Boolean Polynomials, BDDs and CRHS Equations - Connecting the Dots with CryptaPath

John Petter Indrøy, Nicolas Costes, and Håvard Raddum

Simula UiB, Bergen, Norway

Abstract. When new symmetric-key ciphers and hash functions are proposed they are expected to document resilience against a number of known attacks. Good, easy to use tools may help designers in this process and give improved cryptanalysis. In this paper we introduce CryptaPath, a tool for doing algebraic cryptanalysis which utilizes Compressed Right-Hand Side (CRHS) equations to attack SPN ciphers and sponge constructions. It requires no previous knowledge of CRHS equations to be used, only a reference implementation of a primitive.
The connections between CRHS equations, binary decision diagrams and Boolean polynomials have not been described earlier in literature. A comprehensive treatment of these relationships is made before we explain how CryptaPath works. We then describe the process of solving CRHS equation systems while introducing a new operation, dropping variables.

Keywords: algebraic cryptanalysis · binary decision diagram · equation system · block cipher · tool · open source

1 Introduction

It is not enough to simply propose a new design for symmetric ciphers. Alongside the design, there must be design rationale and security evaluation which describe how this design is resistant against attacks. This can be quite a laborious task, even if one includes only the most common attacks. As attack vectors are becoming more and more complex, experience and good intuition is important while designing the cipher. We therefore recognize the need of some sort of tool for assisting researchers designing a new symmetric primitive, which allows for automated analysis, enabling efficient testing of alternatives and leading to informed decisions. Ideally, this tool would cover all the most common attack techniques. That would be a large undertaking, and this ambition needs to be divided into several projects.

Fortunately, this is also recognized by other researchers, and an automated tool to use with linear and differential cryptanalysis has already been published: CryptaGraph [14]. We wish to add to this contribution by proposing a tool for algebraic cryptanalysis. There are many algebraic attacks, like Gröbner base computations, SAT-solving and interpolation attacks. We decided to go for Compressed Right-Hand Sides (CRHS) due to their compact representation of a set of binary vectors and the promising results for solving non-linear equation systems in [18,23,31]. Our tool is named CryptaPath, as we have drawn inspiration from CryptaGraph. The name is not the only similarity; with only small adjustments a reference implementation made for CryptaPath can be used with CryptaGraph and vice versa. A difference from CryptaGraph is that our tool also extends to sponge constructions.

Algebraic Cryptanalysis The first step of an algebraic attack is to convert the primitive into a system of equations. Next, we try to solve this system. If the complexity of solving such a system is lower than the complexity of the brute force attack, the cipher is considered broken.

When designing new ciphers, the focus is often on defending against linear and differential attacks. This was also the case for PURE, a variant of the KN cipher [22]. The KN cipher is provably secure against differential cryptanalysis. PURE was broken by an interpolation attack in [22]. In [21], a combined attack using differential paths and an (minimally modified) of-the-shelf SAT solver was able to generate full collisions for the hash functions MD4 and MD5. Last year, a successful Gröbner basis attack against Jarvis and Friday was presented [2]. This goes to show that algebraic cryptanalysis can be efficient on symmetric primitives.

There are various ways to model a cipher as a system of equations, and subsequently attack the cipher via trying to solve the system:

- *SAT solving* first converts the cipher into a Boolean formula, and then tries to find values to the arguments such that the formula evaluate to true [21,30].
- A *Gröbner basis* is a particular kind of generating set of an ideal in a polynomial ring. Finding a Gröbner basis is the crux of this attack. Well-known Gröbner basis finding algorithms are F4 [9] and F5 [10].
- *Compressed Right-Hand Sides* equations models the cipher as a system of linear equations with multiple right-hand sides. The hard problem here is to identify only the few right-hand side vectors which yield a consistent system of linear equations [23,26].

The solution to any of these systems of equations will contain the secret values we are looking for, i.e. the secret key of a symmetric cipher, or a pre-image for a hash function.

Existing research tools Our work focuses extensively on the correspondence between polynomials in the Boolean polynomial ring and binary decision diagrams (BDD). PolyBoRi [4] is an existing framework that has the exact same focus. However, PolyBoRi's way to represent polynomials using BDDs differs from ours. While PolyBoRi associates one monomial with every path in the BDD, we associate paths with the assignment of values to the variables themselves. This difference will become clear in Section 2.2.

There exist many tools for BDD manipulation [12,20,32,15,29], the most utilized one probably being CUDD [29]. Unfortunately, none of them suits our needs. We decided to make our own implementation of CRHS equations using Rust. Rust is fast and memory-efficient, with memory-safe and thread-safe guarantees and many classes of bugs being eliminated at compile time.

1.1 Our contribution

We propose a new tool called CryptaPath for assisted algebraic cryptanalysis using the CRHS representation. CryptaPath allows for algebraic analysis of any symmetric primitive that can be described as an SPN structure, such as most block ciphers, and sponge constructions. Running this tool on an SPN block cipher takes a single plaintext – ciphertext pair, converts it into a system of CRHS equations, and then tries to solve the system. If successful, it will return all solutions to the system, including all keys transforming the given plaintext into the given ciphertext. In the case of a sponge-based hash function, the tool will take in a hash digest, and try to find a matching pre-image. The researcher is only required to provide a reference implementation for CryptaPath to work, but may choose to dive deeper under the hood of the analysis if desired.

The caveat is the amount of memory required to launch a successful attack. For this reason, we have included the possibility of fixing bits in the key or pre-image. This allows CryptaPath to solve systems in practice. The number of rounds in the primitive is also a parameter which is possible to vary.

This tool builds on theory developed over several decades. CHRS equations can be described as a unification of MRHS equations [24] and BDDs. Earlier work describes how CRHS equation systems can be solved, but a thorough explanation of the relationships between Boolean polynomials in algebraic normal form, BDDs and CRHS equations has not been made before. We address this gap in literature in Section 2.

In addition, we have included a novel operation to the toolbox of CRHS: *dropping variables*. Dropping of variables is a technique which allows the solver to reduce the size of the system, and thus to save space. This operation comes with its own caveat, see Section 4.2 for details.

Finally, the source code of CryptaPath is available at
`https://github.com/Simula-UiB/CryptaPath`.

2 Preliminaries

Algebraic attacks are attacks where a cipher is represented as a system of equations and one tries to break the cipher by solving the system. While it is well known that the general MQ-problem is NP-hard [11], it is less known how to argue convincingly that a system of equations representing one particular cipher specification *must* be hard to solve. If the equation system is represented as Boolean polynomials in algebraic normal form (ANF) one may try to estimate the minimal degree a Gröbner base solver will reach before producing linear forms, and then give a lower bound on the attack complexity based on that. However, there can always be other ways of representing the equations, giving systems that are easier to solve. In this paper we use the CRHS representation, and start by explaining the correspondence between binary decision diagrams and multivariate polynomials in the Boolean polynomial ring $\mathbb{F}_2[x_0,\ldots,x_{n-1}]/(x_0^2+x_0,\ldots,x_{n-1}^2+x_{n-1})$.

2.1 Binary Decision Diagrams and Boolean Functions

A *Binary Decision Diagram* (BDD) is an efficient way to represent and evaluate Boolean functions [5]. Boolean functions have numerous use cases, with examples found in computer assisted design [6], network analysis [16], formal verification [6], artificial intelligence, risk assessment [13], cryptology [23,26], and more.

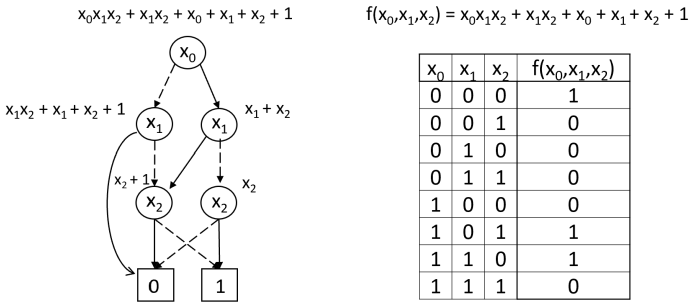
A BDD is a rooted, directed acyclical graph (DAG), with labeled nodes. There are two kinds of nodes, *decision nodes* and *terminal nodes*. A terminal node is labeled either with the value 0 or 1, while each decision node N is labeled by a Boolean variable x_i. A decision node has two children, often called the *low child* and the *high child*. The edge from decision node N to its low (high) child represents an assignment of the associated Boolean variable x_i to 0 (1). These edges are drawn as dashed (solid) lines in all figures.

To construct a BDD representing a given Boolean function $f(x_0,\ldots,x_{n-1})$, we start with the root node and associate f to it. Choose a variable from f, say x_0, as the decision variable, or label, for the root node and create its low and high child. Associate $f(0,x_1,\ldots,x_{n-1})$ with the low child and $f(1,x_1,\ldots,x_{n-1})$ with the high child. Continue recursively from each of the children by deciding on the next variable, then creating more decision nodes associated with polynomials made from partial assignments to f. If several nodes get associated to the same polynomial they will be merged into one. In the end the last variable gets fixed, so the only two nodes created at the bottom will be the terminal nodes 0 and 1.

Conversely, to find the ANF of the Boolean function associated to a given BDD we start with the terminal nodes 0 and 1 and find the ANFs associated to the nodes by going upwards in the BDD. Assume a decision node N decides on variable x_i and that the ANFs corresponding to its low and high children have already been computed as g_0 and g_1, respectively. By the theory of Shannon expansion [28], the ANF of N will then be $x_ig_1+(x_i+1)g_0$. Recursively computing ANFs for the nodes in the BDD this way will eventually compute the ANF f associated with the root node. This f will be the ANF of the Boolean function associated with the BDD.

Figure 1 shows a small example of a BDD with the ANFs associated to each node. The ANF associated to the root node is the Boolean function associated with the complete BDD.

Following a path from the root node through the BDD can therefore be viewed as assigning values to the arguments of a Boolean function, and the value of the function for those assignments is given by the terminal node in which the path ends. Each variable x_i can only occur once on any path of the BDD. Every decision node has two children, so all possible assignments are present as paths. The BDD therefore encodes the complete truth table of a Boolean function



(a) (RO)BDD, with associated (sub-) Boolean functions.

$f(x_0,x_1,x_2) = x_0x_1x_2 + x_1x_2 + x_0 + x_1 + x_2 + 1$

x_0	x_1	x_2	$f(x_0,x_1,x_2)$
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

(b) ANF and truth table.

Fig. 1: Example of BDD, ANF and the truth table for a Boolean function. Dashed lines represent 0-assignments, solid lines represent 1-assignments.

associated with the BDD. If we encounter the Boolean variables in the same order for each path in the BDD, we say that the BDD is *ordered*. The size of the BDD (i.e., its number of nodes) may be sensitive to the order we choose for the variables. Finding the optimal order of variables is an NP-hard problem [3]. Because a BDD utilizes a DAG, evaluating the Boolean function can be done very efficiently: in n steps or less, where n is the number of variables of the Boolean function.

Size wise, truth tables, Karnaugh maps and other classical representations of Boolean functions grow exponentially with the number of variables involved. There exist more practical approaches where its size is dependent on the Boolean function it represents, and where sub-exponential growth is possible. BDDs fall into this category.

Another desirable property of BDDs, is that a BDD can be reduced to a canonical representation, i.e. for every function there exists a unique BDD representing it, up to the ordering of variables, which has a minimal number of nodes. A BDD in this state is called *reduced* (see [5, Sec. 4.2]).

BDDs may also be understood as a compressed representation of sets or relations, where operations are executed directly on this compressed representation. This view is closer to how we use and understand BDDs in terms of CRHS equations.

2.2 Compressed Right-Hand Sides and Boolean Equations

We use reduced ordered BDDs (ROBDDs) as the fundamental building block of Compressed Right-Hand Side equations. As they are, ROBDDs are too strict in its definition for us to use them the way we would like. We will therefore redefine some of the rules regarding ROBDDs, and call them Compressed Right-Hand Side equations. The changes we make consists of one minor generalization, and two major changes to the definition of ROBDDs, “transforming” them into CRHS equations:

First, we divide the ordered BDD into levels where each level has nodes of only the same Boolean variable. This allows us to generalize the notation slightly, by associating the decision variable with a level instead of individual nodes. Second, we have only one terminal node, the 1-terminal node, instead of both. This means that we no longer associate the Boolean function $f(x_0, \ldots, x_{n-1})$ with the root node. Instead, the root node is now associated with the Booelan *equation* $f(x_0, \ldots, x_{n-1}) = 1$. Third, and more significantly, we allow *linear combinations* of variables to be associated with a level, and not only single variables. We also allow the same variable to be associated with multiple levels, or more generally, we do not require the linear combinations of the levels to be linearly independent. This means that where standard ROBDDs have as many levels as variables, CRHS equations may have both more or fewer variables than levels.

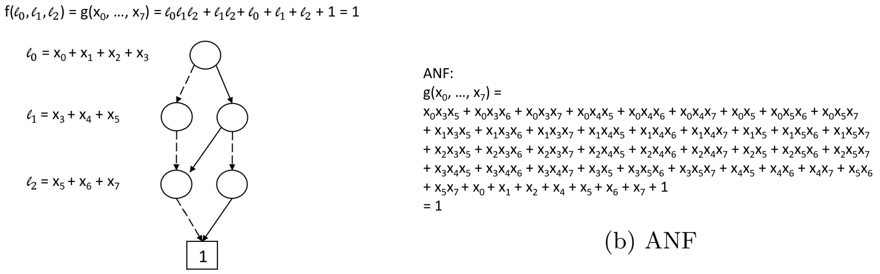
As the CRHS equation is an evolution from the ROBDD, we base the definition of CRHS equations on ROBDDs:

Definition 1. *A CRHS equation is a reduced, ordered BDD with a single terminal node and linear combinations of variables associated to each level. The set of linear combinations is referred to as the* left-hand side *of the CRHS equation, and the paths of the DAG as the equation's* right-hand sides*. A CRHS equation represents the Boolean equation $f(x_0, \ldots, x_{n-1}) = 1$, where f is the Boolean function corresponding to the BDD.*

Having linear combinations instead of single variables still allows us to use Shannon expansion to compute the ANF of the individual nodes in the CRHS equation, and therefore also for the ANF of the Boolean equation the CRHS equation represents. However, since CRHS equations allow linear combinations to be associated with the levels, it can be even more effective, in terms of nodes, in compressing a polynomial than a standard BDD. Figure 2a shows the CRHS equation made from the same BDD as in Figure 1a, but where the levels now are associated with some linear combinations. The linear combinations have been randomly chosen for the sake of demonstrating a concrete example. In Figure 2b the Boolean equation is written out in ANF.

While we have only 6 nodes in the CRHS equation, the ANF contains 46 terms. The BDD representing the same ANF with single variables will contain 18 nodes. In general it is easy to construct CRHS equations where the number of terms in the associated ANF is exponential in the number of nodes in the DAG.

We think of the linear combinations as the left-hand sides of a set of linear equations and all the paths compressed in the DAG as the set of right-hand sides. Choosing a path through the DAG in a CRHS equation, as seen in Figure 3a, is then the same as fixing a right-hand side vector for the set of linear combinations in the equation's left-hand side (Figure 3b). This system of linear equations can than be solved using standard linear algebra.


$f(\ell_0, \ell_1, \ell_2) = g(x_0, \dots, x_7) = \ell_0\ell_1\ell_2 + \ell_1\ell_2 + \ell_0 + \ell_1 + \ell_2 + 1 = 1$
$\ell_0 = x_0 + x_1 + x_2 + x_3$
$\ell_1 = x_3 + x_4 + x_5$
$\ell_2 = x_5 + x_6 + x_7$
1
ANF:
$g(x_0, \dots, x_7) =$
$x_0x_3x_5 + x_0x_3x_6 + x_0x_3x_7 + x_0x_4x_5 + x_0x_4x_6 + x_0x_4x_7 + x_0x_5 + x_0x_5x_6 + x_0x_5x_7$
$+ x_1x_3x_5 + x_1x_3x_6 + x_1x_3x_7 + x_1x_4x_5 + x_1x_4x_6 + x_1x_4x_7 + x_1x_5 + x_1x_5x_6 + x_1x_5x_7$
$+ x_2x_3x_5 + x_2x_3x_6 + x_2x_3x_7 + x_2x_4x_5 + x_2x_4x_6 + x_2x_4x_7 + x_2x_5 + x_2x_5x_6 + x_2x_5x_7$
$+ x_3x_4x_5 + x_3x_4x_6 + x_3x_4x_7 + x_3x_5 + x_3x_5x_6 + x_3x_5x_7 + x_4x_5 + x_4x_6 + x_4x_7 + x_5x_6$
$+ x_5x_7 + x_0 + x_1 + x_2 + x_4 + x_5 + x_6 + x_7 + 1$
$= 1$

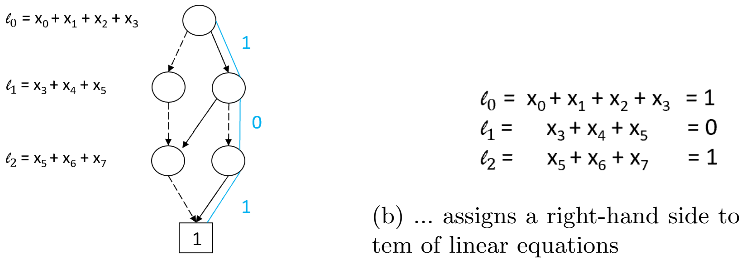
(b) ANF

(a) CRHS

Fig. 2: Example of a CRHS equation and its corresponding ANF.

Definition 2. *The solution set of a CRHS equation is the union of the solution sets of all linear equation systems given by the left-hand side and the CRHS equation's right-hand sides.*

This solution set of a CRHS equation is precisely the assignments for which the Boolean function associated with the equation's DAG evaluates to 1.


$\ell_0 = x_0 + x_1 + x_2 + x_3$
$\ell_1 = x_3 + x_4 + x_5$
$\ell_2 = x_5 + x_6 + x_7$
1
0
1
1
$\ell_0 = x_0 + x_1 + x_2 + x_3 = 1$
$\ell_1 = x_3 + x_4 + x_5 = 0$
$\ell_2 = x_5 + x_6 + x_7 = 1$

(a) Choosing a path (blue) through a CRHS equation...

(b) ... assigns a right-hand side to the system of linear equations

Fig. 3: Example of CRHS equation and one associated linear system.

While we normally ignore the underlying Boolean polynomials associated with the nodes, including the ANF associated with the root, they are useful for showing that operations available to a BDD can be done on CRHS equations without changing the solution set of the equation.

2.3 Basic operations on CRHS equations

Traditionally, there have been two operations on BDDs relevant for CRHS equations: Reduction of a BDD [5] and the swapping of the variables of two adjacent levels of a BDD [25]. With the transition from Multiple Right Hand Sides equations [24] to CRHS equations, two more operations were introduced [26,?]: adding the linear combination of one level onto the level below, and level extraction. Both of these operations are a natural consequence of the introduction of linear dependencies among the linear combinations of the CRHS equation. Combined with swapping, they allow for an adapted version of Gaussian elimination to be performed on the linear combinations of the levels. How these operations are used together will be covered in Section 4. Here we will briefly describe the operations, for full details see [25,26].

The *reduction algorithm* merges together nodes that have the same Boolean polynomial associated with them. They can easily be identified, since if two nodes have the same low child and high child, they must represent the same Boolean polynomial. The DAG of a CRHS equation can end up in an unreduced state when any of the other operations is performed.

Level extraction can be applied in the special case when the "linear combination" l associated with a level is just a constant $b \in \{0, 1\}$. In that case all outgoing edges from the nodes on the level assigning the value $(b + 1)$ give an inconsistency and should be deleted. When only b-edges remain as outgoing edges, it can be shown using Shannon expansion that the polynomial associated with a node on the b-level is equal to the polynomial associated with its remaining child. We can therefore merge the parent and child node. Since all nodes on the level can be merged this way, the whole level is effectively removed, and the number of levels in the CRHS equation decreases by 1.

The *swap operation* is an algorithm which swaps the linear combinations of two adjacent levels, taking care to rearrange the nodes and edges in such a way that the underlying ANF of the root node is preserved. In other words, doing a swap operation does not change the solution set of a CRHS equation.

Adding two levels in a CRHS equation is akin to the matrix operation of adding one row onto another. The first row stays the same, while the second row becomes the sum of the two. However, where any row in a matrix may be added to any other row, adding two levels in a CRHS equation requires the two levels to be adjacent. The procedure adds the linear combination of the top level to the one below it, and modifies edges and nodes in the process. As with the swap operation, the add operation is designed to preserve the underlying Boolean polynomial, so the solution set of a CRHS equation is not changed after an add operation.

One may use the swap operation to achieve both the adjacency and the ordering requirements as needed. In particular, one can use the swap and add operations to produce any linear combination in the span of the linear combinations for the levels, and make it appear on any desired level in a CRHS equation.

Swapping, adding and level extraction may leave the DAG in an unreduced state and it is therefore recommended to run the reduction algorithm afterwards. Swapping and adding levels can increase or decrease the number of nodes on the affected levels. This is entirely deterministic when the levels are known, and the processes are described in [25] and [26,27]. Level extraction will always decrease the number of nodes.

3 Modelling cryptographic primitives as system of CRHS equations

Any cryptographic primitive can be modelled as a system of non-linear equations, where any secret material is represented by variables. In this section we first briefly recall how block ciphers designed as substitution-permutation networks (SPN) are built, before explaining how a system of CRHS equations representing an SPN cipher can be constructed. It is straight forward to adapt this description to other types of ciphers or hash functions, as long as the non-linearity comes from S-boxes or other mappings that operate independently on blocks consisting of relatively few bits.

3.1 The structure of SPN block ciphers

SPN block ciphers are constructed by iterating a round function a number of times. Each round consists of the application of a non-linear transformation of the cipher state followed by an affine transformation and the xor addition of a round key. An SPN cipher starts with the addition of a whitening key to the plaintext, before iterating the round function r times. The output of the last round is the ciphertext. We refer to the block of bits at any point during the encryption procedure as the *cipher state.*

The non-linear layer is typically made by dividing the cipher state into blocks of b bits each, and substituting each block with the value given by a fixed b-bit S-box.

The affine transformation in a round can be constructed in many different ways, with various trade-offs. However, any affine transformation can be thought of as a linear transformation of the cipher state, followed by the addition of a constant. The linear transformation can always be realised as the multiplication of the cipher state with a fixed matrix over $GF(2)$. The only thing we care about in this paper is that each bit in the cipher state after the affine transformation is just a linear combination of the bits at the input, with the possible addition of a constant 1-bit.

An SPN cipher with r rounds needs $r + 1$ round keys, denoted as $K^0, K^1, \ldots, K^r$. The whitening key is K^0 and K^i is used in round i for $i = 1, \ldots, r$. The cipher has a master key K of κ bits, and all round keys are derived from K in a deterministic way. The computation of K^i from K can be linear or non-linear. If the key schedule is linear, each bit in K^i is again just a linear combination of the κ bits in K. If the key schedule is non-linear, the non-linear part in computing K^i typically uses the same S-box as used in the rest of the cipher.

3.2 Variables

We introduce the following set of variables to model an encryption $C = E_K(P)$ of an SPN cipher of block size n and key size κ:

- $K = k_0, k_1, \dots, k_{\kappa-1}$, the bits of the unknown user-selected key
- $P = p_0, p_1, \dots, p_{n-1}$, the bits of the plaintext
- $C = c_0, c_1, \dots, c_{n-1}$, the bits of the ciphertext
- $a_0, a_1, \dots, a_{m-1}$, bits in the cipher state at the output of the S-box layer in rounds $1, \dots, r-1$

For most ciphers $m = n(r-1)$, but if the S-box layer is incomplete, like for LowMC, $m = s(r-1)$ where s is the number of bits passing through S-boxes in each round. If the key schedule is linear these are all the variables that are needed. If the key schedule is non-linear we introduce auxiliary a_i-variables at the output of the non-linear transformations of the key schedule as well. See Figure 4 for an illustration of the setup of variables.

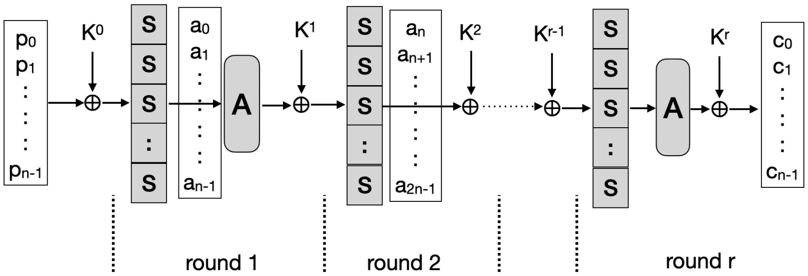


Fig. 4: Variables in a general SPN cipher. The round keys K^i depend on $k_0, \dots, k_{\kappa-1}$.

The introduction of variables can be done in different ways. The important point is that each bit in the cipher state at the input and output of the non-linear transformations can be expressed as a linear combination of the variables we have introduced. Note that it is not necessary to introduce new variables at the output of the S-boxes in the last round, since these bits can be expressed as linear combinations of the bits in K^r and the known ciphertext.

3.3 Constructing CRHS equations and the complete system

We construct the complete system representing the cipher by making one CRHS equation for each S-box instance appearing during the encryption process. For a b-bit S-box, let $l_0, \dots, l_{b-1}$ represent the input to the S-box and $l_b, \dots, l_{2b-1}$ the output. We then build a CRHS equation with $2b$ levels associated with

$l_0, \ldots, l_{2b-1}$. The CRHS equation will be constructed such that its associated polynomial $f(l_0, \ldots, l_{2b-1})$ evaluates to 1 for all values where $l_0, \ldots, l_{b-1}$ and $l_b, \ldots, l_{2b-1}$ is a matching input/output pair of the S-box, and 0 otherwise.

We now explain how to construct such an CRHS equation, using the 3-bit S-box from LowMC [1] as an example. First, assign the b linear combinations in the cipher state at the input of the S-box to the top b levels. Create a complete binary tree from the top node and down to level $b - 1$. Each path in this tree will correspond to the first $b - 1$ bits of a particular input value. See Figure 5 for the resulting structure when $b = 3$.

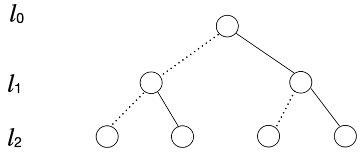


Fig. 5: The three highest levels of the CRHS equation representing the LowMC S-box. The input to the S-box is (l_2, l_1, l_0) with l_0 as least significant bit.

Second, construct a complete tree from the bottom node and upwards to level b. Assign the linear combinations in the cipher state at the output of the S-box to the b lowest levels. From each node on level b down to the bottom node there is now a unique path, representing an output value of the S-box. See figure 6 for the 3-bit S-box example.

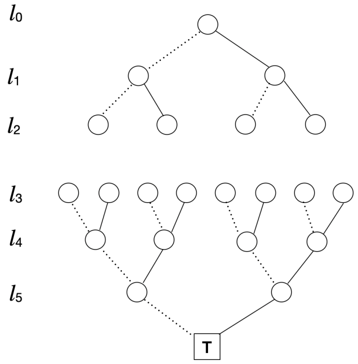


Fig. 6: All nodes and levels of the CHRS equation representing a 3-bit S-box. The output of the S-box is (l_5, l_4, l_3) with l_3 as the least significant bit.

Finally, connect nodes on level $b - 1$ to level b according to the look-up table defining the S-box. All complete paths in the CRHS equation will represent all correct input/output values of the S-box. See Figure 7 for the complete CRHS equation representing the 3-bit S-box used in LowMC [1].

We construct one CRHS equation for each application of the S-box in the cipher. The complete set of equations makes up the CRHS equation system representing the cipher.

Recall that each path in a CRHS equation gives a right-hand side to a system of $2b$ linear equations. To solve the equation system representing the cipher, we need to find one path in each CRHS equation such that the combined system of linear equations from all CRHS equations is consistent. For a fixed plaintext/ciphertext pair we only need to solve this system to find the values of all variables, in particular finding the variables representing the unknown key. We proceed to explain the techniques used in CryptaPath for solving a CRHS equation system.

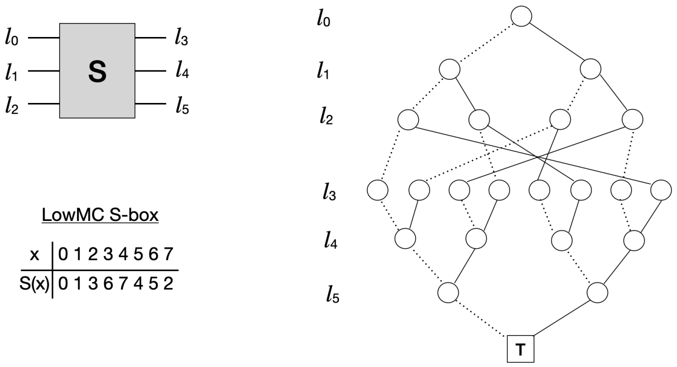


Fig. 7: The CRHS equation representing the LowMC S-box.

4 Solving a System of CRHS equations

A system of CRHS equations (SOC) is the set of CRHS equations which models one instance of a primitive. The *solution set* to the SOC is the intersection of the solution sets of each CRHS equation, the challenge is to find this set.

The solution set of the SOC is dependent on the paths in its CRHS equations. Collectively, the number of combinations of paths in the SOC is exponential in the number of CRHS equations. Yet we have only one associated system of linear combinations, namely the set of all linear combinations from the CHRS equations. Only a few selections of the paths will yield a consistent linear equation system when assigned to the associated linear combinations, resulting in a solution to the SOC. We call these paths *consistent* and identifying these paths will allow us to calculate the values of all the variables, including any key or pre-image variables. We see that the solution set of the SOC is given by all the consistent paths. *Solving a system of CRHS equation* is therefore a matter of identifying the consistent paths of the SOC, and removing the inconsistent ones.

4.1 Finding the Solution

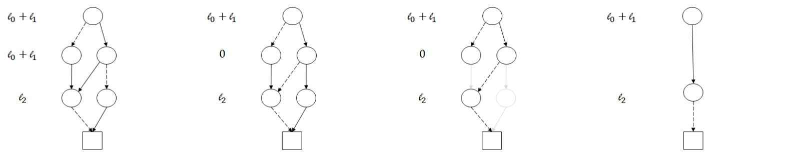
Allowing arbitrary linear combinations to be associated with levels may give rise to linear dependencies in the set of linear combinations in a CRHS equation. For a well-defined cipher, a single CRHS equation in the initial system will not have any dependencies among its linear combinations as it would imply a non-invertible linear transformation in the cipher. We therefore need to join multiple CRHS equations to give rise to linear dependencies.

Joining two CRHS equations E_1 and E_2 is a straightforward and memory efficient operation to execute. We simply replace E_1's terminal node with E_2's top node. The resulting CRHS equation contains one fewer node than the combined total of E_1 and E_2. It also contains all possible concatenations of paths from E_1 with paths from E_2, thus preserving the space of possible right-hand side vectors. This operation allows us to easily string together some, or all, CRHS equations into fewer, or even only one, CRHS equation(s).

Identifying linear dependencies in a SOC is straightforward. We extract the set of all linear combinations from all the CHRS equations in the SOC into one matrix, and use normal linear algebra to identify linear combinations that are linearly dependent. We keep track of where the linear combinations come from, and can use this information to decide which CRHS equations to join, and in what order. After joining, the resulting CRHS equation contains dependencies among its linear combinations. We then use linear absorption to remove the linear dependencies.

Linear absorption [27] is the process of resolving one linear dependency from the SOC. Resolving one linear dependency will remove all paths that give right-hand sides in the associated linear system (see Figure 3) that are inconsistent with this particular dependency. The idea is simple: Adding the relevant levels onto each other, as defined by the linear dependency, will result in a level whose "linear combination" is the constant 0. Since this level now has a constant value, we can remove the level using level extraction. Linear absorption is therefore the repeated applications of swap and add, ending in a level extraction. Figure 8 shows a simple example of linear absorption.

Solving the SOC is an iterative process: when there are no linear dependencies in any of the existing CRHS equations, join some CRHS equations together that give rise to some linear dependencies. Then use linear absorption to remove all these dependencies. In the end, when all CRHS equations have been joined together and all linear dependencies have been absorbed, we are left with only a single CRHS equation, containing only consistent paths. Any of these paths will give us a consistent system of linear equations that can be solved.



(a) Initial CRHS equation (b) After add operation, creating 0-level (c) Level extraction, part 1: remove 1-edges (d) Level extraction, part 2: remove the level

Fig. 8: Linear Absorption

4.2 Supporting techniques

We have now seen the core techniques required in order to solve a SOC. However, we also have two techniques which may aid in this process: the extraction and injection of linear equations, and the dropping of variables.

Extracting and injecting linear equations Extracting a linear combination is similar to level absorption. If at any given point all outgoing edges from all nodes on a level with linear combination l_i are 0 (or 1), we know that the linear equation $l_i = 0$ (or $l_i = 1$) must be true. This information is useful in two ways. First, we may use this information to eliminate one variable from the system, by choosing to eliminate any one variable x_j that appears in l_i. This is done by simply adding l_i (or $l_i + 1$) to any linear combination in the system that contains the variable x_j. Note that here we mean "add" in the simple sense of just xoring l_i (or $l_i + 1$) onto any other linear combination without modifying the BDD at all, not the add operation as described in Sec 2.3.

Second, for the level where we extracted this information, we will get 0 as the linear combination for that particular level. This level should then be removed in the same way as for level extraction. We note that the linear equation $l_i = 0$ (or $l_i = 1$) may be needed after all linear dependencies have been absorbed. It should therefore be stored, so that it can be added back into the final consistent linear system in the end.

We can similarly inject a constraint where we do not know the actual value in order to make a guess. If the guess is wrong the system will have no solutions. A system with no solutions is identified when a 0-level with only outgoing 1-edges appears, showing the contradiction. Deleting all 1-edges will in this case disconnect the top node from the bottom node, leaving no complete paths in the CHRS equation.

Dropping variables We introduce a novel technique, dropping variables, which has not been described before. Dropping a variable means to completely remove a variable from the SOC. This should therefore only be used on auxiliary variables, whose values we do not really care about, and not on variables representing the key of a cipher or a pre-image of a hash value.

We can remove any variable x_v from the SOC as follows: First, find all CRHS equations that have linear combinations containing x_v, and join them together. Now x_v only exists in the joined CRHS equation. Second, pick one level where x_v occurs, and use the add and swap operations to add this level to all other levels where x_v occurs. Now x_v only exists in the linear combination of one single level. This level is then moved, using the swap operation, to the lowest level, just above the terminal node. Finally, all incoming edges to the level with x_v are redirected directly to the bottom node and the x_v-level is completely removed, eliminating the last instance of x_v from the system.

Dropping a variable does not disturb the solution space of the variables we care about. This fact can be seen as follows: The consistent path that goes through the level is still valid, since the linear combination containing the single instance of x_v can not be part of any dependency. Assume that a consistent path will fix all other variables in the linear combination of the removed x_v-level. This path will then simply determine the value of x_v, but as x_v does not appear elsewhere in the system no inconsistencies can arise. Note, however, that we will never learn the actual value of dropped variables when solving the remaining system.

The benefit of dropping is that the SOC will contain fewer variables, and the CRHS equation may be simplified after removing a level and reducing. The cost of dropping is the number of add and swap operations that must be performed, possibly increasing the number of nodes. Note also that dropping variables does not resolve any linear dependencies and does not bring us closer to a solution in that sense. It just simplifies the system by eliminating a variable. In practice, variable dropping should only be done when a particular variable is already only contained in a single CRHS equation and the involved levels are already close to the bottom.

4.3 Complexity

We now turn to the complexity of the procedures described above. Absorbing one linear dependency is linear in the number of levels, and the number of dependencies must be less than the number of levels. Hence solving a system is at most quadratic in the number of levels, and the time complexity therefore mostly depends on the number of nodes the levels contain. Solving a non-linear equation system over $GF(2)$ is NP-complete in general and solving systems representing ciphers is still hard. For a cipher to be secure, the number of nodes in the SOC must increase significantly during an attempted solving of the SOC. We will see that all our operations are running in linear time in the number of nodes, and that it is not the run time that is crucial, but rather the memory consumption due to the increase in the number of nodes. We will therefore use the total number of nodes seen during solving as the measure of complexity.

Complexity of the operations Running the reduction on a CRHS equation is linear in its number of nodes and will only affect memory by removing nodes, so this operation has no cost in terms of memory. Adding and swapping levels are local operations, in the sense that only two levels are involved, and it only affects the number of nodes on the lower level. Nodes on the lower level may be removed and added, and in the worst case the number of nodes may end up being double that of the upper level.

Linear absorption of one linear dependency in a CRHS equation makes use of repeated applications of the swapping and adding operations, but each level is only involved once. The number of nodes can increase or decrease after resolving a dependency, and in the worst case the number of nodes in the CRHS equation may double when resolving a single linear dependency. This leads to the memory complexity for solving a SOC being potentially exponential in the number of initial dependencies.

As dropping a variable means moving the level to the bottom of the CRHS equation before being removed, repeated use of the swap algorithm may be needed. As with linear absorption, this is linear in terms of affected levels, but may in the worst case double the number of nodes. Finally, the level extraction and extracting linear equations (if any exist) are very quick to do and can only reduce the number of nodes.

Order of operations influences effective complexity In [23] it is pointed out that the process of solving a SOC can be summed up as three processes.

1. Joining CRHS equations.
2. Absorbing all linear dependencies.
3. Selecting a path from the remaining consistent paths and solving the linear system.

Of these three processes, absorbing dependencies is the hard one. As noted above, the number of nodes on a level may become the double of the number of nodes on the level above when performing the add and swap operations. That in turn means the number of add and swap operations, and the order of executing said operations are the driving factors in the growth of the memory complexity. Solving a system of CRHS equations will see a growth of memory complexity until a “tipping point” is reached, the point from where the memory usage will decrease towards a solution. Therefore, the order in which the dependencies are absorbed should be considered when solving a SOC, in an attempt to minimize the number of nodes at this tipping point.

Finding the best order for absorbing linear dependencies, and in turn the best order to join CRHS equations, is still an open research question.

5 CryptaPath

CryptaPath is a tool both for those who only want to perform an algebraic cryptanalytical attack on a primitive, and for those who wish to do research on CRHS equations. Only needing a reference implementation of a primitive to begin an attack ensures accessibility for those coming from other areas than algebraic cryptanalysis. For those who wish to go further, ways to specialize the solving algorithm are provided. Finally, being open source means that anyone can adapt the tool, changing it to their needs. An overview of how CryptaPath is organized and used is given in Appendix A.

5.1 Example usage and results

The simplest way of using CryptaPath is for example by giving the following command:

```
./cryptagraph cipher -c skinny64128 -r 4
```

This command will:

- Generate a random plaintext p and random key K for an instance of Skinny reduced to 4 rounds with 64-bit block and 128-bit key.
- Use this instance to encrypt p to a ciphertext c with K.
- Discard K.
- Create a SOC and fix the appropriate values of the variables corresponding to p and c.
- Run the default solver to remove all the dependencies in the system.
- Get the solution(s) from the solved SOC.
- Validate that the solution(s) correctly encrypt p to c, and output them.

Additional CLI parameters are available such as providing a known plaintext/ciphertext pair or providing a partially known key. In Table 1 we present several results of instances of round-reduced ciphers we were able to break using CryptaPath, with both time and the memory complexity given as number of nodes. We present both the maximal number of rounds without guessing any bits that we were able to solve as well as some larger instances that we were able to solve with several known key bits. In Table 2 we give some results on finding pre-images for a few variants of the Keccak hash function. The experiments were run on a laptop with an i7-4720HQ CPU @ 2.60GHz processor and 16 GB of RAM, which limit the maximum complexity to $\approx 2^{28}$ nodes for this particular hardware.

A few remarks on the numbers and the instances in Table 1: Cryptanalytic results using only one single plaintext/ciphertext pair is not very common, so for some of the ciphers there is little to compare against. In [23] both DES and a small version of AES, $SR^*(r, 2, 2, 4)$, are attacked with a similar approach as in this paper. For DES, 6 rounds can be broken with a dedicated strategy and using 6 chosen plaintexts, while with a single plaintext/ciphertext pair only 4 rounds can be attacked. The complexities are lower than in our case, showing that solving strategy plays a role. DES with a single plaintext/ciphertext pair is also attacked algebraically in [7], where the authors break 6 rounds after guessing more than 20 bits of the key.

cipher	number of rounds attacked	number of known bits	runtime	# nodes
DES	3 of 16	0/56	0:0.143	$2^{14.644}$
DES	4 of 16	10/56	7:31.102	$2^{26.899}$
LOWMC 64-1-80	19 of 164	0/80	14:17.784	$2^{26.528}$
LOWMC 64-1-80	27 of 164	26/80	9:28.118	$2^{26.199}$
LOWMC 128-31-80	1 of 12	0/80	0:0.849	$2^{17.741}$
LOWMC 128-31-80	2 of 12	68/80	14:34.702	$2^{26.845}$
LOWMC 256-1-256	24 of 458	0/256	11:24.846	$2^{26.540}$
LOWMC 256-1-256	45 of 458	65/256	9:42.992	$2^{26.228}$
PRESENT 80	2 of 31	7/80	10:0.642	$2^{27.004}$
PRESENT 80	2 of 31	8/80	1:18.747	$2^{24.480}$
PRINCE	2 of 12	0/128	0:5.865	$2^{19.831}$
PRINCE	4 of 12	87/128	5:31.046	$2^{26.153}$
PRINCE-CORE	4 of 12	21/64	0:13.592	$2^{22.298}$
SKINNY 64-128	4 of 36	0/128	0:0.437	$2^{14.975}$
SKINNY 64-128	5 of 36	70/128	14:1.398	$2^{27.120}$
SKINNY 128-128	3 of 40	0/128	0:0.444	$2^{15.285}$
SKINNY 128-128	4 of 40	32/128	16:29.616	$2^{27.247}$
SKINNY 128-128	4 of 40	34/128	3:46.160	$2^{25.825}$
SR* 2-2-8	1	0/32	0:0.108	$2^{15.298}$
SR* 2-2-8	2	12/32	0:0.705	$2^{18.060}$
SR* 2-2-8	3	12/32	6:4.170	$2^{26.743}$
SR* 2-2-8	4	23/32	0:8.904	$2^{21.128}$
SR* 4-4-4	1	0/64	0:0.074	$2^{12.053}$
SR* 4-4-4	2	25/64	0:25.430	$2^{22.970}$
SR* 4-4-4	3	46/64	2:52.634	$2^{25.479}$

Table 1: Results on block ciphers (runtimes in min:sec.milliseconds)

6 Conclusions and further work

There are two purposes of this paper. The first is to have a thorough explanation of the connection between CRHS equations and Boolean equations represented as ANF polynomials, since this has not been described earlier. The second purpose is to advertise an easy to use tool for doing algebraic cryptanalysis.

CRHS equations give a memory efficient representation of a Boolean equation in several variables. Many Boolean polynomials that are too big to be represented in ANF in practice can still be represented as CRHS equations. The size of a CRHS equation does not depend so much on the degree of its associated Boolean polynomial, but rather on how much ”regularity” there is in its paths. The theory for solving CRHS equation systems is now better understood, and with CryptaPath it has been compiled into a library that is available for anyone to use and adapt to their own needs. The optimal solving strategy is cipher dependent, and CryptaPath provides API’s to experiment with various strategies.

Another goal of CryptaPath is to provide a user interface for doing algebraic cryptanalysis of a particular cipher, without needing knowledge of how CRHS equations are constructed, and without needing to know how solving systems

rounds	rate	capacity	message-length	hash-length	number of known bits	runtime	# nodes
1	240	160	240	80	0/240	0:9.411	$2^{12.21}$
2	40	160	80	80	(39+32)/80*	5:33.516	$2^{25.64}$
2	80	120	80	80	49/80	2:20.401	$2^{24.37}$

Table 2: Results on Keccak variants (runtimes in min:sec.milliseconds)
*39 fixed variables in first message block, and 32 in the second.

of CRHS equations work. This is inspired from the tool CryptaGraph, which has an equally simple interface for applying a search for differential or linear characteristics.

Further work: In a longer perspective, we hope there will be more tools for analysing symmetric key primitives, that can be applied by only giving a reference implementation of the cipher in question. Right now it is not possible to simply copy the Rust source code of the ciphers in CryptaGraph's portfolio and apply them to CryptaPath, due to small differences in the Rust traits used by the two tools. For that reason, a standardized way of coding reference implementations needs to be agreed upon.

In our current work we have focused on attacks recovering the secret key in SPN ciphers or finding pre-images for hash functions. There are several directions further research can take for applying CRHS equations on other problems. In [17] CRHS equations are applied on the cipher GOST [8], which uses addition modulo 2^n for including round keys. Checking whether CRHS equations gives a good model for attacking ARX ciphers in general is one avenue to explore. Another topic for further work is applying CRHS equations on a search for the best linear hull or differential in a cipher. This is a hard problem in general and involves keeping a large number of partial solutions in memory at the same time, exactly the feature that a CRHS equation is suitable for.

Last, it is possible to generalize a BDD to a p-ary decision diagram, having p edges out of each node for $p > 2$. To keep the compactness of the CRHS equation p can not be too large. Apart from ciphers (like MiMC) that are defined over $\mathbb{F}_p$ where p is large, we are only aware of the hash function Troika [19] that uses a non-binary field at its base. Troika is defined over $\mathbb{F}_3$ and could be attacked using CRHS equations containing ternary decision diagrams. In contrast, SAT-solvers are inherently binary and can not be adapted as easily to solve problems defined over non-binary fields.

References

1. Martin Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. Ciphers for mpc and fhe. In *Advances in Cryptology - EUROCRYPT 2015*, pages 430–454, 2015. LNCS 9056.
2. Martin R Albrecht, Carlos Cid, Lorenzo Grassi, Dmitry Khovratovich, Reinhard Lüftenegger, Christian Rechberger, and Markus Schofnegger. Algebraic cryptanalysis of stark-friendly designs: application to marvellous and mimc. In *International*

Conference on the Theory and Application of Cryptology and Information Security, pages 371–397. Springer, 2019.

3. Beate Bollig. On the complexity of some ordering problems. In Erzsébet Csuhaj-Varjú, Martin Dietzfelbinger, and Zoltán Ésik, editors, *Mathematical Foundations of Computer Science 2014*, pages 118–129, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.
4. Michael Brickenstein and Alexander Dreyer. Polybori: A framework for gröbner-basis computations with boolean polynomials. *Journal of Symbolic Computation*, 44(9):1326 – 1345, 2009. Effective Methods in Algebraic Geometry.
5. Randal E Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, 100(8):677–691, 1986.
6. Randal E Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys (CSUR)*, 24(3):293–318, 1992.
7. Nicolas T. Courtois and Gregory V. Bard. Algebraic cryptanalysis of the data encryption standard. In Steven D. Galbraith, editor, *Cryptography and Coding*, pages 152–169, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.
8. V. Dolmatov. GOST 28147-89: Encryption, Decryption, and Message Authentication Code (MAC) Algorithms. RFC 5830 (Informational), March 2010.
9. Jean-Charles Faugère. A new efficient algorithm for computing gröbner bases (f4). *Journal of Pure and Applied Algebra*, 139:61–88, 1999.
10. Jean Charles Faugère. A new efficient algorithm for computing gröbner bases without reduction to zero (f5). In *ISSAC '02*, 2002.
11. Michael R Garey and David S Johnson. A guide to the theory of np-completeness. *Computers and intractability*, pages 641–650, 1979.
12. F. Gossen, A. Murtovi, J. Linden, and B. Steffen. Add-lib 2.0.0 beta, 2018.
13. Katrina Groth, Chengdong Wang, and Ali Mosleh. Hybrid causal methodology and software platform for probabilistic risk assessment and safety monitoring of socio-technical systems. *Reliability Engineering & System Safety*, 95(12):1276–1285, 2010.
14. Mathias Hall-Andersen and Philip S. Vejre. Generating graphs packed with paths estimation of linear approximations and differentials. *IACR Transactions on Symmetric Cryptology*, 2018(3):265–289, Sep. 2018.
15. Ioannis Filippidis AKA johnyf. dd, 2020.
16. Jun Kawahara, Koki Sonoda, Takeru Inoue, and Shoji Kasahara. Efficient construction of binary decision diagrams for network reliability with imperfect vertices. *Reliability Engineering & System Safety*, 188:142–154, 2019.
17. Oleksandr Kazymyrov, Roman Oliynykov, and Håvard Raddum. Influence of addition modulo 2^n on algebraic attacks. *Cryptography and Communications*, 8(2):277–289, 2016.
18. Matthias Krause. Bdd-based cryptanalysis of keystream generators. In Lars R. Knudsen, editor, *Advances in Cryptology — EUROCRYPT 2002*, pages 222–237, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.
19. Stefan Kölbl, Elmar Tischhauser, Patrick Derbez, and Andrey Bogdanov. Troika: a ternary cryptographic hash function. *Designs, Codes and Cryptography*, 88(1):91–117, 2020.
20. Jorn Lind-Nielsen, Haim Cohen, and Nikos Gorogiannis. Buddy, 2014.
21. Ilya Mironov and Lintao Zhang. Applications of sat solvers to cryptanalysis of hash functions. In Armin Biere and Carla P. Gomes, editors, *Theory and Applications of Satisfiability Testing - SAT 2006*, pages 102–115, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

22. Kaisa Nyberg and Lars Ramkilde knudsen. Provable security against a differential attack. *Journal of Cryptology*, 8(1):27–37, Dec 1995.
23. Håvard Raddum and Oleksandr Kazymyrov. Algebraic attacks using binary decision diagrams. In *International Conference on Cryptography and Information Security in the Balkans*, pages 40–54. Springer, 2014.
24. Håvard Raddum and Igor Semaev. Solving multiple right hand sides linear equations. *Designs, Codes and Cryptography*, 49(1):147–160, Dec 2008.
25. Richard Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proceedings of 1993 International Conference on Computer Aided Design (ICCAD)*, pages 42–47. IEEE, 1993.
26. Thorsten Ernst Schilling and Håvard Raddum. Analysis of trivium using compressed right hand side equations. In Howon Kim, editor, *Information Security and Cryptology - ICISC 2011*, pages 18–32, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
27. Thorsten Ernst Schilling and Håvard Raddum. Solving compressed right hand side equation systems with linear absorption. In Tor Helleseth and Jonathan Jedwab, editors, *Sequences and Their Applications – SETA 2012*, pages 291–302, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
28. Claude E Shannon. A symbolic analysis of relay and switching circuits. *Electrical Engineering*, 57(12):713–723, 1938.
29. Fabio Somenzi. Cudd: Cu decision diagram package release 3.0.0. FTP download link (broken?): vlsi.Colorado.EDU.
30. Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT solvers to cryptographic problems. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, volume 5584 of *Lecture Notes in Computer Science*, pages 244–257. Springer, 2009.
31. Dirk Stegemann. Extended bdd-based cryptanalysis of keystream generators. In Carlisle Adams, Ali Miri, and Michael Wiener, editors, *Selected Areas in Cryptography*, pages 17–35, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.
32. Tom van Dijk AKA trolando. Sylvan, 2019.

A Overview of the code and usage of CryptaPath

The code base of CryptaPath is broken into two parts:

- The Crush library which provides an implementation of the CRHS equations and System of CRHS equations along with several APIs for the operations that one can be performed on them (swap, add, absorb, drop and more). An interface (a Rust trait) to construct solvers, with default implementation for several methods is also provided.
- the CryptaPath tool uses the Crush library. The tool itself is composed of a simple command line interface (CLI), a set of generic methods for building specifications for a SOC from an implementation of a cipher, and several example ciphers that we implemented for analysis. It also provides a generic solver, built from the interface of the Crush library.

We decided to make this separation from the belief that the usage of CRHS equations can be explored outside of cryptanalysis, and in that case the Crush library as a standalone will be sufficient. However, when used in the case of cryptanalysis, the main obstacle to usage for researchers would be to generate the SOC for every cipher and variant they want to analyze. The goal of CryptaPath is to simplify this task. By specifying an implementation that respects the provided interface, the tool will generate the SOC from the Rust source code.

While we provide several implementations of primitives (reduced versions of AES, LowMC, Skinny, Prince, Present, DES and Keccak) we encourage users to add their own if they want to analyze it. To facilitate any future implementation job we are providing several helper functions making it possible to run an implementation against test vectors to ensure its correctness. As already mentioned, we provide a general good solving algorithm which will work out of the box for any SPN cipher or sponge construction implemented in Rust. As a user gets familiar with the tool, tailor made solvers can be created and tested.

A.1 Usage

Simple usage of the tool can be made by using the provided CLI. A user can generate a SOC for any of the primitives implemented in CryptaPath for any number of rounds and run the solver on it. The user can provide a specific plaintext/ciphertext pair and solve for the key. The user may also fix arbitrary bits of the key to see how much easier solving becomes with a partially guessed key. If no plaintext/ciphertext pair is provided CryptaPath will generate a random plaintext and a random key respecting any fixed bits, and compute the corresponding ciphertext at runtime. Any solution found will be validated by encrypting the plaintext and ensuring the result matches the ciphertext. The system of CRHS equations can be output in the form of a .bdd file for studying and fed back into CryptaPath later.

As specified earlier, it is possible and encouraged to add new ciphers into CryptaPath. We provide for that purpose a `Cipher` trait which a reference implementation has to follow. Existing ciphers can be used as examples on how to make an implementation.

We provide two similar solvers which we believe to be a good general fit for all algorithms. The main difference between them is the use of the drop operation which as noted earlier can either increase or decrease the complexity.

In the case of the solver which uses dropping of variables we consider variables that can be dropped without any joining of CRHS equations, and compare the cost of dropping them against the cost of absorbing the cheapest dependency found. The cost of resolving a dependency or dropping a variable is estimated by summing up the number of nodes in the levels that have to be swapped or added to resolve it. There are a lot of heuristics which can be explored to improve the solving, and in particular we expect a tailor made solver to outperform ours when targeting a specific algorithm. A new solver can be implemented using the traits we provide with a minimal amount of code to rewrite.

A specific part of the solver which we encourage users to tweak is the `feedback` function. This function is called by the solver every time it completes an operation on the system and is used to provide feedback to the user. Its role is to allow for gathering data from the SOC during the solving process. Our default implementation prints several metrics on the terminal window such as the number of individual CRHS equations left in the system, the maximal number of node reached and the number of absorbed dependencies.