Analysis of Trivium Using Compressed Right Hand Side Equations

Thorsten Ernst Schilling, Håvard Raddum
thorsten.schilling@ii.uib.no,havard.raddum@ii.uib.no

Selmer Center, University of Bergen

Abstract. We study a new representation of non-linear multivariate equations for algebraic cryptanalysis. Using a combination of multiple right hand side equations and binary decision diagrams, our new representation allows a very efficient conjunction of a large number of separate equations. We apply our new technique to the stream cipher Trivium and variants of Trivium reduced in size. By merging all equations into one single constraint, manageable in size and processing time, we get a representation of the Trivium cipher as one single equation.

Key words: multivariate equation system, BDD, algebraic cryptanalysis, Trivium

1 Introduction

In this paper we present a new way of representing multivariate equations over $GF(2)$ and their application in algebraic cryptanalysis of the stream cipher Trivium.

In algebraic cryptanalysis one creates an equation system of the cipher being analyzed and tries to solve it. The solution will reveal the key or some other secret information. Solving the system representing a cipher in time faster than exhaustive search will be a valid attack on the cipher.

There exist several ways to represent such a system, e.g., ANF, CNF [1] or MRHS [2]. Along these representations different families of algorithms to solve equation systems have been proposed, e.g., Gröbner Basis like algorithms [3], XL [4] SAT-solving [1] and Gluing/Agreeing algorithms [5, 2, 6].

For the stream cipher Trivium, which has an especially simple structure, one can easily construct an equation system describing its inner state constraints using some known keystream bits. Attempts at solving this system have nevertheless been unsuccessful. While reduced versions of Trivium could be broken [1], there is no attack better than brute-force known for the full version.

Previous methods describe the Trivium-equation system as a set of non-linear constraints, which have to be true in conjunction. One can simplify those equation systems by joining several constraints into a single new one. Unfortunately the conjunction operation usually leads to exponentially big objects, which quickly become too big for today's computers.

In this paper we present a new way of representing the constraints given by a non-linear equation system. This representation allows all equations in the Trivium-equation system to be merged into one single equation. The process of merging equations has asymptotically exponential complexity, but using our new technique we are nevertheless still able to complete it in practice, with an actual complexity far lower than the $O(2^{80})$-bound for Trivium.

The paper is organized as follows. In Section 2 we explain the Multiple Right Hand Side equation representation and Binary Decision Diagrams as well as some operations on both constructions. The cipher Trivium is also briefly described. Section 3 introduces Compressed Right Hand Side equations and shows how a solution to such equations can be found. In Section 4 we present our experimental results and explain how to reduce the Trivium equation system to a single Compressed Right Hand Side equation. Section 5 concludes the paper. The appendix contains examples for several of the used constructions and algorithms.

2 Preliminaries

2.1 Multiple Right Hand Side Equation Systems

The Multiple Right Hand Side (MRHS) representation [2, 5] is an efficient way to represent equations containing much inherent linearity. Equation systems coming from cryptographic primitives are well suited for MRHS representation, since cryptographic algorithms are usually built using both linear and non-linear components.

A MRHS equation is a linear system with, as the name suggests, multiple right hand sides. We write one MRHS equation as $Ax = B$, where A and B are matrices with the same number of rows, and x is a vector of variables. Any assignment of x such that Ax equals some column in B satisfies the equation.

We construct a system of MRHS equations from a cryptographic primitive as follows. First we assign variable names to the bits of cipher states at several places in the encryption process. The assignment of variables should be done such that the bits of the input and output of any non-linear component can be written as linear combinations of variables. Then we construct one MRHS equation $Ax = B$ for each non-linear component f. The rows of A are the input and output linear combinations of f. Finally, we list all possible inputs to f, with their corresponding outputs. Each input/output pair becomes a column in B. An example of this can be found in the appendix.

Following this procedure we can construct a system of MRHS equations

$$A_1x = B_1, \ldots, A_mx = B_m$$

for any cryptographic primitive that uses relatively small non-linear components.

For a given solution to the system, there is exactly one column in each B_i corresponding to this solution. We say such a column is *correct*. If the system has a unique solution, there is only one correct right hand side in each B_i. Solving

MRHS equation systems means identifying columns in the B_i that cannot be correct, and delete them.

Several techniques for solving MRHS systems exist. One of them is called *gluing* and is used in this paper. Gluing means to merge two equations into one, making sure that only solutions that satisfy both original equations are carried over into the new (glued) equation.

Gluing two equations reduces the number of equations by one. The process of gluing can be repeated, packing all initial equations into one MRHS equation. The resulting equation is nothing more than a system of linear equations, and can easily be solved. The solution we find will necessarily satisfy all the original initial MRHS equations, so this strategy will solve the system in question.

The problem we face when applying the technique of gluing in practice, is that the number of right hand sides in glued equations tends to increase exponentially. Only when there are just a few equations remaining, with large A-matrices, will the restrictions on potential solutions be so limiting that the number of possible right hand sides rapidly decreases. As we shall see, however, the problem of exponential growth in the number of right hand sides may be circumvented using *binary decision diagrams*.

2.2 Binary Decision Diagrams

In this section we will introduce binary decision diagrams (BDDs). A BDD is a directed acyclic graph used to represent a set of binary vectors or a Boolean formula. They are mostly used in design and verification systems and were introduced by S.B. Akers [7]. Later implementations and refinements led to a broad interest in the computer science community as BDDs allow the manipulation of large propositional formulae [8, 9] in compressed form. Sometimes they are used as an alternative to *guess-and-verify* solvers of propositional problems since they enable one to keep track of all satisfying assignments at once and offer polynomial time algorithms to count the number of solutions of a propositional problem given in the form of a BDD.

The use of BBDs in cryptanalysis for LFSRs was proposed by Krause [10] and successfully applied to Grain with NLFSRs by Stegemann [11].

Definition 1 (Binary Decision Diagram). *A binary decision diagram is a pair $\mathcal{D} = (G, L)$ where $G = (V, E)$ is a directed acyclic graph, and $L = (l_0, l_1, \ldots, l_{r-1}, \epsilon)$ is an ordered set of variables.*

The vertices of G are $V = \{v_0, v_1, \ldots, v_{s-1}\} \cup \{\top, \bot\}$ where all v_i denote inner vertices and contain exactly one root vertice with no incoming edges. Every inner vertex v has exactly two outgoing edges, which we call the 1-edge and the 0-edge. We call $\top$ and $\bot$ terminal vertices, they have no outgoing edges. Every vertex v is associated with a variable, denoted $L(v)$, and for all edges (u, v) we have $L(u)$ appearing before $L(v)$ in L. We always have $L(\top) = L(\bot) = \epsilon$.

We denote with $G(v)$ the subgraph of G rooted at v, i.e., the graph consisting of vertices and edges along all directed paths originating at v. For any pair of vertices u, w it holds that if $G(u) = G(w)$ then $u = w$.

There exist other definitions of BDDs which do or do not include the order L or the reducedness property of unequal subgraphs. The definition above is also known as a reduced ordered BDD and is canonical [9]. We denote the number of vertices in a binary decision diagram $\mathcal{D}$ by $\mathcal{B}(\mathcal{D}) = |G|$. The size of a BDD depends heavily on the order L. Finding the optimal ordering to minimize $\mathcal{B}(\mathcal{D})$ is an NP-hard problem [9].

In Definition 1 L induces a partial order of the vertices. We visualize a BDD by drawing it from top to bottom, with vertices of the same order on the same line, and we say that these vertices are at the same *level*. There is only one root vertex and it must necessarily associated with the first variable in L. This node associated with l_0 is drawn on top, and the nodes $\top$ and $\bot$ are drawn on the bottom. An example of a BDD can be found in the appendix.

Definition 2 (Accepted Inputs of a BDD). *In a BDD $\mathcal{D}$ every path from the root vertex to the terminal vertex $\top$ is called an accepted input of $\mathcal{D}$.*

Since every inner node is associated with a variable, we can regard every edge as a *variable assignment*. To find a variable assignment (or vector) which is accepted by the BDD, we start with an empty vector of length $|L|$. Following a path from the root vertex to $\top$ we visit at most one node at each level.

Whenever we go from v through a 1-edge, we say that $L(v)$ is assigned to 1, and $L(v) = 0$ whenever we go via a 0-edge. A path that ends up in $\top$ gives us one accepted input in terms of variable assignments. Likewise, a path from the root vertex to $\bot$ gives us a rejected input to a specific BDD. By traversing all paths to $\top$ we can build the set of all vectors which are accepted by the BDD.

If a path from the root to $\top$ *jumps* a level, i.e. the assignment to a variable l_k is undefined since the path does not contain a vertex v with $L(v) = l_k$, both assignments to this variable are accepted and we get two different variable assignments. If an accepted input jumps r levels in total we get 2^r different satisfying assignments from this path. An example of accepted inputs of a BDD can be found in the appendix.

AND-Operation on BDDs. As shown above, we can use BDDs to represent the set of vectors that satisfy a Boolean equation. By the nature of our equation systems, we need a way to merge solution sets from different equations. Below is a simple recursive algorithm which does this. A more general version of the algorithm can be found in [12].

Let $\mathcal{D}$ and $\mathcal{D}'$ be two BDDs with v_0 as the root of $\mathcal{D}$ and u_0 the root of $\mathcal{D}'$. The conjunction of $\mathcal{D}$ and $\mathcal{D}'$ into a new BDD $\mathcal{E}$ is done as follows.

First we need to define an ordering on the union of variables from $\mathcal{D}$ and $\mathcal{D}'$. Next, we set the root node of $\mathcal{E}$ at the top level, and label it (v_0u_0). Then we perform Algorithm 1, which will fill in nodes and edges in $\mathcal{E}$, from top to bottom.

The paths in the BDD that results after merging $\mathcal{D}$ and $\mathcal{D}'$ using Algorithm 1 will correspond to vectors that satisfy both Boolean equations related to $\mathcal{D}$ and $\mathcal{D}'$. One feature of the conjunction of two BDDs is that all nodes in the new BDD can be labelled with (vu) where v and u come from the two orginal BDDs.

Algorithm 1 Merging BDDs $\mathcal{D}$ and $\mathcal{D}'$ into $\mathcal{E}$

while $\exists$ a node (vu) in $\mathcal{E}$ without outgoing edges **do**
  Let v^e be child of v in $\mathcal{D}$ through e-edge
  Let u^e be child of u in $\mathcal{D}'$ through e-edge
  if $L(v) = L(u)$ **then** ▷ v and u are at the same level
    Insert (v^0u^0) at level $\min\{L(v^0), L(u^0)\}$ with 0-edge from (vu).
    Insert (v^1u^1) at level $\min\{L(v^1), L(u^1)\}$ with 1-edge from (vu).
  end if
  if $L(v) < L(u)$ **then** ▷ v is higher up than u
    Insert (v^0u) at level $\min\{L(v^0), L(u)\}$ with 0-edge from (vu).
    Insert (v^1u) at level $\min\{L(v^1), L(u)\}$ with 1-edge from (vu).
  end if
  if $L(v) > L(u)$ **then** ▷ u is higher up than v
    Insert (vu^0) at level $\min\{L(v), L(u^0)\}$ with 0-edge from (vu).
    Insert (vu^1) at level $\min\{L(v), L(u^1)\}$ with 1-edge from (vu).
  end if
end while

It is then not hard to see that the following upper bound holds

$$\mathcal{B}(\mathcal{E}) \leq \mathcal{B}(\mathcal{D})\mathcal{B}(\mathcal{D}'). \tag{1}$$

We will use this fact later in the paper. For a more detailed description and analysis of operations on BDDs one might consult [12, 9, 8]. An example of the AND-operation on BDDs can be found in the appendix.

2.3 Trivium

Trivium [13] is a synchronous stream cipher and part of the ECRYPT Stream Cipher Project portfolio for hardware stream ciphers. It consists of three connected non-linear feedback shift registers (NLFSR) of lengths 93, 84 and 111. These are all clocked once for each key stream bit produced.

Trivium has an inner state of 288 bits, which are initialized with 80 key bits, 80 bits of IV, and 128 constant bits. The cipher is clocked 1152 times before actual keystream generation starts. The generation of keystream bits and updating the registers is very simple. The pseudo-code in [13] is a good and compact description of the whole process of generating keystream as shown in Algorithm 2.

Here z_i is the key stream bit, and the registers are filled with the bits $s_1, \ldots, s_{288}$ before clocking.

For algebraic cryptanalysis purposes one can create four equations for every clock; three defining the inner state change of the registers and one relating the inner state to the key stream bit. Solving this equation system in time less than trying all 2^{80} keys is considered a valid attack on the cipher.

Small Scale Trivium. For our experiments we considered small scale versions of Trivium. While reduced versions of a cipher sometimes dismiss some structural

Algorithm 2 Trivium Pseudo-Code

for $i = 1$ to N **do**

$t_1 \leftarrow s_{66} + s_{93}$
$t_2 \leftarrow s_{162} + s_{177}$
$t_3 \leftarrow s_{243} + s_{288}$

$z_i \leftarrow t_1 + t_2 + t_3$ ▷ Keystream bit

$t_1 \leftarrow t_1 + s_{91} \cdot s_{92} + s_{171}$
$t_2 \leftarrow t_2 + s_{175} \cdot s_{176} + s_{264}$
$t_3 \leftarrow t_3 + s_{286} \cdot s_{287} + s_{69}$

$(s_1, s_2, \ldots, s_{93}) \leftarrow (t_3, s_1, \ldots, s_{93})$
$(s_{94}, s_{95}, \ldots, s_{177}) \leftarrow (t_1, s_{94}, \ldots, s_{176})$
$(s_{178}, s_{179}, \ldots, s_{288}) \leftarrow (t_2, s_{178}, \ldots, s_{287})$

end for

component of the full scale cipher, e.g. Bivium [1], we try to keep our reduced versions as close to Trivium as possible.

We scale with respect to the number of bits in the state. When we speak about Trivium-N, we are speaking about a cipher with N bits of internal state, that is, scaled down by a factor $\alpha = N/288$. The lengths of the two first registers will be 93α and 84α, rounded to the nearest integers. The length of the last register will be what remains to get N as the total number of state bits (either $\lfloor 111\alpha \rfloor$ or $\lceil 111\alpha \rceil$).

In the full Trivium, the three top positons in each register are all used as tap positions. This property is also carried over to all the scaled versions. For the tap positions appearing elsewhere in the registers, we simply scale their indices with α. For example, as 66 is used as a tap position in the full Trivium, for Trivium-N the corresponding tap position will be 66α, rounded to the nearest integer, with the following exception: Tap positions that are close to each other in the full Trivium may get the same indices in some Trivium-N if α is small enough. When this happens, we reduce the tap position of the smaller index by one, thus ensuring that all tap positions in Trivium-N are distinct. The equation systems representing Trivium-N and Trivium will then have similar structures.

3 Compressed Right Hand Side Equation Systems

With MRHS equations a clear separation between the linear and the non-linear part of an equation was introduced. Overall it yielded a much smaller representation for equations typical in algebraic cryptanalysis. Nevertheless, solving MRHS equations has been limited to relatively small-scale examples because of the problem with a big number of right hand sides.

It was shown in [7] that representing Boolean equations as BDDs is canonical with respect to the ordering of variables. This way of recording sets of assign-

ments gives us the advantage that we may have a moderate number of nodes in a BDD, but very many paths from the root leading to $\top$. Rather than writing out all satisfying assignments, or a truth table for a Boolean equation, only a BDD is retained in memory. However, when experimenting with equations from certain ciphers, BDDs may also become too big to keep in computer memory [11].

By combining the MRHS and BDD approaches, we get a new way to handle large equation systems in algebraic cryptanalysis. We call this representation of equations *Compressed Right Hand Sides* (CRHS) equations.

Definition 3 (CRHS). *A compressed right hand side equation is written as* $Ax = \mathcal{D}$*, where* A *is a* $k \times n$*-matrix with rows* $l_0, \ldots, l_{k-1}$ *and* $\mathcal{D}$ *is a BDD with variable ordering (from top to bottom)* $l_0, \ldots, l_{k-1}$*. Any assignment to* x *such that* Ax *is a vector corresponding to an accepted input in* $\mathcal{D}$*, is a satisfying assignment.*

An easy example of a CRHS equation can be found in the appendix.

CRHS Gluing. If we are given two Boolean equations $f_1(X_1) = 0, f_2(X_2) = 0$ and we want to find vectors in variables $X_1 \cup X_2$ which satisfy both equations simultanously we can do this by investigating their individual satisfying vectors at common variables. If two vectors have the same values at common variable indices we have found a vector which satisfies both equations. This operation is part of the Gluing operation described in Section 2.1.

If we are given two CRHS equations $[C_1]x = \mathcal{D}_1, [C_2]x = \mathcal{D}_2$ and we want to compute their common solutions we use a similar technique called *CRHS Gluing*. The result of gluing both equations above is

$$\begin{bmatrix} C_1 \\ C_2 \end{bmatrix} x = \mathcal{D}_1 \wedge \mathcal{D}_2.$$

Any assignment of x such that $\begin{bmatrix} C_1 \\ C_2 \end{bmatrix} x$ is an accepted input in the conjunction $\mathcal{D}_1 \wedge \mathcal{D}_2$ gives a solution to both initial equations simultanously. Like the Gluing operation on MRHS equations the right hand side BDD contains all possible combinations of vectors from the original equations. The difference is that satisfying vectors are no longer explicit in the computer memory, but are recorded in a compressed format, namely as paths in the BDD.

It is easy to output all possible vectors from the paths in a BDD. There also exists an easy polynomial-time (in the number of nodes) algorithm to count the number of accepted inputs to a BDD. An example of CRHS-gluing can be found in the appendix.

3.1 Dependencies among linear combinations

The left hand side in a CRHS equation is equal to the left hand side in a MRHS equation, namely a set of linear combinations $\{l_0, \ldots, l_{k-1}\}$ in the variables of

the system. If we glue several CRHS equations together, it might happen that the resulting left hand side matrix in the glued equation does not have full rank, that is, the set of linear combinations in the left hand side contains linear dependencies.

The BDD on the right hand side treats the l_i as variables, and is oblivious to the constraint that some of them should sum to zero or one. Therefore, an accepted input in the BDD may or may not satisfy the linear dependencies known to the left hand side. These paths should be taken out of the BDD in order to not produce false solutions.

The straight-forward way to remove paths that do not satisfy some linear dependency is to use the AND-operation. The number of nodes in the BDD representing a linear equation $g(l_0, \ldots, l_{k-1})$ is two times the number terms in g. It is then easy to construct the BDD for any g, and combine it with the BDD in the equation using the AND-operation. This will remove all false solutions.

4 Experimental Results

While exploring the possibilities of CRHS equations we used a software library called *Cudd*[14]. The Cudd software library implements various types of BDDs and algorithms/operations which can be performed on BDDs. The code base is optimized and usable on a personal computer even for very big BDDs.

We used Cudd together with C++ code and developed a program capable of reading different equation systems representing scaled Triviums and then gluing the equations together.

It was crucial in the experiments to find out the size of the resulting CRHS equation when gluing many of them together. This number is important to determine in order to evaluate the feasability of our method. Theoretically the size of the final CRHS equation C is upper bounded by

$$\mathcal{B}(C) \leq \mathcal{B}(c_0) \cdot \mathcal{B}(c_1) \cdot \ldots \cdot \mathcal{B}(c_{r-1})$$

when gluing CRHS equations $c_0, c_1, \ldots, c_{r-1}$ into C. This value is exponential in the number of nodes and might lead to infeasible sizes of BDDs, even for quite small versions of Trivium. However, our experiments showed that the size of the BDD for the glued CRHS equations was far smaller than the upper bound, and stayed manageable. Thus we are indeed, in contrast to MRHS equation systems, able to glue *all* equations in large CRHS equation systems together. For MRHS equation systems, gluing all equations together will reveal the solutions to the system. As we explain below, it is more complicated for CRHS equation systems, due to false solutions in the right hand side BDD.

In the experiments reported below, we created CRHS equation systems representing Trivium-N for various values of N. Then we glued all equations into one single big CRHS equation. We examined different aspects of the equation systems, which can tell us something about their solvability with our method. For several small scale versions we measured the following properties:

Value	Description
n	# of variables = # of initial CRHS equations
k	# of different linear combinations of variables
$\mathcal{B}$	# vertices in BDD in final equation
lc	# of linear constraints for solution
Sol.	# paths in final BDD
Mem.	Memory consumption in MB

N	n	k	$\mathcal{B}$	lc	Sol.	Mem.
35	85	173	$2^{18.86}$	88	$2^{85.67}$	87
40	94	191	$2^{20.57}$	97	$2^{93.77}$	182
45	106	215	$2^{21.68}$	109	$2^{106.60}$	358
50	115	233	$2^{21.15}$	118	$2^{115.60}$	258
55	127	257	$2^{21.55}$	130	$2^{127.60}$	329
60	138	282	$2^{22.34}$	144	$2^{140.35}$	560
65	148	299	$2^{22.66}$	151	$2^{148.60}$	687
70	160	323	$2^{22.42}$	163	$2^{160.49}$	588
75	171	349	$2^{22.78}$	178	$2^{173.83}$	742

Table 1. Experimental results

Initial equations have 4 nodes in the BDD, so we see from Table 1 that the size of the BDD after gluing all equations together is far from the theoretical upper bound. However, the growth of $\mathcal{B}$ is exponential just with a very small constant. It is worth to notice that $\mathcal{B}$ is not strictly increasing with N. We also see that the expected number of paths that satisfy all constraints given by lc is between 2^{-4} and 2^{-2}.

A point worth mentioning is that the exponential upper bound for gluing CRHS equations together is tight, in general. There *are* equations that will achieve the bound when glued together. Equation systems coming from ciphers tend to be very sparse, in the sense that each initial equation contain few variables, and each variable only appears in a few equations. This is also the case for Trivium. Two equations that do not share any variables have a linear size when glued together. As shown in (5), the gluing in this case is basically putting one BDD on top of the other. This may explain why it is particularly easy to glue together CRHS equations coming from scaled versions of Trivium.

Full Trivium. So what about $N = 288$? For full Trivium our computer ran out of memory before finishing gluing all equations together. On the other hand, we were able to glue 404 of the 666 initial equations together, producing a CRHS equation C_1 of size $2^{22.9}$. Then we glued the remaining initial equations into C_2, of size $2^{24.8}$. By using the upper bound (1) for merging two BDDs, we have then demonstrated that the single CRHS equation representing the full Trivium has a size smaller than $2^{47.7}$. The true size of the BDD for the full Trivium is probably

a lot smaller than $2^{47.7}$, given that the upper bound we use has proved to be very loose for the systems we study. In any case, we know that the size of the CRHS equation representing the full Trivium is quite far from the 2^{80}-bound for a valid attack.

4.1 Solving Attempts

If a single CRHS equation gave a solution as readily as a MRHS equation, we would be done, and have an algebraic attack on Trivium with complexity much smaller than the $O(2^{80})$-bound for exhaustive search. As noted above, we can not deduce a solution straight from the CRHS equation, since we have eventually to find a path in the BDD that satisfies a number of linear constraints. For scaled Triviums, we have of course tried the straight-forward approach mentioned in Section 3.1. Gluing BDDs representing linear constraints onto the BDD of the cipher CRHS equation unfortunately makes the size grow too large very rapidly.

Another solving method we have tried works as follows. Let the set of linear constraints to be satisfied be contained in a matrix LC. We set LC at the (single) top node in the BDD, and will propagate the matrix through the whole BDD according to Algorithm 3.

Algorithm 3 Propagating linear constraints through BDD with k levels.

```
for i = 0 to k do
    for every node a at level i do
        if a contains matrix then
            Build matrix M of linear constraints present in all matrices in a
            if l_i = 0 is consistent with M then
                Send M|_{l_i=0} through 0-edge
            end if
            if l_i = 1 is consistent with M then
                Send M|_{l_i=1} through 1-edge
            end if
        end if
    end for
end for
```

What we are bascally doing is to fix the value of l_i in LC to 0 or 1 when passing LC through a 0- or 1-edge out of a node at level i. If the linear constraints of LC would become inconsistent by sending it across an edge, the matrix is not propagated in that direction. Nodes receiving more than one LC-matrix will only keep linear constraints present in all matrices.

A node containing a matrix could be interpreted as saying *"Any path below me must satisfy the linear constraints in my matrix."* We hope that the matrix ending up in the $\top$-node will contain some other linear constraints than the ones we started with. If this is the case, we can repeat Algorithm 3 with increasingly large LC.

In small examples (that can be checked by hand) the method of propagating the linear constraints through the BDD works, but for Trivium-35 it did not, as there were no new linear constraints in the matrix arriving at the bottom. What we did see for Trivium-35 however, was that there is a significant amount of nodes at levels $113 - 138$ in the BDD that did not receive any matrices (due to inconsistencies). At some levels almost half of the nodes were empty. We learn from this that there is no path satisfying the linear constraints in LC that can pass through these nodes, and so they can be deleted. Hence we can use Algorithm 3 to prune the BDD, and reduce its size.

5 Conclusion & Further Work

In this paper we have introduced a new way of representing algebraic equations, and shown its advantages compared to previously known representations. With the CRHS representation it is possible to merge many more equations together, than what is possible by other approaches. Building the CRHS equation system for Trivium, we have shown that Trivium may be described by a single CRHS equation with a BDD of size $2^{47.7}$ nodes, at most.

We have not yet been able to solve big CRHS equation systems, due to the many false solutions appearing in the right hand side BDD. The problem that needs to be solved is: **How do we efficiently find a path in a BDD that satisfies a set of linear constraints?** The method of matrix propagation helps in reducing the size of the BDD, and may be an approach worth pursuing. This is a topic for further research.

Finally, we should keep in mind that the operation of merging equations in a system is a process with exponential complexity. This is also true for CRHS equations, but for systems representing versions of Trivium we can do full merging anyway, because of the structure of the system. Solving non-linear equation systems is NP-hard in general, so we cannot hope to have a solving algorithm without any exponential step in it. Gluing all equations together is an exponential step, and full gluing normally solves the system. We can then speculate that after gluing all initial equations into one, we have overcome the exponential step and that the remaining problem for finding a solution can be solved efficiently. It is not clear that the problem of finding a path in a BDD subject to a set of linear constraints must have exponential complexity in the number of nodes. Further investigation into this question is needed.

References

1. McDonald, C., Charnes, C., Pieprzyk, J.: Attacking Bivium with MiniSat. eSTREAM, ECRYPT Stream Cipher Project, Report 2007/040 (2007) `http://www.ecrypt.eu.org/stream`.
2. Raddum, H., Semaev, I.: Solving Multiple Right Hand Sides linear equations. Designs, Codes and Cryptography **49**(1) (2008) 147–160
3. Faugère, J.: A new efficient algorithm for computing Gröbner bases (F4). Journal of Pure and Applied Algebra **139**(1-3) (1999) 61–88

4. Courtois, N., Alexander, K., Patarin, J., Shamir, A.: Efficient algorithms for solving overdefined systems of multivariate polynomial equations. Lecture Notes in Computer Science **1807** (2000) 392–407
5. Raddum, H.: MRHS Equation Systems. Lecture Notes in Computer Science **4876** (2007) 232–245
6. Semaev, I.: Sparse algebraic equations over finite fields. SIAM Journal on Computing **39**(2) (2009) 388–409
7. Akers, S.: Binary decision diagrams. IEEE Transactions on Computers **27**(6) (1978) 509–516
8. Somenzi, F.: Binary decision diagrams. In: Calculational System Design, volume 173 of NATO Science Series F: Computer and Systems Sciences, IOS Press (1999) 303–366
9. Knuth, D.: The Art of Computer Programming. Number Vol 4, Fascicles 0-4 in The Art of Computer Programming. ADDISON WESLEY (PEAR) (2009)
10. Krause, M.: BDD-based cryptanalysis of keystream generators. Lecture Notes in Computer Science **1462** (2002) 222–237
11. Stegemann, D.: Extended BDD-Based Cryptanalysis of Keystream Generators. Lecture Notes in Computer Science **4876** (2007) 17–35
12. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers **35** (1986) 677–691
13. Cannière, C.D., Preneel, B.: Trivium specifications. ECRYPT Stream Cipher Project (2005)
14. Somenzi, F.: CUDD: CU Decision Diagram Package. `http://vlsi.colorado.edu/~fabio/CUDD/` (2009)

Appendix

Example 1 (MRHS). The basic non-linear component in Trivium is the bitwise multiplication found in the function updating the registers. The new bit (x_6) coming into a register at some point is related to the old ones ($x_1, \ldots, x_5$) by

$$x_1 \cdot x_2 + x_3 + x_4 + x_5 = x_6.$$

The multiplication is the non-linear component, with inputs x_1 and x_2, and a single linear combination as output, namely $x_3 + x_4 + x_5 + x_6$. There are four different inputs to this function, hence there will be four columns in the B-matrix. The corresponding MRHS equation is

$$\begin{bmatrix} 1&0&0&0&0&0 \\ 0&1&0&0&0&0 \\ 0&0&1&1&1&1 \end{bmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{pmatrix} = \begin{bmatrix} 0&1&0&1 \\ 0&0&1&1 \\ 0&0&0&1 \end{bmatrix}. \tag{2}$$

Example 2 (BDD). Figure 1 shows an example BDD. The vertex v_0 is the root. Solid lines indicate 1-edges and dashed lines indicate 0-edges. In this example the order is (l_0, l_1, l_2) as indicated to the left.

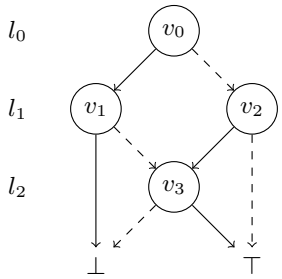

Fig. 1. Example BDD

Example 3 (Accepted Inputs). The accepted inputs for the BDD in Figure 1 are the vectors (l_0, l_1, l_2):

$$(0,0,0)$$
$$(0,0,1)$$
$$(0,1,1)$$
$$(1,0,1)\ .$$

One can see that $(0,0,0)$ and $(0,0,1)$ are on the same path from v_0 to $\top$. On that path no node associated with l_2 is visited, so l_2 can be assigned both values.

A Boolean equation may be characterized by its set of satisfying assignments. Building a BDD whose accepted inputs match the set of satisfying assignments, gives us another representation of the same equation. For example, the Boolean equation corresponding to the BDD in Figure 1 is $l_0l_1 + l_0l_2 + l_1l_2 + l_0 + l_1 = 0$.

Example 4 (AND operation). The top half of Fig. 2 shows the BDDs of two Boolean functions. The left BDD shows $l_0+l_1+l_2 = 0$, the right BDD represents $l_0l_1 + l_2 = 0$. Both BDDs share the same order of variables, and the resulting BDD of their conjunction after reduction is shown below the two original BDDs.

Example 5 (CRHS). We write equation (2) from Example 1 as a CRHS equation by converting the right hand side into a BDD.

Instead of writing out the left hand matrix of equation (2), we write down the corresponding linear combinations, and give them the names l_0, l_1, l_2.

$$\begin{bmatrix} x_1 & = l_0 \\ x_2 & = l_1 \\ x_3 + x_4 + x_5 + x_6 & = l_2 \end{bmatrix} = \left\{ \begin{array}{ll} l_0 & \\ & \\ l_1 & \\ & \\ l_2 & \\ & \\ & \end{array} \right. \quad \begin{array}{c} v_0 \\ \\ v_1 \\ \\ v_2 \quad v_3 \\ \\ \top \quad \bot \end{array} \tag{3}$$

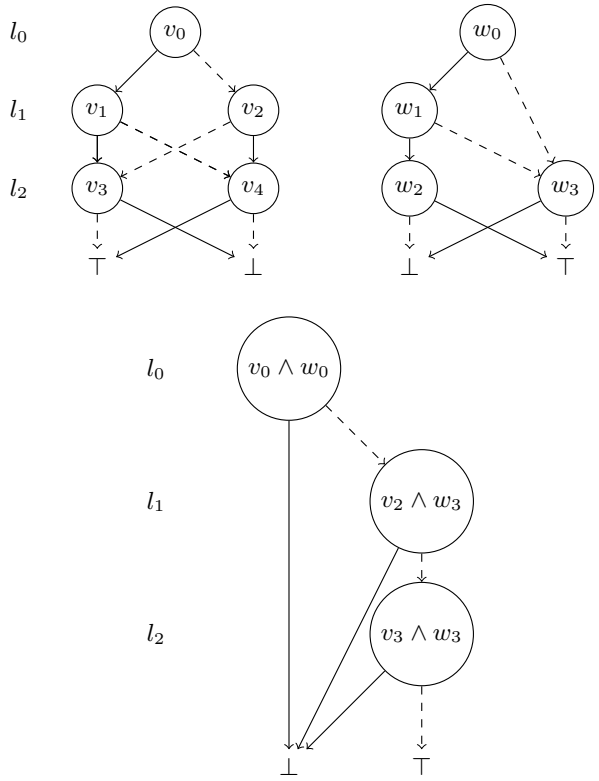

Fig. 2. AND-operation example

The right hand side of the CRHS equation is a *compressed* version of the right hand side in a MRHS equation. Every accepted input in the graph of the CRHS equation stands for one right hand side of the corresponding MRHS equation. The example above contains the edge (v_0, v_3). This edge is *jumping* over a level, i.e. every path through this edge does not contain any vertex at level l_1. That means that for a path containing the edge (v_0, v_3), the variable l_1 can take any value. The path $\langle v_0, v_3, \top \rangle$ thus contains two vectors for (l_0, l_1, l_2), namely $(0, 0, 0)$ and $(0, 1, 0)$.

Example 6 (CRHS Gluing). The following two equations are similar to equations in a Trivium equation system. In fact, the right hand sides of the following are taken from a full scale Trivium equation system. The left hand matrices have

been shortened.

$$\begin{bmatrix} x_1 & = l_0 \\ x_2 & = l_1 \\ x_3 + x_4 & = l_2 \end{bmatrix} = \left\{ \begin{array}{ll} l_0 & u_0 \\ l_1 & u_1 \\ l_2 & u_2 \quad u_3 \\ & \top \quad \bot \end{array} \right. , \quad \begin{bmatrix} x_4 & = l_3 \\ x_5 & = l_4 \\ x_6 + x_7 & = l_5 \end{bmatrix} = \left\{ \begin{array}{ll} l_3 & v_0 \\ l_4 & v_1 \\ l_5 & v_2 \quad v_3 \\ & \top \quad \bot \end{array} \right. \tag{4}$$

The gluing of the equations above is

$$\begin{bmatrix} x_1 & = l_0 \\ x_2 & = l_1 \\ x_3 + x_4 & = l_2 \\ x_4 & = l_3 \\ x_5 & = l_4 \\ x_6 + x_7 & = l_5 \end{bmatrix} = \left\{ \begin{array}{ll} l_0 & w_0 \\ l_1 & w_1 \\ l_2 & w_2 \quad w_3 \\ l_3 & w_4 \\ l_4 & w_5 \\ l_5 & w_6 \quad w_7 \\ & \top \end{array} \right. , \tag{5}$$

where $\bot$-paths in this last graph are omitted for better readability. Note that omitting these paths does not decrease the overall number of vertices. The resulting equation has 8 nodes where the corredsponding MRHS equation would have 16 right hand sides.