as a tool, and using it we are able to find differential trails for the Klein and Prince ciphers with higher probabilities than previously published.

## 1 Introduction

Block ciphers have been around for decades, with the 20-year old Advanced Encryption Standard (AES) as the most prominent example. Still, there have been a number of different new symmetric ciphers proposed over the years. Lightweight ciphers are designed to be used in constrained devices and are designed to minimize the gate count, chip size or energy consumption [6,24,11,7,14,4]. Others are designed to be used with other specific cryptographic constructions like FHE, MPC, or SNARKs [2,9,19,12,1]. We can therefore expect new designs to come up in the future as well, and these will need to be cryptanalyzed for security.

Two of the oldest types of attacks on block ciphers are differential [5] and linear attacks [18]. Showing resistance to differential and linear attacks is important when proposing a new design, but it may be hard to give an accurate estimate on the strength of a cipher against these attacks. Lower bounds on the number of active S-boxes in a differential or linear trail are sometimes proved over a few rounds [10,7,14] and used to show that the full cipher must be resistant to differential and linear attacks. However, the true complexity of the attack is given by all trails in a differential or a linear hull, and it is generally

unknown how many low-weight trails they contain when the number of rounds increases.

To simplify the analysis work for new designs there is and has been a need for algorithms and tools that estimate a cipher's strength against linear and differential attacks.

### 1.1 Previous Work

Several methods and tools to aid in estimating a cipher's strength against differential and linear cryptanalysis have been proposed. As early as 1994 Matsui proposed his branch and bound algorithm [18], which recursively searches through the whole search space of trails with weight lower than a given bound. This method was successfully applied to DES, but for most new ciphers this exhaustive search technique has too high complexity to be applied in practice.

Another suggested method is to represent the problem as a mixed integer linear programming (MILP) problem [21,25,13,27,28]. This turns the problem into an optimization problem one can attempt to solve using a MILP solver. The drawback of converting the trail search into a MILP problem is that the complexity of solving increases sharply with the number of rounds. Moreover, very many inequalities must be used to exactly model the possible differentials of an S-box. To get a reasonable running time only a subset of the inequalities are used in practice, which means the solver may return solutions that do not correspond to actual trails. Converting the problem into a SAT- or SMT instance and run a SAT/SMT solver on it has also been suggested [20,16,3] and used with success. However, the complexity of using this method also increases sharply with the number of rounds. Both of these approaches have have had some success in finding trails of low weights. Some of these works have been made into tools that are supposed to simplify the job for the cryptanalyst to use, but the user must still code in the inequalities (for MILP) or clauses (for SAT/SMT) him/herself.

The newest approach to the problem is to use a representation via a staged directed acyclic graph (DAG). In [15] the problem is attacked in this way, where each node corresponds to a cipher state in a trail, and each path from start to end in the DAG corresponds to a full trail. This approach has the benefit of being able to combine (even exponentially) many trails in a linear hull or a differential and add up their weights for an accurate attack complexity. The drawback of the approach is that one can only store a limited number of nodes (i.e., cipher block states) at every stage, out of all $2^n$ possible states. It is difficult to tell beforehand which states to include in the node set of the constructed graph, and in [15] they simply choose the states with the lowest weights, limited by available memory.

The method from [15] has been implemented as a tool called CryptaGraph [26]. This tool is arguably the easiest to use for a cryptanalyst among the published tools for finding differential or linear trails. The user only needs to give a reference implementation of a given cipher (but must be programmed in Rust), and does not need to understand anything about how the underlying method works.

### 1.2 Our Contribution

In this paper we follow up on the work from [15] and give a new method for searching for linear and differential trails. We also take the approach of a staged graph, but use it in a different way than in [15]. Instead of having a 1-1 correspondence between nodes and cipher states, we let partial paths in the graph represent the cipher states. It is then not necessary to make a choice of which cipher states to include in the search space. We can simply start with a graph containing $n$ vertices and $2^n$ paths, representing all possible cipher states of $n$ bits. From that starting point, we build on the theory of CRHS equations [23,22] to construct the full graph.

We compare the results of our work to [15], and improve on some of CryptaGraph's results for differential cryptanalysis. We find some new low-weight trails that CryptaGraph misses, and we can explain why. We have also made a tool, called PathFinder, implementing our proposed method. PathFinder is as easy to use as CryptaGraph; the cryptanalyst only needs to provide the same reference implementation to PathFinder to use it. In fact, for our tests we reused all implemented SPN ciphers in [26].

We found one interesting result on the block cipher Prince that is worth mentioning here. In [7] the designers of Prince prove that four rounds of the cipher must contain at least 16 active S-boxes, and deduce that any trail in the full 12-round Prince must have at least 48 active S-boxes. To our knowledge, it has not been previously known how tight the bound is, i.e. whether it is actually possible to join three 4-round trails with the minimum active S-boxes together to form a 12-round trail with 48 active S-boxes. PathFinder finds a differential trail with 48 active S-boxes for Prince, showing that the bound given by the designers is indeed tight and can not be improved.

Finally, we highlight the strong and weak parts of our method and [15], and sketch an idea of how they can be combined to take advantage of each other's strengths. A combined tool will most likely outperform both CryptaGraph and PathFinder.

**Outline:** The paper is organized as follows. In Section 2 we recall the necessary basics of linear and differential cryptanalysis and introduce notation. We give the basics of CRHS equations in Section 3. Our proposed method for finding low-weight trails is explained in Section 4, and in Section 5 we present the results, with comparisons to CryptaGraph. Section 6 concludes the paper.

## 2 Differential and Linear Cryptanalysis

Differential [5] and linear [17] cryptanalysis are some of the earliest attacks on modern block ciphers. The attacks can be applied to ciphers which use S-boxes for non-linear mappings. In this paper we only consider SPN ciphers, but the techniques we describe can be applied to Feistel ciphers, ciphers with incomplete S-box layers, or other constructions that only use S-boxes for non-linearity.

### 2.1 Cipher model

We now describe the model we use for a general SPN cipher $\varepsilon(P, K)$ that encrypts plaintexts $P$ using the secret key $K$. The plaintext block consists of $n$ bits, which is transformed by applying a key-dependent round function $r$ times. The round function in round $i$ is denoted $\mathcal{R}_i$ and starts with the application of an S-box layer called $\mathcal{S}$, followed by a (possibly round-dependent) linear transformation $\mathcal{L}_i$, an addition of a round constant, and a key addition:

$$\mathcal{R}_i(x) = \mathcal{L}_i(\mathcal{S}(x)) \oplus z_i \oplus k_i,$$

where $k_i$ is an $n$-bit round key and $z_i$ is the round constant for round $i$. The S-box layer $\mathcal{S}$ consists of the parallel application of $m$ S-boxes, each substituting one $b$-bit chunk of the cipher state with another one according to a given table, where $n = bm$. The complete cipher starts with an initial key addition on the plaintext $P$, followed by applying the round function $r$ times. The output is the ciphertext $C$:

$$C = \varepsilon(P, K) = \mathcal{R}_r \circ \ldots \circ \mathcal{R}_1(P \oplus k_0).$$

When searching for differential or linear trails we disregard the additions of the round keys and round constants. We are therefore not concerned with modelling the key schedule in this work.

### 2.2 Differential distribution table and linear approximation table

The basis for both differential and linear attacks are the imbalances that exist in the S-box $S$ that is used. For differential attacks, we exploit that some input/output differences in the S-box are more likely than others. For given $b$-bit differences $\alpha$ and $\beta$ we define the *differential count* $DC(\alpha, \beta)$ to be

$$DC(\alpha, \beta) = |\{x \in GF(2)^b \mid S(x) \oplus S(x \oplus \alpha) = \beta\}|.$$

By varying $\alpha$ and $\beta$ we can build a *differential distribution table* (DDT) of size $2^b \times 2^b$ containing all possible differential counts of an S-box:

$$\mathrm{DDT}[\alpha][\beta] = DC(\alpha, \beta).$$

The entries in the DDT that are 0 indicate impossible differentials, i.e. input/output differences that can not occur. These differences can not be used when constructing a differential trail through an SPN cipher. More generally, $\mathrm{DDT}[\alpha][\beta]$ indicate the probability for getting the specific output difference $\beta$ for a given input difference $\alpha$.

The imbalance that is exploited in a linear attack is the fact that some linear combinations of input/output bits are more correlated than others. For two *masks* $\gamma$ and $\delta$ we define the *linear correlation* $LC(\gamma, \delta)$ of $S$ to be

$$LC(\gamma, \delta) = |\{x \in GF(2)^b \mid \langle \gamma, x \rangle = \langle \delta, S(x) \rangle\}|,$$

where $\langle \cdot, \cdot \rangle$ denote the bit-wise inner product between two bit-strings of equal length. By running through all combinations of $\gamma, \delta$ we can construct the *linear approximation table* (LAT) of size $2^b \times 2^b$ in a similar fashion as the DDT:

$$\text{LAT}[\gamma][\delta] = |2LC(\gamma, \delta) - 2^b|.$$

By defining the LAT in this way, the input/output masks that give no bias in the correlation get the value 0 in the LAT. These input/output masks cannot be used when making a linear trail through a cipher. In the same way higher numbers in a DDT indicate higher probabilities of a differential to occur, higher numbers in the LAT indicate stronger bias for a correlation. Finally, we also have $\text{DDT}[0][0] = \text{LAT}[0][0] = 2^b$.

Since the DDT and the LAT share the properties that only non-zero values in the table can be used for constructing trails and that higher numbers mean better trails (from an attacker's point of view), we will treat them at the same time in the text that follows, and use the term *base table* (BT) for referring to either one of them.

### 2.3 Trails

Given a base table for an S-box, we are interested in expanding the differential counts or linear correlations to cover the whole cipher $\varepsilon$. In other words, we are interested in finding $(\alpha, \beta) \in GF(2)^n \times GF(2)^n$ such that $Pr[\varepsilon(P, K) \oplus \varepsilon(P \oplus \alpha, K) = \beta]$ is high, or finding $(\gamma, \delta) \in GF(2)^n \times GF(2)^n$ such that $\langle \gamma, P \rangle = \langle \delta, \varepsilon(P, K) \rangle$ with a probability bounded away from 1/2.

In the following we will use the term *input* to an S-box to mean either an input difference or an input mask to the S-box. Similarly, the term *output* can refer to both output difference for a differential or output mask for a linear approximation. Furthermore, an S-box whose input and output are both 0 is called a *passive* S-box, while an S-box with non-zero input/output is called an *active* S-box.

The input and output for the whole S-box layer $\mathcal{S}$ is constructed in the natural way, by concatenating the inputs and outputs of individual S-boxes to $n$-bit strings. Given the output $u_i$ from $\mathcal{S}$ in round $i$, the input to $\mathcal{S}$ in round $i + 1$ is given as $\mathcal{L}_i(u_i)$. In contrast, for a given input to $\mathcal{S}$ there are in general many different possible outputs. All passive S-boxes must have the output 0, but each active S-box in $\mathcal{S}$ can have a number of possible outputs. For a given input $\alpha_i$ to S-box $i$, any $\beta_i$ such that $\text{BT}[\alpha_i][\beta_i] \neq 0$ is possible. A *trail* through $\varepsilon$ is defined as a sequence of $n$-bit strings

$$\boldsymbol{u} = (u_0, u_1, \ldots, u_r)$$

where $u_0$ is the difference or mask for the plaintext block $P$ and $u_i$ is the output of $\mathcal{S}$ in round $i$ for $i = 1, \ldots, r$. We furthermore split each $u_i$ into $u_i = (u_{i,1}, \ldots, u_{i,m})$, where each $u_{i,j}$ is the substring that aligns with S-box $j$ in $\mathcal{S}$. For a trail to be *valid* the following conditions must be met: The input/output $u_0/u_1$ of $\mathcal{S}$ in $\mathcal{R}_1$ must satisfy $\text{BT}[u_{0,j}][u_{1,j}] \neq 0$ for $1 \leq j \leq m$, and the input/output

$\mathcal{L}_i(u_i)_j / u_{i+1,j}$ for S-box $j$ in round $i+1$ satisfies $\mathrm{BT}[\mathcal{L}_i(u_i)_j][u_{i+1,j}] \neq 0$ for all $i = 1, \ldots, r-1$ and $1 \leq j \leq m$. Note that $u_r$ does not represent the ciphertext state, but the ciphertext difference or mask is uniquely determined by $u_r$ as $\mathcal{L}_r(u_r)$.

The trails through $\varepsilon$ determine the complexity for a differential or linear attack on the cipher. The numbers in the base table for all S-boxes give what we call the *trail weight* $w(\boldsymbol{u})$, which is given as

$$w(\boldsymbol{u}) = \sum_{j=1}^{m} -\log_2 \left( \frac{\mathrm{BT}[u_{0,j}][u_{1,j}]}{2^b} \right) + \sum_{i=1}^{r-1} \sum_{j=1}^{m} -\log_2 \left( \frac{\mathrm{BT}[\mathcal{L}_i(u_i)_j][u_{i+1,j}]}{2^b} \right). \tag{1}$$

Lower weight means lower complexity of mounting an attack, and we see that passive S-boxes do not add anything to the trail weight since $\mathrm{BT}[0][0]$ is always equal to $2^b$. For the security analysis of a particular cipher we are therefore interested in finding trails that give the lowest trail weight. This is a difficult task in itself, since there is a very large search space of all possible trails.

For fixed $u_0, u_r$ there are many valid trails that start with $u_0$ and end with $u_r$. We call the set of all valid trails that start with $u_0$ and ends with $u_r$ for a *hull*, and denote the set with $u_0 \diamond u_r$.

The weight of all the paths in a hull gives a good approximation for the complexity of a differential or linear attack on $\varepsilon$. The exact complexity is dependent on the actual key used in the cipher, and the weights of all S-boxes in $\varepsilon$ are not independent from each other. However, disregarding the effect of the key and the dependencies that exist between different S-boxes still gives a good approximation of the complexity of a linear or differential attack. In the literature one often considers the *expected differential probability* (EDP) and the *expected linear potential* (ELP) to estimate the complexity of an attack using a chosen hull. With our notation, we have

$$\mathrm{EDP} \approx \sum_{\boldsymbol{u} \in (u_0 \diamond u_r)} 2^{-w(\boldsymbol{u})},$$

and

$$\mathrm{ELP} \approx \sum_{\boldsymbol{u} \in (u_0 \diamond u_r)} 2^{-2w(\boldsymbol{u})}.$$

Some ciphers may have many trails that contribute approximately equally to the weight of a hull, while others may have only a few dominating trails that make up most of the weight of a hull. Either way, a good strategy for finding a hull with a low weight is to search for trails with the least number of active S-boxes. We therefore define the number of active S-boxes in a trail $\boldsymbol{u}$ as

$$a(\boldsymbol{u}) = |\{u_{i,j} \mid u_{i,j} \neq 0, 1 \leq i \leq r, 1 \leq j \leq m\}|.$$

In the following sections we describe an efficient algorithm that searches for valid trails with the lowest number of active S-boxes, and use them to give a lower bound on the EDP or ELP for $\varepsilon$.

# 3 CRHS Equations

The algorithm searching for low-weight trails uses Compressed Right-Hand Side (CRHS) equations [23] as its building block. A CRHS equation is a data structure which may be understood as a compressed representation of a large number of linear equation systems over some variable set.

## 3.1 Basics of a CRHS equations

A *Compressed Right-Hand Side Equation* (CRHS equation) is a special kind of a Directed Acyclic Graph (DAG). The DAG of a CRHS equation has exactly one source and one sink node. Each node may have at most two outgoing edges, called the 0-edge and the 1-edge. This particular class of DAG's is also known as a *Binary Decision Diagram* (BDD). The nodes in the BDD are divided into levels. We draw the DAG in a top-down fashion with the source node on the top, the sink node on the bottom, and all intermediate nodes on horizontal levels. All edges go from a node on one level to a node on the level below. If we talk about level $l$ for some number $l$, we always mean level number $l$ counted from the top, where the counting starts at 0.

Each level, except for the one containing the sink, have linear combinations of variables associated with them. These linear combinations are referred to as the *Left-Hand Side* (LHS) of the CRHS equation, while the paths in the DAG are referred to as the *Right-Hand Side* (RHS) of the CRHS equation. A *complete path* in a CRHS equation is a path which starts in the source node and ends in the sink. One such path will consist of as many edges as there are levels in the DAG, minus one.

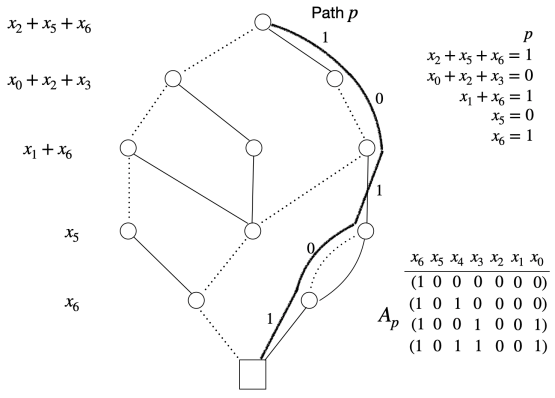



Fig. 1: CRHS equation example. Path $p$ in the CRHS equation gives a linear system with the solution set $A_p$.

Each node in the DAG can have at most two outgoing edges, the 0-edge and the 1-edge. As the names suggest, each edge has a value associated with it: 0 or 1. Choosing an outgoing edge from a node is viewed as assigning that value to the linear combination associated with that edge's level. Thus, choosing a complete path through the DAG is the same as assigning a value to all the linear combinations in the CRHS equation. By doing so, the LHS and the now assigned right-hand side becomes a system of linear equations.

Let $p(E)$ denote the set of all paths in a CRHS equation $E$, and let $A_p$ be the solution space to the linear system given by a path $p$ in $p(E)$. The *solution set* of $E$ is then given as $\cup_{p \in p(E)} A_p$. See Figure 1 for an example of a CRHS equation, the associated linear equation system for one of its paths $p$, and the solution set $A_p$.

**Operations on CRHS equations** If some linear combinations of a CRHS equation's LHS are linearly dependent there will in general exist paths in the RHS which give inconsistent linear systems, having $A_p = \emptyset$. As will become clear later, we want to remove these paths in order to find the solutions we are looking for. We remove these paths using *linear absorption*, whose operations we now explain.

**Swap levels:** This operation swaps the linear combinations of two adjacent levels, and updates the nodes and edges on these levels such that the solution set of the CRHS equation remains unchanged. The purpose of this operation is to move linear combinations up or down in the LHS of the CRHS equation.

**Adding levels:** As the name says, the linear combinations of two adjacent levels are added (xor'ed) together. When doing so, the linear combination of the lower level becomes the sum of the two, while the linear combination of the upper level stays the same. The nodes and edges in the RHS of these levels are updated accordingly so the solution set of the CRHS equation remains unchanged.

**Linear Absorption:** By using add and swap iteratively, we can do the same operations on the LHS of a CRHS equation as we can do on a binary matrix. In particular, if some linear combinations in the LHS are linearly dependent we can add them together and create a level in the CRHS equation that has $\mathbf{0}$ as its linear combination. We call such a level for a $\mathbf{0}$-level.

The paths that would give inconsistent linear systems (i.e., $A_p = \emptyset$) due to a linear dependency can be readily identified after creating a $\mathbf{0}$-level from the dependency. All paths with a 1-edge going out from a $\mathbf{0}$-level give the "equation" $\mathbf{0} = 1$ in the linear system, and hence an inconsistency. All these paths are removed by simply deleting all outgoing 1-edges from nodes on the $\mathbf{0}$-level. The last stage is to remove the whole $\mathbf{0}$-level itself. To do so, all incoming edges to nodes on the $\mathbf{0}$-level are redirected to point directly to the node at the end of the node's 0-edge (if it exists). After redirecting all incoming edges, all nodes on the $\mathbf{0}$-level are deleted. We say that the linear dependency we started with has been *absorbed*, and the CRHS equation now has one level less.

If there are several linear dependencies in the LHS of a CRHS equation, we can remove them one at a time using linear absorption. When all linear

dependencies in the LHS of a CRHS equation are absorbed, all remaining paths will give non-empty $A_p$'s, and thus the only paths left are the ones which actually contribute to the solution set of the CRHS equation. The drawback of linear absorption is that add and swap may increase the number of nodes on the affected levels, increasing the memory consumption of the CRHS equation.

Executing one linear absorption will in general leave the BDD of the CRHS equation in an unreduced state. Some nodes close to the **0**-level may have no incoming or outgoing edges; these nodes are deleted from the DAG. Moreover, some nodes may be merged, following the reduction procedure for producing a reduced ordered BDD [8]. Reduction is always performed after doing one linear absorption, to keep the number of nodes in the CRHS equation low.

### 3.2 Systems of CRHS equations

A *system of CRHS equations* (SoC) is a set of CRHS equations, all defined over the same variable set. Individual CRHS equations have their own solution set, where each path will yield a number of valid solutions to the corresponding system of linear equations. Similarly, the SoC has a solution set. The *solution set* to a SoC is the intersection of the solution sets of each of its individual CRHS equations.

**Solving a System of CRHS Equations** In order to find the solution set to a SoC, we need to absorb all the linear dependencies which exist across all the CRHS equations in the system. To enable us to identify the linear dependencies in the SoC, we use an important operation on the SoC:

*Joining* two CRHS equations is an operation where the sink node of one CRHS equation is replaced with the source node of another CRHS equation, effectively merging them into one CHRS equation. This new CHRS equation will contain all combinations of paths from the two CRHS equations.

New linear dependencies may arise in the new CRHS equation, even though the two individual CRHS equations before the join had all their dependencies resolved prior to joining. When linear dependencies appear in a joined CRHS equation, we use linear absorption to remove them. Iteratively joining two CRHS equations into one, and then absorbing all linear dependencies which arise will result in two things:

1. The SoC will eventually consist of only one CRHS equation.
2. The solution set to this CRHS equation is the solution set to the SoC.

All paths left in the final CRHS equation will give a system of linear equations with non-empty solution sets, and the union of all solution sets give the complete solution set to the SoC. Iteratively joining CRHS equations and absorbing all linear dependencies that arise is therefore a general algorithm for solving a SoC.

## 4 New Method for Finding Differential and Linear Trails

Our new method uses the theory of SoCs at its core. As will become clear, a path from the source node to the sink node in the single CRHS equation remaining in a solved SoC will represent a complete trail $(u_0, \ldots, u_r)$, and we sometimes use the terms path and trail interchangeably. For instance, we may talk about the number of active S-boxes in a path.

The cipher is represented by the SoC, and each path in its solution space corresponds to a trail, as given by the base tables and specified linear layers in $\varepsilon$. Finding the solution space in practice for full-scale ciphers will most often result in CRHS equations that are too large to handle, so we introduce a *pruning* technique as part of the solving process. Finding the part of the solution space we are interested in is done by the repeated applications of joining, linear absorption, and pruning.

When the solution space is found, we could calculate the weight of each path, and use this to find the the hull(s) with the lowest weight. In practice, the number of paths will be exponential in the number of nodes, and we need to estimate which input/output pair $u_0, u_r$ is most likely to yield the hull of lowest weight. The way our cipher is modelled allows for a linear time algorithm for counting the number of active S-boxes in all paths, and we use these counts to find our estimated pair $u_0, u_r$. The last step is then to calculate the actual weight of the hull $u_0 \diamond u_r$.

It is important to note that the pruning process will also remove valid paths from the SoC, meaning that we reduce the solution space. We can therefore only give an estimate of the weight for the best hull. The rest of this section will look at each part of this process in more detail.

### 4.1 Constructing CRHS equation from base table

We start by explaining how an individual CRHS equation is constructed from a given $b$-bit S-box with corresponding base table BT. Let the input to the S-box be represented with $\alpha = (\alpha_{b-1}, \ldots, \alpha_0) \in GF(2)^b$ and the output by $\beta = (\beta_{b-1}, \ldots, \beta_0)$.

We start by initializing a CRHS equation with $2b + 1$ levels and a DAG that initially contains only the source and sink nodes. Let the linear combinations of the levels, from top to bottom, be $\alpha_0, \alpha_1, \ldots, \alpha_{b-1}, \beta_0, \ldots, \beta_{b-1}$. Next, build a complete binary tree from the source node to level $\beta_0$. Each path from the source node to a node on level $\beta_0$ then corresponds to a fixed input $a$, and there is a unique path leading to each of these nodes. We can therefore identify a node on level $\beta_0$ with the path leading to it, so the path leading to $n_a$ represents the value $a$. For each node $n_a$ on level $\beta_0$, look up the corresponding row BT$[a]$ in the base table. For each non-zero entry BT$[a][b]$, build the path representing $b$ from $n_a$ to the sink node.

The paths in the resulting CRHS equation then encodes exactly all input/output pairs that have non-zero values in the base table. See Figure 2 for an example of a CRHS equation representing the DDT of a 3-bit S-box.

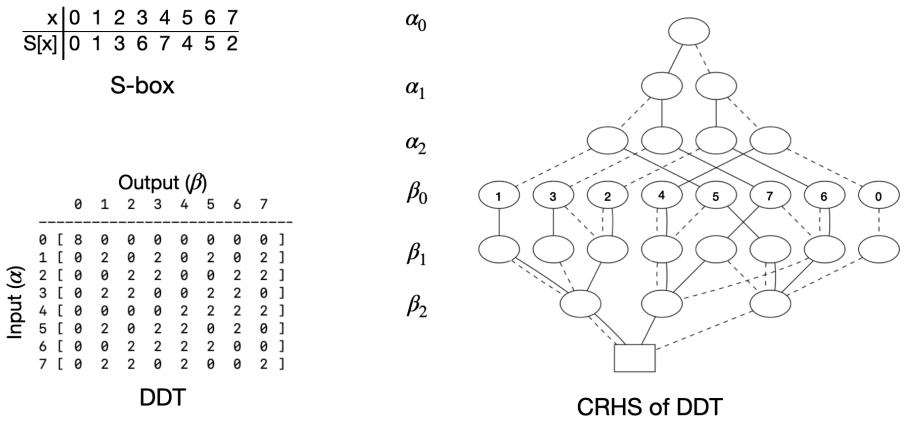



Fig. 2: DDT of a 3-bit S-box with its corresponding CRHS equation.

### 4.2 Constructing the SoC

For constructing a SoC representing a whole SPN cipher we first need to introduce variables at various points in the encryption function. The variables we introduce will not represent actual cipher states during encryption, but rather differences or masks used for differential or linear cryptanalysis, i.e. the bits that a trail is made from. We follow the cipher model described in Section 2.1.

The bits in the input to $\mathcal{S}$ in $\mathcal{R}_1$ are labelled $u_0 = (x_0, \ldots, x_{n-1})$. The bits in the output of $\mathcal{S}$ in $\mathcal{R}_i$ for $i = 1, \ldots, r$ are given as $u_i = (x_{ni}, \ldots, x_{ni+n-1})$. The input state to $\mathcal{S}$ in round $i+1$, namely $\mathcal{L}_i(u_i)$, will then be given as $n$ linear combinations in the variables $x_{ni}, \ldots, x_{ni+n-1}$, for $i = 1, \ldots, r-1$. See Figure 3 for the set-up of variables. The variable set for the SoC will be $x_0, \ldots, x_{nr+n-1}$.

There are $mr$ S-boxes used in total in $\varepsilon$. We construct one CRHS equation for each of them, following the description given in Section 4.1. The input bits of each S-box can be written as linear combinations in the variables we have introduced, and the output bits are single variables. The linear combinations of the input are inserted as the left-hand sides on the $b$ highest levels in each CRHS equation, while the variables in the output are inserted on the $b$ lowest levels. All of these CRHS equations are included in the SoC.

This way of modelling a crypto primitive as a SoC is not limited to SPN ciphers, ciphers with complete S-box layers, or S-boxes with same input and output size. With simple modifications, CRHS equations can be used to model any symmetric cipher using S-boxes for non-linearity.

In addition to the $mr$ CRHS equations constructed from the S-boxes used in $\varepsilon$, we include one more CRHS equation in the SoC. We call this for the *Master* CRHS equation, and it is constructed as follows: It consists of $n+1$ levels, with $x_0, \ldots, x_{n-1}$ as the LHS on each level, from top to bottom. There is only one

node on each level, with both the 0- and 1-edges pointing to the node on the level below, see Figure 4a.

The Master CRHS equation initially contains $2^n$ paths, representing *all* possible inputs to $\mathcal{S}$ in $\mathcal{R}_1$.

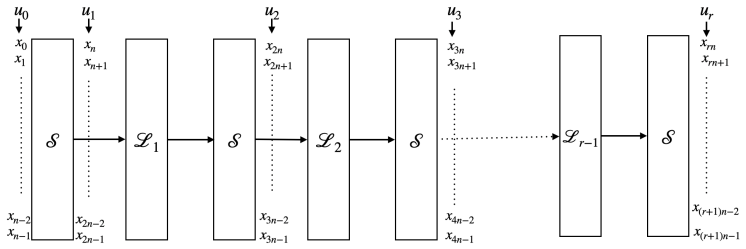



Fig. 3: Variables in the SoC, and where they appear in $\varepsilon$

### 4.3 Solving the SoC - finding valid trails

With the SoC representation, we are free to join CRHS equations in whichever order we want when running the generic solving algorithm. In our particular case we will always join CRHS equations from the S-boxes to the bottom of the Master CRHS equation, and absorb the dependencies that arises. To ensure an orderly solving process, CRHS equations will only be joined to Master if it respects Invariant 1:

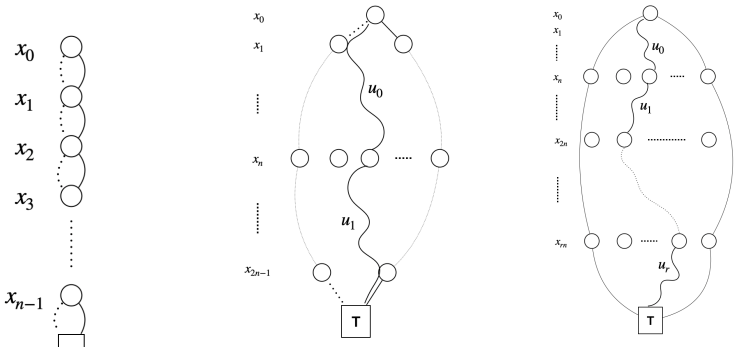
**Invariant 1** *A CRHS equation $E$ can only be joined with Master if all variables in the linear combinations on the $b$ highest levels in $E$ are already present in the LHS of Master. The $b$ linear dependencies that arises after a join operation are immediately absorbed.*

Invariant 1 ensures that all linear combinations on the $b$ highest levels of any new CRHS equation joined to Master are included in a linear dependency and can be absorbed. In the solving process we always absorb all of these dependencies after a join. Each absorption is done by taking a linear combination $lc$ from the top of the newly joined CRHS equation, and moving it upwards in Master using the swap operation. Every time $lc$ is adjacent to a level in Master that contains a variable appearing in $lc$, the add operation is used to eliminate it from $lc$. Eventually $lc = \mathbf{0}$, and is absorbed. Note that only the linear combinations are moved, so the order of all other levels with single variables are kept unchanged. When the $b$ linear combinations have been absorbed, only the single variables from the bottom of the joined CRHS equation remain on the $b$ lowest levels of Master.

Initially, Master contains all variables present in the top levels of all CRHS equations from the first round, so all of those can be joined to Master and

uphold Invariant 1. After all of these have been joined and all dependencies absorbed, Master will have $2n$ levels with the single variables $x_0, \ldots, x_{2n-1}$ as linear combinations on each of them, see Figure 4b. It is then possible to join CRHS equations from $\mathcal{R}_2$. Invariant 1 ensures we always join CRHS equations onto Master from one round $\mathcal{R}_i$ at a time.

We join the CRHS equations to Master in the natural order within each round. That is, the first CRHS equation to be joined is the one representing the first S-box in $\mathcal{R}_i$, the next one is the CRHS equation representing the second S-box in $\mathcal{R}_i$, etc., for $i = 1, \ldots, r$. This order keeps the direct association between a path and a (partial) trail; any path from the top node to level $n$ directly sets a value for $u_0$, any path from level $n$ to $2n$ gives $u_1$, etc. So every complete path from the source node to the bottom node gives a full trail $\boldsymbol{u} = (u_0, u_1, \ldots, u_r)$. Figure 4c shows a sketch of the Master CRHS equation after all joins and linear absorptions have been done.




(a) Initial Master CRHS equation

(b) Master after joining and absorbing CRHS equations from the first round

(c) Master after solving the SoC. Every path corresponds to a valid trail $(u_0, \ldots, u_r)$.

Fig. 4: Master CRHS equation at various stages.

### 4.4 Counting active S-boxes

Counting the number of active S-boxes, $a(\boldsymbol{u})$, for a given path $\boldsymbol{u}$ is fundamental both to the pruning algorithm, as well as the estimation of the input/output pair $u_0, u_r$ giving the hull $u_0 \diamond u_r$ with lowest weight. In both cases, finding the weights of each path would solve this task. However, as the number of paths is exponential in the block size $n$, searching through all of them is infeasible. On the other hand, with a bit of preparation we can count the number of active S-boxes for all paths, with a complexity that is linear in the number of nodes. We now explain this process.

We have already laid the ground work for a simple counting method with the introduction of Invariant 1. We follow up with another invariant, that always holds when joining CRHS equations and absorbing as explained above:

**Invariant 2** *The levels in Master with variables from the same S-box are adjacent.*

Invariant 2 ensures that the output of every S-box corresponds to a path of length $b$ in Master. Each of these paths start in a node on some level $l$ where $l \bmod b = 0$ and $l \geq n$, and extends to a node on level $l+b$. An S-box is counted as active iff at least one of the edges in such a path is a 1-edge. Because of Invariant 1, every CRHS equation joined into Master will add $b$ levels to the DAG which satisfy Invariant 2 after the dependencies have been absorbed.

Algorithm 1 counts the number of active S-boxes in all trails in Master, and we elaborate on it here. We first introduce the *activity distribution* $d$ for a given node $N$, defined as a vector of length $rm+1$ of integers:

$$d_N = (d_N[0], d_N[1], \ldots, d_N[rm]) \text{ where } d_N[i] = |\{\text{paths } \boldsymbol{u} \text{ below } N | a(\boldsymbol{u}) = i\}|.$$

In other words, the activity distribution counts how many sub-trails there are among the paths starting in a particular node, with a given number of active S-boxes. The activity distribution is defined for nodes on levels $l$ where $l$ is a multiple of $b$ and $l \geq n$, but can be extended to other levels by splitting a path of length $b$ in two.

Let the sink node be $T$, and initialize $d_T = (1, 0, 0, \ldots, 0)$ (that is, there is only the empty path going from $T$ to $T$, and it has no active S-boxes). The algorithm for computing the activity distributions fills the $d_N$ recursively, level by level, starting from the bottom of the DAG.

Assume that the activity distributions for every node on level $l$ have been computed with their correct numbers. Let $N$ be a node on level $l-b$. Then $d_N$ is computed as follows:

- Let $p_1, \ldots, p_k$ be the paths of length $b$ from $N$ down to nodes on level $l$. Let $N_i$ be the node on level $l$ where $p_i$ ends. As each node in the DAG has at most two outgoing edges, the number of paths of length $b$ from $N$ is at most $2^b$, so $k \leq 2^b$.
- For each $p_i$, let $w_i = d_{N_i}$ if $p_i$ is the all-zero path (indicating a passive S-box), and let $w_i = (0, d_{N_i}[0], d_{N_i}[1], \ldots, d_{N_i}[rm-1])$ if $p_i$ is not the all-zero path. When $p_i$ is non-zero, the vector $d_{N_i}$ is shifted by one position because the non-zero $p_i$ adds one active S-box to the partial trails. So if there are $j$ paths with $a$ active S-boxes from $N_i$ to $T$, there will be $j$ paths with $a+1$ active S-boxes from $N$ to $T$ that starts with $p_i$.
- Let $d_N = \sum_{i=1}^{k} w_i$. Adding up all the $w_i$ gives the number of paths below $N$, and how many active S-boxes there are in each of them.

See Figure 5 for a small example of how the number of active S-boxes in a partial trail is counted.

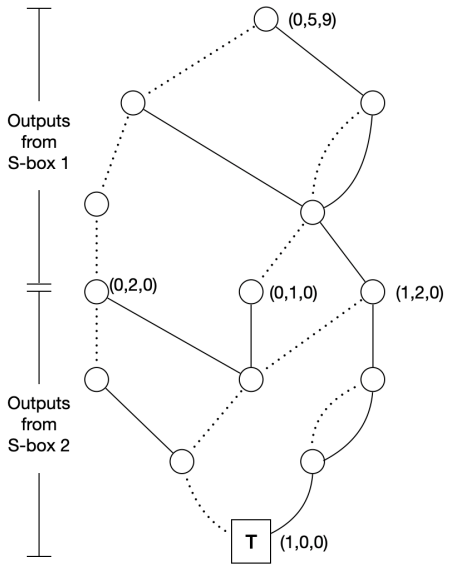



Fig. 5: Counting the number of active 3-bit S-boxes in all sub-trails involving two S-boxes, with all activity distributions shown.

This process is repeated for every node on level $l - b$, before continuing with the nodes on level $l-2b$, etc. We stop after computing $d_N$ for all nodes on level $n$. A path from any node $A$ on level $n$ to $T$ gives a trail $(u_1, \ldots, u_r)$, and $d_A[i]$ gives the number of such trails having $a(u_1, \ldots, u_r) = i$. By traversing the nodes on level $n$ and looking at their $d_N$-vectors it is then easy to find what the minimum number of active S-boxes in any trail is. Moreover, given the vectors on all levels it is easy to backtrack from a node on level $n$ down to $T$ to extract any of the trails with minimum $a(\boldsymbol{u})$.

### 4.5 Pruning - setting soft limit $\sigma$

The solving algorithm explained in Section 4.3 will give the complete picture of all possible trails in $\varepsilon$, assuming we have unlimited memory. When joining CRHS equations to Master and absorbing all the linear dependencies, the number of nodes in Master will grow. In practice there will be an upper limit on how many nodes our hardware is able to handle. When the number of nodes in Master starts to approach this limit we need to prune nodes from the DAG in order to continue extending the partial trails by joining and absorbing new CRHS equations.

The algorithm therefore uses a parameter we call *soft limit*, denoted by $\sigma$. The user must set $\sigma$ according to the memory available on the machine running the solving algorithm. Let $\mathcal{N}$ be the number of nodes in Master at any given point. If $\mu$ is the maximum $\mathcal{N}$ the machine can reasonably handle (a hard limit),

**Algorithm 1:** Computing number of active S-boxes in all trails in master CRHS equation

**Result:** Distribution of number of active S-boxes in all trails.

```
T ← sink node on level t
d_T ← (1, 0, 0, ..., 0)
l ← t − b
while l ≥ n do
    for all nodes N on level l do
        for all paths p_i of length b from N do
            N_i ← node on level l + b where p_i ends
            if p_i only has 0-edges then
                w_i = d_{N_i}
            else
                w_i = (0, d_{N_i}[0], d_{N_i}[1], ..., d_{N_i}[mr − 1])
            end if
        end for
        d_N = Σ w_i
    end for
    l ← l − b
end while
```

then $\sigma$ should be set to $\sigma \leq \mu/2^b$. Whenever $\mathcal{N} > \sigma$, pruning will delete nodes until $\mathcal{N} \leq \sigma$ before the next join and absorb will be done.

It is known that absorbing one linear dependency in a CRHS equation may in the worst case double the number of nodes in the DAG, so after joining one new CRHS equation and absorbing the $b$ dependencies that arise, $\mathcal{N}$ will in the worst case be $2^b\sigma$ before pruning. This is still below the hard limit $\mu$, so by introducing pruning and correctly setting $\sigma$ we are guaranteed that the solving algorithm will never consume too much memory when run in practice. In our current implementation of CRHS equations one node consumes 24 bytes. In addition there is some overhead, but this gives a starting point for determining $\mu$ when running PathFinder on a specific machine.

**Pruning strategy** When doing the pruning we wish to retain as many solutions in the SoC's solution space as possible. The first goal of the pruning strategy is therefore to remove as few valid trails from Master as possible. We also want the partial trails we remove to be the ones that have the most active S-boxes, since these are the least likely to turn into complete trails with few active S-boxes and low weight. While we cannot guarantee that the remaining trails will be those of minimum weight, we believe the pruning strategy explained below achieves this to a large degree and keeps the trails with the smallest number of active S-boxes, independently of which cipher is used. This is evidenced by the results from Section 5. The second goal of the pruning strategy is therefore to only delete paths with the highest number of active S-boxes.

The pruning is described in Algorithm 2, where the notation $\mathcal{N}(l)$ is used for the number of nodes on level $l$.

**Algorithm 2:** Pruning nodes from Master CRHS equation with $t+1$ levels

**Result:** Master CRHS equation with $\mathcal{N} \leq \sigma$
  **while** $\mathcal{N} > \sigma$ **do**
    $w \leftarrow$ level index such that $\mathcal{N}(w) \geq \mathcal{N}(l)$ for $0 \leq l \leq t$
    $sw \leftarrow$ level index such that $\mathcal{N}(sw) \geq \mathcal{N}(l)$ for $0 \leq l \leq t, l \neq w$
    compute $d_N$ for all nodes $N$ on level $w$
    **for all** nodes $N$ on level $w$ **do**
      $a_N = \min\{i | d_N[i] > 0\}$
    **end for**
    $A \leftarrow \max\{a_N\}$
    **while** $\mathcal{N}(w) > 0.9 \cdot \mathcal{N}(sw)$ and $\exists N$ with $a_N = A$ **do**
      delete nodes $N$ from level $w$ with $a_N = A$
    **end while**
  **end while**

The pruning algorithm starts by finding the level with the most number of nodes on it, which we call the *widest* level. We choose to always delete nodes from the widest level as this yields the best "memory to paths-lost ratio" of all the levels: All paths go through every level, so the total number of paths going through a level is constant and the same for any level. This in turn means that the average number of paths passing through a node on the widest level of Master will be the lowest for all levels. Deleting one node will thus, on the average, delete the fewest number of paths with it, which achieves our first goal of the pruning strategy.

Next we compute the weight distribution $d_N$ for every node $N$ on the widest level, and record $a_N = \min\{i | d_N[i] > 0\}$ for each node. Let $A = \max\{a_N\}$. The nodes on the widest level which have $a_N = A$ are the ones with only high-weight trails below them, and are eligible for deletion.

When deleting nodes from the widest level, some care has to be taken. First, deleting one node may trigger other deletions on adjacent levels, in order to keep the CRHS equation reduced. Hence we need to periodically check what $\mathcal{N}$ is, especially when $\mathcal{N}$ is getting close to $\sigma$, and abort as soon as $\mathcal{N} \leq \sigma$. Second, there might be many nodes on the widest level with $a_N = A$, and deleting all of them could lead to $\mathcal{N} \ll \sigma$. Third, the second-widest level in Master may only have slightly fewer nodes than the widest one, and may quickly become the widest once a few deletions have occurred. Ideally we would like to always delete nodes from the widest level. However, if two levels have approximately equally many nodes then we need to switch the level to delete from very often, with a re-computation of all the $d_N$ every time. This would increase the computational complexity from $\mathcal{O}(\mathcal{N})$ towards $\mathcal{O}(\mathcal{N}^2)$, which quickly becomes very inefficient.

Instead, we compromise by checking for widest level and recalculating the $d_N$-vectors once the widest level is reduced to 90% of the initial size of the second-widest level. The 90% has been decided somewhat arbitrarily, but works well in practice.

Combined, these choices dynamically try to keep as many trails as possible, and saves trails with a low count of active S-boxes. Moreover, always deleting from the widest level ensures that the number of nodes on different levels are somewhat balanced. Unless $\sigma$ is set very low (like, allowing only one node on every level), we do not risk emptying a level for nodes and therefore lose all the trails. Overall, we are therefore guaranteed to find trails with a relatively low number of active S-boxes, regardless of the number of rounds in $\varepsilon$.

This feature is in contrast to CryptaGraph, which has the number of *S-box patterns* as its guiding parameter for memory usage. If the number of S-box patterns is set too low, CryptaGraph will not return any trails at all, where "low" depends both on the cipher and the number of rounds.

### 4.6 Estimating Hull of Lowest Weight

Given one path $p$ in Master, it is easy to calculate the weight of the trail $\boldsymbol{u}$ that $p$ represents. Starting from level $n$ in Master, where the block $u_1$ starts, all sub-paths of $p$ of length $b$ will give all $u_{i,j}$, the outputs of all S-boxes in $\varepsilon$. From each $u_i$ it is possible to compute $L_i(u_i)$, and then to compute $w(\boldsymbol{u})$ as given by (1).

We are interested in finding the hull(s) $u_0 \diamond u_r$ with the lowest weight, but searching through all paths in Master will be infeasible as the number of paths is typically exponential in the block size $n$. We therefore need a more efficient algorithm for finding good input/output pairs $u_0/u_r$, for which we calculate the exact $w(u_0 \diamond u_r)$ for all trails remaining in Master that start with $u_0$ and end in $u_r$. We will again use the activity distributions as our foundation, as the complexity for computing $a(\boldsymbol{u})$ is linear in the number of nodes.

An added benefit from adhering to Invariants 1 and 2 when we solve the SoC is that every $n$ levels come from the same round, and every round is added in increasing order. This means that any path starting in node A on level $n$ and ending in the sink $T$ is a trail on the form $(u_1, \ldots, u_r)$, which gives all the output states from all the $\mathcal{S}_i$ in $\varepsilon$.

We begin the search for the best hull $u_0 \diamond u_r$ by calculating the activity distributions $d_A$ for all nodes $A$ on level $n$, as level $n$ is the beginning of $u_1$ in Master.

Let $a_A = \min\{i | d_A[i] > 0 \text{ and } i > 0\}$. Then $B = \min\{a_A\}$ is the fewest number of active S-boxes any trail in Master can have. Let $D$ be the set of all nodes $A$ which have $a_A = B$. Then all trails starting in the source and ending in a node in $D$ are input differences or masks for the plaintext block $P$ yielding path(s) with the lowest possible number of active S-boxes, and any one of these trails are candidates as the $u_0$ in our final best hull $u_0 \diamond u_r$.

Calculating the activity distributions from the sink $T$ to level $n$ has allowed us to identify the lowest number of active S-boxes any trail may have, as well as

which inputs to $\varepsilon$ may yield such a trail. However, we still do not know what the corresponding $u_r$ is. To do so, we need to know which input $u_0$ in $D$ is connected to which output $u_r$.

We do this by by making two adaptations to our algorithm for counting active S-boxes. The first one changes it such that we are able to start and end in arbitrary levels $l$, where $l = 0 \bmod b$. This is done by initializing every node $N$ on the level where the count starts with $d_N = (1, 0, 0, \ldots, 0)$. The algorithm continues recursively as normal from there on.

The second change is to make the activity distributions remember which nodes on the starting level they came from. We call these activity distributions for *node-to-node distributions*, as they give the activity distribution for the paths between two fixed nodes in the DAG.

We start counting node-to-node distributions on level $rn$, the level where the block $u_r$ starts, and end on level $n$. We can then go through each node in $D$ and see which nodes on level $rn$ they are connected to. For every pair of nodes $N_\alpha$ on level $n$ and $N_\beta$ on level $rn$ we have the node-to-node activity distribution for all paths starting in $N_\alpha$ and ending in $N_\beta$. We first filter the $(N_\alpha, N_\beta)$-pairs and only keep the pairs that have paths with the least number of active S-boxes between them. As every $N_\alpha$ specifies some $u_0$ and every $N_\beta$ specifies some $u_r$, these pairs form a set of $(u_0, u_r)$ candidates for the hull with the lowest weight.

Having found our set of $(u_0, u_r)$ candidates that give low $w(u_0 \diamond u_r)$, we need to estimate which of the pairs are most likely to yield the hull with the lowest weight. Ideally, we would like to calculate the weights for each hull generated by each pair, but for larger SoC's with many nodes and trails, this would be infeasible. Instead, we calculate the average weight, $k$, the non-zero entries in the base table BT contribute to the weight of a path (excluding $\mathrm{BT}[0][0] = 2^b$, which indicates a passive S-box). Finally, we use $k$ to make an estimate of $w(\boldsymbol{u})$ for each trail $\boldsymbol{u}$ between $N_\alpha$ and $N_\beta$ as

$$w(\boldsymbol{u}) \approx a(u_1, \ldots, u_r)k,$$

and sum up these estimates to find an estimate on $w(u_0 \diamond u_r)$.

We fix $(N_\alpha, N_\beta)$ as the pair of nodes that gives the lowest estimated $w(u_0 \diamond u_r)$. Finally, the actual weight of every path, or as many paths we can afford in the case this number is very big, between $N_\alpha$ and $N_\beta$ is calculated and summed up to give a lower bound on the true $w(u_0 \diamond u_r)$.

## 5 Results and Discussion

We have implemented the algorithm described in the previous sections, and made an easy-to-use tool that searches for hulls of differential or linear trails with the largest EDP or ELP. We have named the tool *PathFinder*, and it can be found at `https://github.com/Simula-UiB/CRHS`. PathFinder is written in Rust, and reuses implementations of the various block ciphers made for CryptaGraph [26].

### 5.1 Results and Comparison with CryptaGraph

We have run PathFinder on most of the same instances as tabled in [15] for comparison. The results are listed in Tables 1 and 2. The implementation was run on a Dell server with 96 CPUs and 192GB memory. PathFinder is not parallelized so only a single CPU was used, and due to other users using the same server we limited our memory consumption to max 32GB. It is worth noting that counting node-to-node distributions, searching for the optimal hull, consume more memory than the DAG itself. This is due to the fact that more metadata is required per node than during the pruning process.

For about half of the ciphers in Table 1 we get slightly worse ELP than CryptaGraph, and for the rest we get significantly lower ELP. Some of these can be explained by the fact that CryptaGraph can calculate over the complete hull, while PathFinder has to compute the weight of one trail at the time, and add them up. We have currently set an upper limit on $2^{26}$ trails in the sum, in order for the program to complete in a reasonable time. In other cases, like for Mantis or Midori, PathFinder finds more trails than CryptaGraph, but of higher weight. Almost all of the ciphers we have run PathFinder on have 64-bit blocks. However, PathFinder is not limitied to 64-bit block sizes. Rather, our choices of ciphers allow us to make direct comparison with CryptaGrap's portfolio of ciphers. If the soft limit is kept constant, increasing the block size from 64 to 128 bits means that the number of levels will double and the average number of nodes per levels is halved. Doubling the soft limit will counteract this. In other words, the memory requirement to keep the number of nodes per level unchanged grows linearly with the block size.

For differential trails the situation is different for some ciphers, where we get a higher EDP than CryptaGraph finds. There are in general fewer valid differential trails in a cipher than linear trails, and for Klein and Prince PathFinder is able to find some of of low weight that CryptaGraph misses. By investigating some example trails that PathFinder finds, we see that these are cases where there exists a round in the trails that have rather many active S-boxes, but still have few active S-boxes in total for the whole trail. For Klein with 5 and 6 rounds, the number of active S-boxes in each round of the example trails are $(1, 4, 7, 4, 1)$ and $(2, 3, 7, 4, 2, 3)$, respectively. As CryptaGraph will include all cipher states with 6 or fewer active S-boxes before including any with 7 active S-boxes, the number of S-box patterns must be set very high for CryptaGraph to incorporate these trails in its search space. The complete example trails for Klein are listed in Appendix A.

For Prince with 6 rounds, we see the same phenomenon. The trails PathFinder finds with the lowest weight overall have rounds containing 6 active S-boxes. These have probably been missed by CryptaGraph since the number of S-box patterns must be set very high to include states with 6 active S-boxes. The number of active S-boxes in every round of the example trail provided is $(2, 2, 6, 6, 2, 2)$ and can be found in Appendix B. In [7], the designers of Prince give a theorem saying that four consecutive rounds in a trail must contain at least 16 active S-boxes. We see that the 6-round trail PathFinder found meets

| Cipher (Total Rounds, block size) | Rounds | Soft Lim | Hull Size (Used, Found) | ELP | CG result |
|---|---|---|---|---|---|
| AES | 3 | $2^{16}$ | 2, 2 | $2^{-136.77}$ | $2^{-53.36}$ |
| (10, 128) | 4 | $2^{16}$ | 1, 1 | $2^{-243.26}$ | $2^{-147.88}$ |
| EPCBC-48 | 15 | $2^{18}$ | $2^{26}$, $2^{27.83}$ | $2^{-46.57}$ | $2^{-43.74}$ |
| (32, 48) | 16 | $2^{18}$ | $2^{26}$, $2^{29.63}$ | $2^{-49.71}$ | $2^{-46.77}$ |
| EPCBC-96 | 31 | $2^{18}$ | $2^{26}$, $2^{32.83}$ | $2^{-100.50}$ | $2^{-94.47}$ |
| (32, 96) | 32 | $2^{18}$ | $2^{26}$, $2^{31.67}$ | $2^{-102.48}$ | $2^{-97.59}$ |
| FLY | 8 | $2^{16}$ | 5, 9 | $2^{-82.99}$ | $2^{-54.83}$ |
| (20, 64) | 9 | $2^{16}$ | 1, 6 | $2^{-86.00}$ | $2^{-63.00}$ |
| GIFT-64 | 11 | $2^{18}$ | 2, 2 | $2^{-59.00}$ | $2^{-55.00}$ |
| (28, 64) | 12 | $2^{18}$ | 2, 2 | $2^{-69.00}$ | $2^{-64.00}$ |
| KHAZAD | 2 | $2^{16}$ | 1, 1 | $2^{-44.21}$ | $2^{-37.97}$ |
| (8, 64) | 3 | $2^{16}$ | 1, 1 | $2^{-90.00}$ | $2^{-68.01}$ |
| KLEIN | 5 | $2^{18}$ | 6, 6 | $2^{-52.25}$ | $2^{-46.00}$ |
| (12, 64) | 6 | $2^{18}$ | 44, 50 | $2^{-70.16}$ | $2^{-66.00}$ |
| LED | 4 | $2^{18}$ | 4, 8 | $2^{-72.91}$ | $2^{-48.68}$ |
| (32, 64) | | | | | |
| $\text{MANTIS}_7$ | $2 \cdot 4$ | $2^{18}$ | $2^{17.45}$, $2^{18.64}$ | $2^{-109.61}$ | $2^{-49.05}$ |
| $(2 \cdot 8, 64)$ | | | | | |
| MIDORI64 | 6 | $2^{18}$ | $2^{21.62}$, $2^{23.89}$ | $2^{-85.03}$ | $2^{-53.02}$ |
| (16, 64) | 7 | $2^{18}$ | $2^{26}$, $2^{29.66}$ | $2^{-108.42}$ | $2^{-62.88}$ |
| PRESENT | 23 | $2^{18}$ | $2^{26}$, $2^{37.03}$ | $2^{-69.23}$ | $2^{-61.00}$ |
| (31, 64) | 24 | $2^{18}$ | $2^{26}$, $2^{38.60}$ | $2^{-73.23}$ | $2^{-63.61}$ |
| | 25 | $2^{18}$ | $2^{26}$, $2^{39.65}$ | $2^{-76.54}$ | $2^{-66.21}$ |
| PRIDE | 15 | $2^{18}$ | 1, 1 | $2^{-58.00}$ | $2^{-58.00}$ |
| (20, 64) | 16 | $2^{18}$ | 7, 7 | $2^{-65.99}$ | $2^{-63.99}$ |
| PRINCE | $2 \cdot 3$ | $2^{18}$ | 19, 19 | $2^{-55.57}$ | $2^{-54.00}$ |
| $(2 \cdot 6, 64)$ | $2 \cdot 4$ | $2^{18}$ | 214, 214 | $2^{-92.90}$ | $2^{-63.82}$ |
| PUFFIN | 32 | $2^{18}$ | $2^{26}$, $2^{52.55}$ | $2^{-83.69}$ | $2^{-51.90}$ |
| (32, 64) | | | | | |
| QARMA | $2 \cdot 3$ | $2^{18}$ | 612, 1433 | $2^{-95.75}$ | $2^{-53.71}$ |
| $(2 \cdot 8, 64)$ | | | | | |
| RECTANGLE | 12 | $2^{18}$ | $2^{16.66}$, $2^{16.66}$ | $2^{-56.75}$ | $2^{-52.27}$ |
| (25, 64) | 13 | $2^{18}$ | $2^{17.16}$, $2^{17.16}$ | $2^{-64.22}$ | $2^{-58.14}$ |
| | 14 | $2^{18}$ | $2^{16.51}$, $2^{16.51}$ | $2^{-68.48}$ | $2^{-62.98}$ |
| SKINNY-64 | 8 | $2^{18}$ | $2^{26}$, $2^{27.51}$ | $2^{-113.81}$ | $2^{-50.46}$ |
| (32, 64) | 9 | $2^{18}$ | $2^{26}$, $2^{37.55}$ | $2^{-143.15}$ | $2^{-69.83}$ |

Table 1: Details on hulls and ELP found by PathFinder for various ciphers. Hull size indicates both the total number of trails found in the hull, and the number of trails used for calculating the ELP. CryptaGraph's results are in the last column for comparison.

this bound with equality; the number of active S-boxes in any four consecutive rounds is 16. The Prince designers use this bound to deduce that any trail of the

| Cipher (Total Rounds, block size) | Rounds | Soft Lim | Hull Size (Used, Found) | EDP | CG result |
|---|---|---|---|---|---|
| AES | 3 | $2^{16}$ | 1, 1 | $2^{-130.00}$ | $2^{-54.00}$ |
| (10, 128) | 4 | $2^{16}$ | 1, 1 | $2^{-179.00}$ | $2^{-150.00}$ |
| EPCBC-48 | 13 | $2^{18}$ | 356, 356 | $2^{-48.84}$ | $2^{-43.86}$ |
| (32, 48) | 14 | $2^{18}$ | 531, 531 | $2^{-53.46}$ | $2^{-47.65}$ |
| EPCBC-96 | 20 | $2^{18}$ | 21, 21 | $2^{-94.62}$ | $2^{-92.73}$ |
| (32, 96) | 21 | $2^{18}$ | 20, 20 | $2^{-102.90}$ | $2^{-97.78}$ |
| FLY | 8 | $2^{20}$ | 180, 104 | $2^{-59.0}$ | $2^{-55.76}$ |
| (20, 64) | 9 | $2^{20}$ | 76, 76 | $2^{-82.63}$ | $2^{-63.35}$ |
| GIFT-64 | 12 | $2^{18}$ | 10, 10 | $2^{-57.81}$ | $2^{-56.57}$ |
| (28, 64) | 13 | $2^{18}$ | 5, 15 | $2^{-63.19}$ | $2^{-60.42}$ |
| KHAZAD | 2 | $2^{16}$ | 1, 1 | $2^{-47.49}$ | $2^{-45.42}$ |
| (8, 64) | 3 | $2^{20}$ | 1, 1 | $2^{-79.66}$ | $2^{-81.66}$ |
| KLEIN | 5 | $2^{18}$ | 8, 8 | $2^{-44.39}$ | $2^{-45.91}$ |
| (12, 64) | 6 | $2^{22}$ | 4, 4 | $2^{-55.25}$ | $2^{-69.00}$ |
| LED | 4 | $2^{22}$ | 6, 18 | $2^{-55.61}$ | $2^{-49.42}$ |
| (32, 64) | | | | | |
| $\text{MANTIS}_7$ | $2 \cdot 4$ | $2^{22}$ | $2^{24.94}$, $2^{26.64}$ | $2^{-100.87}$ | $2^{-47.98}$ |
| $(2 \cdot 8, 64)$ | | | | | |
| MIDORI64 | 6 | $2^{22}$ | $2^{20.28}$, $2^{21.50}$ | $2^{-63.60}$ | $2^{-52.37}$ |
| (16, 64) | 7 | $2^{22}$ | $2^{23.82}$, $2^{25.49}$ | $2^{-71.75}$ | $2^{-61.22}$ |
| PRESENT | 15 | $2^{18}$ | $2^{15.42}$, $2^{15.42}$ | $2^{-65.69}$ | $2^{-58.00}$ |
| (31, 64) | 16 | $2^{18}$ | $2^{15.97}$, $2^{16.29}$ | $2^{-69.71}$ | $2^{-61.80}$ |
| | 17 | $2^{18}$ | $2^{17,76}$, $2^{17.76}$ | $2^{-74.87}$ | $2^{-63.52}$ |
| PRIDE | 15 | $2^{22}$ | 1, 1 | $2^{-58.00}$ | $2^{-58.00}$ |
| (20, 64) | 16 | $2^{22}$ | 1, 1 | $2^{-64.00}$ | $2^{-63.99}$ |
| PRINCE | $2 \cdot 3$ | $2^{22}$ | 16, 20 | $2^{-49.45}$ | $2^{-55.91}$ |
| $(2 \cdot 6, 64)$ | $2 \cdot 4$ | $2^{22}$ | 36, 36 | $2^{-80.67}$ | $2^{-67.32}$ |
| PUFFIN | 32 | $2^{18}$ | $2^{26}$, $2^{37.25}$ | $2^{-79.71}$ | $2^{-59.63}$ |
| (32, 64) | | | | | |
| QARMA | $2 \cdot 3$ | $2^{18}$ | 5, 5 | $2^{-97.48}$ | $2^{-56.47}$ |
| $(2 \cdot 8, 64)$ | | | | | |
| RECTANGLE | 13 | $2^{18}$ | 166, 166 | $2^{-58.37}$ | $2^{-55.64}$ |
| (25, 64) | 14 | $2^{18}$ | 57, 171 | $2^{-62.60}$ | $2^{-60.64}$ |
| | 15 | $2^{18}$ | 388, 388 | $2^{-70.63}$ | $2^{-65.64}$ |
| SKINNY-64 | 8 | $2^{18}$ | $2^{18.74}$, $2^{20.14}$ | $2^{-113.70}$ | $2^{-50.72}$ |
| (32, 64) | 9 | $2^{18}$ | $2^{22.50}$, $2^{23.74}$ | $2^{-126.91}$ | $2^{-69.64}$ |

Table 2: Details on hulls and EDP found by PathFinder for various ciphers. Hull size indicates both the total number of trails found in the hull, and the number of trails used for calculating the EDP. CryptaGraph's results are in the last column for comparison.

full 12-round Prince must have at least 48 active S-boxes, and hence be secure against differential and linear attacks. We ran PathFinder on the full 12-round

Prince cipher, and interestingly enough it turns out that there indeed do exist differential trails in the full cipher with exactly 48 active S-boxes. So we find that the lower bound for full Prince is met with equality (example trail given in Appendix B). The EDP PathFinder gives for 12-round Prince, which has a 128-bit key, is $2^{-124.06}$. While this does not lead to a valid differential attack since Prince only has a 64-bit block, it does give a differential probability that is higher than $2^{-128}$.

### 5.2 Combining PathFinder and CryptaGraph

PathFinder has several similarities to CryptaGraph. Both are tools with a simple command line interface. In either of them the user specifies cipher, number of rounds and memory limits, and the tool returns good differential or linear trails with an estimate on the probability or the bias of the hull they belong to. Both of them exploit the fact that a DAG with a relatively small number of nodes may contain exponentially (in the number of nodes) many paths. Hence encoding information as paths in the DAG lets us handle very large data sets. Both CryptaGraph and PathFinder encodes trails of a cipher as paths in a DAG.

The difference between them comes from the underlying graphs used in the two tools. Each node in CryptaGraph represents a particular cipher state (of $n$ bits), and an edge is the transition from one state to a possible next state, as given by the base table. In PathFinder we make the full step and let the cipher states themselves also be encoded as paths (of length $n$). This means PathFinder can handle many more states at a particular point in a cipher than CryptaGraph can. This is maybe best illustrated in Figure 4a, where PathFinder's initial DAG of $n$ nodes contains all $2^n$ possible plaintext states while CryptaGraph would need $2^n$ nodes to do the same.

This difference leads to the tools having different features which complement each other. The strength of CryptaGraph is its ability to calculate the weight of a hull. Even if CryptaGraph's DAG contains an exponential number of paths representing trails belonging to the same hull, CryptaGraph can efficiently compute the sum of weights of each trail. PathFinder can not do this in a similar way, since an edge in PathFinder's DAG does not represent a transition between two individual states. Hence PathFinder computes the weight of each trail in a hull individually, and can not efficiently sum up the weights of an exponential number of trails in a hull.

The weakness of CryptaGraph is the limited set of states it can handle in each round of a cipher. There are few ways of telling beforehand which states that will be present in the best trails, except that they will probably have few active S-boxes. So CryptaGraph's strategy for selecting states (i.e. nodes) for its DAG is simply to take the ones with the highest probabilities or biases for going from one state to the next. In practice this resolves to the states with the least number of active S-boxes. But this means that every state in a trail that CryptaGraph returns must come from this limited set of states, otherwise CryptaGraph finds nothing. This problem is partially resolved by the technique of *anchoring*, which is to greatly expand the set of states for the first and last

round in a cipher. However, this does not help if the state with many active S-boxes occur in the middle of the cipher, like for Klein and Prince.

PathFinder on the other hand has no such limitations, and starts with the complete set of $2^n$ states. Eventually the pruning of nodes will delete states in PathFinder as well, but we do not need to define which states to keep and which to discard. Instead this is done dynamically at run-time, guided by keeping the states with the lowest number of active S-boxes. The pruning strategy ensures that there will always be many valid trails encoded in PathFinder's DAG, and that it will never return empty-handed.

For further work in this direction we therefore propose to combine the two tools in a way that plays to each others strengths. This is beyond the scope of the current work, but the idea is as follows:

1. Run PathFinder to find a set of states that actually occur in the best trails.
2. Run CryptaGraph, where the set of nodes in CryptaGraph's DAG represents this particular set of states.

Letting PathFinder guide CryptaGraph's set of states in this way will ensure that CryptaGraph will find the same trails as PathFinder, but we can then exploit CryptaGraph's better calculation of hull weights.

## 6 Conclusion

Using graphs for finding linear and differential trails in ciphers is a new direction in cryptanalysis. The strength of directed acyclic graphs is that they can contain exponentially (in the number of nodes) many paths. Hence representing the data we are interested in as paths in a DAG may allow us to efficiently search an exponentially big search space. The work done in [15] started this with CryptaGraph, and in this paper we have followed up with complementary work in the same direction.

Our work complements that in [15] and uses the paths in the DAG in a different way. By representing the DDT or LAT of an S-box as a CRHS equation, we can use existing methodology for solving a system of CRHS equations to construct a DAG containing trails we are interested in. One general problem with solving systems of CRHS equations is its memory complexity. We overcome this problem by pruning the graph when it grows too big, thus controlling the memory consumption. We have presented a pruning strategy that keeps the most promising trails contained the graph while discarding the rest. This allows us to find other good trails than CryptaGraph finds, and do the search for an arbitrary number of rounds.

Both methods have been implemented as easy-to-use and compatible tools, where only a reference implementation of a cipher is needed in order to do the trail search. The same reference implementation made for CryptaGraph can be used on PathFinder, and in fact PathFinder already reuses Cryptagraph's portfolio of cipher implementations. It has been well understood for two decades how to make ciphers secure against differential or linear cryptanalysis, but designers

always need to take these types of attacks into account when proposing a new cipher. These tools can help in the design process.

We have compared CryptaGraph and PathFinder, and looked at strengths and limitations of both. For further work, we suggest to combine the two into one, in a way that exploits the strong parts of both.

## References


1. M. Albrecht, L. Grassi, C. Rechberger, A. Roy, and T. Tiessen. MiMC: Efficient Encryption and Cryptographic Hashing with Minimal Multiplicative Complexity. In J. H. Cheon and T. Takagi, editors, *Advances in Cryptology – ASIACRYPT 2016*, pages 191–219, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.
2. M. R. Albrecht, C. Rechberger, T. Schneider, T. Tiessen, and M. Zohner. Ciphers for MPC and FHE. In E. Oswald and M. Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015*, pages 430–454, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.
3. R. Ankele and S. Kölbl. Mind the Gap - A Closer Look at the Security of Block Ciphers against Differential Cryptanalysis. In C. Cid and M. J. Jacobson Jr., editors, *Selected Areas in Cryptography – SAC 2018*, pages 163–190. Springer International Publishing, 2019.
4. R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers. The SIMON and SPECK Lightweight Block Ciphers. In *Proceedings of the 52nd Annual Design Automation Conference*, DAC '15, New York, NY, USA, 2015. Association for Computing Machinery.
5. E. Biham and A. Shamir. Differential Cryptanalysis of DES-like Cryptosystems. *J. Cryptol.*, 4(1):3—-72, 1991.
6. A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin, and C. Vikkelsoe. PRESENT: An Ultra-Lightweight Block Cipher. In P. Paillier and I. Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems - CHES 2007*, pages 450–466, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.
7. J. Borghoff, A. Canteaut, T. Güneysu, E. B. Kavun, M. Knezevic, L. R. Knudsen, G. Leander, V. Nikov, C. Paar, C. Rechberger, P. Rombouts, S. S. Thomsen, and T. Yalçın. PRINCE – A Low-Latency Block Cipher for Pervasive Computing Applications. In X. Wang and K. Sako, editors, *Advances in Cryptology – ASIACRYPT 2012*, pages 208–225, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
8. R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.*, 35(8):677—-691, 1986.
9. A. Canteaut, S. Carpov, C. Fontaine, T. Lepoint, M. Naya-Plasencia, P. Paillier, and R. Sirdey. Stream Ciphers: A Practical Solution for Efficient Homomorphic-Ciphertext Compression. In *Revised Selected Papers of the 23rd International Conference on Fast Software Encryption - Volume 9783*, FSE 2016, pages 313—-333. Springer-Verlag, 2016.
10. J. Daemen and V. Rijmen. *The Design of Rijndael*. Springer-Verlag Berlin Heidelberg, 2012.
11. C. De Cannière, O. Dunkelman, and M. Knežević. KATAN and KTANTAN — A Family of Small and Efficient Hardware-Oriented Block Ciphers. In C. Clavier and K. Gaj, editors, *Cryptographic Hardware and Embedded Systems - CHES 2009*, pages 272–288, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

12. C. Dobraunig, M. Eichlseder, L. Grassi, V. Lallemand, G. Leander, E. List, F. Mendel, and C. Rechberger. Rasta: A Cipher with Low ANDdepth and Few ANDs per Bit. In H. Shacham and A. Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018*, pages 662–692. Springer International Publishing, 2018.
13. K. Fu, M. Wang, Y. Guo, S. Sun, and L. Hu. MILP-Based Automatic Search Algorithms for Differential and Linear Trails for Speck. In T. Peyrin, editor, *Fast Software Encryption - 23rd International Conference, FSE 2016, Bochum, Germany, March 20-23, 2016, Revised Selected Papers*, volume 9783 of *Lecture Notes in Computer Science*, pages 268–288. Springer, 2016.
14. Z. Gong, S. Nikova, and Y. W. Law. KLEIN: A New Family of Lightweight Block Ciphers. In A. Juels and C. Paar, editors, *RFID. Security and Privacy*, pages 1–18, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
15. M. Hall-Andersen and P. S. Vejre. Generating Graphs Packed with Paths Estimation of Linear Approximations and Differentials. *IACR Transactions on Symmetric Cryptology*, 2018(3):265–289, Sep. 2018.
16. S. Kölbl, G. Leander, and T. Tiessen. Observations on the SIMON Block Cipher Family. In R. Gennaro and M. Robshaw, editors, *Advances in Cryptology – CRYPTO 2015*, pages 161–185, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.
17. M. Matsui. Linear Cryptanalysis Method for DES Cipher. In T. Helleseth, editor, *Advances in Cryptology — EUROCRYPT '93*, pages 386–397, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.
18. M. Matsui. On correlation between the order of S-boxes and the strength of DES. In A. De Santis, editor, *Advances in Cryptology — EUROCRYPT'94*, pages 366–375, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.
19. P. Méaux, A. Journault, F.-X. Standaert, and C. Carlet. Towards Stream Ciphers for Efficient FHE with Low-Noise Ciphertexts. In M. Fischlin and J.-S. Coron, editors, *Advances in Cryptology – EUROCRYPT 2016*, pages 311–343, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.
20. N. Mouha and B. Preneel. Towards Finding Optimal Differential Characteristics for ARX: Application to Salsa20. Cryptology ePrint Archive, Report 2013/328, 2013. `https://eprint.iacr.org/2013/328`.
21. N. Mouha, Q. Wang, D. Gu, and B. Preneel. Differential and Linear Cryptanalysis Using Mixed-Integer Linear Programming. In C.-K. Wu, M. Yung, and D. Lin, editors, *Information Security and Cryptology*, pages 57–76, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
22. H. Raddum and O. Kazymyrov. Algebraic Attacks Using Binary Decision Diagrams. In B. Ors and B. Preneel, editors, *Cryptography and Information Security in the Balkans*, pages 40–54. Springer International Publishing, 2015.
23. T. E. Schilling and H. Raddum. Solving Compressed Right Hand Side Equation Systems with Linear Absorption. In T. Helleseth and J. Jedwab, editors, *Sequences and Their Applications – SETA 2012*, pages 291–302, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
24. T. Shirai, K. Shibutani, T. Akishita, S. Moriai, and T. Iwata. The 128-Bit Blockcipher CLEFIA (Extended Abstract). In A. Biryukov, editor, *Fast Software Encryption*, pages 181–195, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.
25. S. Sun, L. Hu, P. Wang, K. Qiao, X. Ma, and L. Song. Automatic Security Evaluation and (Related-key) Differential Characteristic Search: Application to SIMON, PRESENT, LBlock, DES(L) and Other Bit-Oriented Block Ciphers. In P. Sarkar and T. Iwata, editors, *Advances in Cryptology – ASIACRYPT 2014*, pages 158–178, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

26. P. Vejre. Cryptagraph, 2018. `https://gitlab.com/psve/cryptagraph`.
27. J. Yin, C. Ma, L. Lyu, J. Song, G. Zeng, C. Ma, and F. Wei. Improved Cryptanalysis of an ISO Standard Lightweight Block Cipher with Refined MILP Modelling. In X. Chen, D. Lin, and M. Yung, editors, *Information Security and Cryptology*, pages 404–426. Springer International Publishing, 2018.
28. H. Zhao, G. Han, L. Wang, and W. Wang. MILP-Based Differential Cryptanalysis on Round-Reduced Midori64. *IEEE Access*, 8:95888–95896, 2020.

## A Low-weight differential trails for Klein

The following differential trail for 5-round Klein has probability $2^{-51}$:

```
     Example trail (in hex):
MSB          LSB
000000000000000b Alpha
S-box Layer
0000000000000004
Linear Layer
000000000c080404
S-box Layer
0000000007060603
Linear Layer
010f040b09000509
S-box Layer
080c040404000a0e
Linear Layer
0000080c02020000
S-box Layer
0000090d0b0e0000
Linear Layer
0000000100000000
S-box Layer
0000000800000000 Beta
```

The following differential trail for 6-round Klein has probability $2^{-57}$:

```
     Example trail (in hex):
MSB          LSB
0000050000050000 Alpha
S-box Layer
0000020000020000
Linear Layer
0600040200000000
S-box Layer
0100030500000000
Linear Layer
```

```
0909060001030201
S-box Layer
080e040004040a0e
Linear Layer
080c000000000604
S-box Layer
0b0d000000000809
Linear Layer
000000000d0a0000
S-box Layer
0000000002060000
Linear Layer
04000e0e00000000
S-box Layer
0100030300000000 Beta
```

## B Trails for Prince

The following differential trail for 6-round Prince has probability $2^{-53}$:

```
Example trail (in hex):
MSB          LSB
0000000000000101 Alpha
S-box Layer
0000000000000808
Linear Layer
0008000008000000
S-box Layer
0008000004000000
Linear Layer
8040040840800000
S-box Layer
8080040450500000
Middle involution
8080040450500000
S-box Layer
8040040840800000
Linear Layer
0008000004000000
S-box Layer
0008000008000000
Linear Layer
0000000000000808
S-box Layer
0000000000000101 Beta
```

The following 12-round differential trail for Prince has 48 active S-boxes:

```
Example trail (in hex):
MSB          LSB
0004000008000000 Alpha
S-box Layer
0004000002000000
Linear Layer
4020020400000402
S-box Layer
8080010100000808
Linear Layer
8108000008810000
S-box Layer
8808000004440000
Linear Layer
0000000040800000
S-box Layer
0000000080800000
Linear Layer
0000080000000008
S-box Layer
0000040000000008
Linear Layer
0408408000008040
S-box Layer
0808404000008080
Middle involution
0808404000008080
S-box Layer
0408408000008040
Linear Layer
0000040000000008
S-box Layer
0000080000000008
Linear Layer
0000000080800000
S-box Layer
0000000040800000
Linear Layer
8808000004440000
S-box Layer
8408000008840000
Linear Layer
8080040400000808
S-box Layer
```

8010010800000801
Linear Layer
0080000000100000
S-box Layer
0080000000400000 Beta