Algebraic Attacks Using Binary Decision Diagrams

Håvard Raddum† and Oleksandr Kazymyrov‡

†Simula@UiB, ‡University of Bergen
Norway

Abstract. Algebraic attacks have been developed against symmetric primitives during the last decade. In this paper we represent equation systems using binary decision diagrams, and explain techniques for solving them. Next, we do experiments with systems describing reduced versions of DES and AES, as well as systems for the problem of determining EA-equivalence. We compare our results against Gröbner basis and CryptoMiniSat.
Keywords: binary decision diagram, block cipher, algebraic attack, symmetric primitives

1 Introduction

Comparing the complexity of finding the key to a cryptosystem with solving a system of equations were first mentioned by Claude Shannon [1], and is today known as algebraic attacks. The main idea is to describe an encryption scheme via a system of equations and solve it. However, algebraic attacks against cryptographic primitives began to develop actively only in the early 2000s. Several methods to attack hash functions, stream and block ciphers have been described [2–10].

In the middle of the 20th century, it was proposed to use binary decision diagrams (BDDs) for representing Boolean functions [11, 12]. This representation has several advantages. Many logical operations on BDDs can be implemented by polynomial-time graph manipulation algorithms [12], and the memory consumption can be extremely low, even for very complex Boolean functions. Most modern cryptographic primitives are based on binary logic because of the large spread of binary computers. Therefore, the description of cryptographic transformations using Boolean or vectorial Boolean functions is an easy task.

Several attacks based on BDDs exist for stream ciphers. Their efficiency was demonstrated both for general methods and for particular cases on A5/1, E0 and Trivium [13, 14]. In this paper we extend previous results on block ciphers and present new specific strategies and approaches for solving systems of equations based on BDDs.

We apply the proposed methods on DES with reduced number of rounds, on MiniAES (a small variant of Rijndael) and on the problem of determining EA-equivalence. Our experiments on DES allow us to break six rounds in approximately one minute on a MacBook Air 2013. This is a factor 2^{20} improvement over the best earlier algebraic attack on DES using MiniSAT [5]. There have been several earlier attempts to break MiniAES [7, 15, 16]. Approaches that exploit the short key in MiniAES (only 16 bits) succeed very quickly, but the general methods of F4 and XL/XSL failed to solve systems representing more than one round of MiniAES. The approach we use in this paper does not exploit the short key, while still solving systems representing 10 rounds of MiniAES using approximately 45 minutes and 8GB of memory.

The rest of the paper is organized as follows. Section 2 explains BDDs and the fundamental operations we do on them. Section 3 describes our approach to solving BDD systems, and introduces some solving strategies. Section 4 gives the details and results of our algebraic attack

against DES and MiniAES as well as the EA-equivalence problem, comparing complexities against SAT-solver and Gröbner base techniques. Finally, Section 5 concludes the paper and give some directions for further research.

2 Binary Decision Diagram Fundamentals

The literature discusses several variants of BDDs. For clarity we will always mean a *zero-suppressed, reduced*, and *ordered* BDD in this paper. A comprehensive treatment of BDDs can be found in [12]. In this section we only give a brief description with emphasis on visualization and our use of a BDD.

2.1 Binary Decision Diagrams

A BDD is a directed acyclic graph. Exactly one node in the graph, called the *source node*, has no incoming edges, and exactly one node in the graph, called the *sink node*, has no outgoing edges. All nodes except for the sink node are called *internal* nodes, and have one or two outgoing edges, called the 0-*edge* and/or the 1-*edge*. In most other descriptions of BDDs, each internal node is associated with a variable. In this paper each internal node will be associated with a **linear combination** of variables. There are no edges between nodes associated with the same linear combination.

When visualizing a BDD, we draw the graph from top to bottom, with the source node on top, the sink node at the bottom, and all edges directed downwards. All internal nodes are organized in horizontal *levels* between the sink and source nodes. One level consists of all nodes associated to one particular linear combination, and we write the linear combination to the left of the level. Dotted edges indicate 0-edges while solid lines indicate 1-edges. An example of a BDD with four levels associated with linear combinations in four variables is shown in Fig. 1.

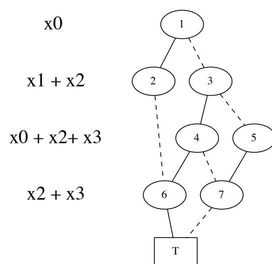


Fig. 1: Example of a BDD with four levels.

In the literature there are various ways to understand a BDD. Some interpret a BDD to represent a family of sets while others see a BDD as an efficient encoding of a Boolean function. In this paper we put emphasis on the fact that a path from the source to the sink node assigns values to the linear combinations of the levels. If we choose the b-edge ($b \in \{0, 1\}$) out from a node, we assign the value b to the linear combination associated with the level of the node. Any path from the source to the sink node gives values to the linear combinations and can be regarded as a right-hand side in a system of linear equations.

2.2 Representing an S-box as a BDD

We are interested in finding a BDD that represents a given S-box with n input bits and m output bits. Let the input bits and output bits of the S-box be $x_0, \dots, x_{n-1}$ and $y_0, \dots, y_{m-1}$, respectively. Let the first n levels be associated with $x_0, \dots, x_{n-1}$ (x_0 for the source node and x_{n-1} for level n), and build a complete binary tree from x_0 to x_{n-1}. Next, assign $y_0, \dots, y_{m-1}$ to the m lowest levels (with y_0 at the highest of these), and build a complete binary tree upwards from the sink node to the y_0-level, branching in 0-edges and 1-edges. Then there will be only one path from a given node at the y_0-level to the sink node. There will be 2^m nodes at the y_0-level, each representing a unique path to the sink node, assigning values to $y_0, \dots, y_{m-1}$.

Any path from the source node down to level x_{n-1} will assign values to the input bits $x_0, \dots, x_{n-2}$. Selecting a 0-edge or a 1-edge out of a node at the x_{n-1}-level will complete the assignment of input bits. This edge is connected to the node at the y_0-level whose unique path to the sink node will give the correct output of the S-box. Joining all nodes at the x_{n-1}-level to all nodes at the y_0-level in this way will complete the construction of the BDD. Fig. 2 shows an example of a BDD representing a 4×4 S-box.

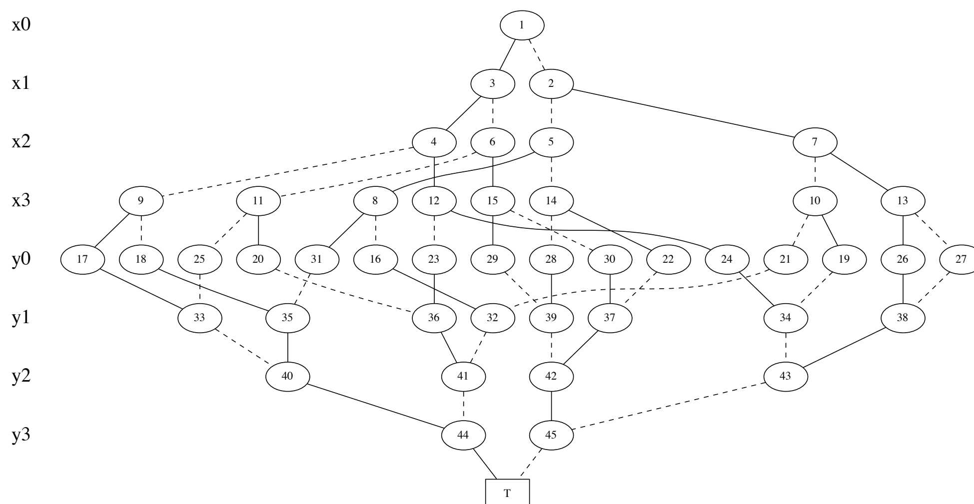


Fig. 2: BDD representing the S-box {5,C,8,F,9,7,2,B,6,A,0,D,E,4,3,1}.

2.3 Basic Operations on a BDD

We must be able to run the *reduction* algorithm [17] on a BDD, bringing the BDD into a reduced state. The reduction algorithm basically merges nodes representing equivalent Boolean functions, thus minimizing the number of nodes in the BDD. For a fixed order of the linear combinations, a reduced BDD is unique. There are two other operations that forms the core of *linear absorption* (explained later). Both were described in [18], but we repeat them briefly here for completeness.

Swapping levels. This operation swaps the linear combinations at two adjacent levels, and was first described in [19], using single variables. When changing the order of the levels, nodes and edges must be re-arranged in the BDD to preserve the underlying function. Fortunately, swapping levels is a local operation, meaning that only nodes and edges at the two involved levels need to be touched while the rest of the BDD remains intact. The time complexity of swapping two levels is linear in the number of nodes on the highest level, but the number of nodes on the lowest level may double in the worst case.

After swapping two levels, the BDD may not be in the reduced state, and it may be necessary to run the reduction algorithm. Hence, the number of nodes in the BDD after swapping two levels may increase or decrease. By repeatedly swapping levels one may put the set of linear combinations for the levels into any desired order. Finding the order of levels that give the fewest nodes is an NP-complete problem [20].

Adding levels. Traditionally, the levels in a BDD have been associated with single variables and not linear combinations. It has therefore not been natural to think of "adding" one level onto another. This changes when we have linear combinations associated with the levels. If l_1 and l_2 are two linear combinations associated to two adjacent levels (l_1 above l_2), we are interested in replacing l_2 with $l_1 + l_2$. The algorithm for adding levels was first described in [18], and follows the same logic as with swapping levels. Nodes and edges at the levels for l_1 and l_2 must be rearranged to preserve the underlying function, but the rest of the BDD remains the same. The complexity is similar to swapping, and the number of nodes at the new level associated to $l_1 + l_2$ may double in the worst case. The reduction algorithm should be run after adding levels to make sure the BDD remains in a reduced state.

3 Solving Systems of Equations with Linear Absorption

By repeatedly swapping and adding linear combinations, we can essentially do all linear operations on the linear combinations of the BDD. We can, for instance, perform Gaussian elimination on the set of linear combinations. As will become clear in the following, a barrier to find a solution to a system of equations arises when there are dependencies among the linear combinations of the levels in a BDD. We overcome this problem by using linear absorption. The technique was first described in [18], but we include an example of the procedure here as it is central in our approach to solve non-linear equation systems.

3.1 Absorbing One Linear Dependency

The attentive reader will have noticed that the linear combinations in the example BDD in Section 2.1 are not independent. If we label them l_0, l_1, l_2, l_3 from top to bottom we have $l_0 + l_2 + l_3 = 0$. Thus, when we select a path in the BDD and create the corresponding linear system of equations, we may or may not get a consistent system. If the values assigned to l_0, l_2 and l_3 sum to 0 we get a solution, if not, the system is inconsistent. We use linear absorption to remove all paths that yield inconsistent systems as follows.

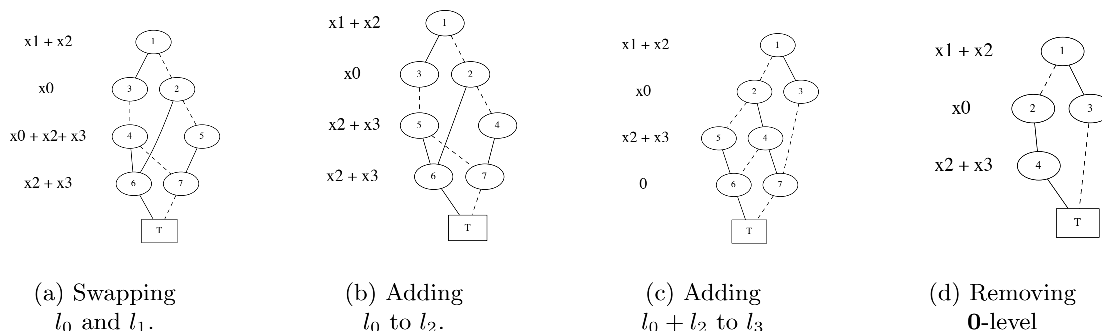
First, swap l_0 and l_1 to obtain the BDD in Fig. 3a. Next, use addition of linear combinations to add l_0 onto l_2, and obtain the BDD in Fig. 3b. Finally, we use addition again to add $l_0 + l_2$ to l_3. This creates the $\mathbf{0}$-vector as linear combination for the lowest level, resulting in the BDD shown in Fig. 3c.

When selecting a path in the BDD now, it does not make sense to choose a 1-edge out of a node on the level associated with **0**. Such a path would yield a "$0 = 1$"-assignment. Hence we can delete all outgoing 1-edges from the nodes at the **0**-level. Now we are certain that any remaining path will yield a system of linear equations that is consistent with the linear dependency $l_0 + l_2 + l_3 = 0$.

Moreover, the whole level associated with **0** can be removed. It is easy to show that the Boolean function represented by a node on this level is equal to the function for the node pointed to by the 0-edge. Hence, all nodes on the **0**-level can be merged with their children along the 0-edge, and the level disappears. We say that the linear dependency $l_0 + l_2 + l_3 = 0$ has been *absorbed*. The resulting BDD for our example is shown in Fig. 3d.

In general, the removal of 1-edges from a level associated with the **0**-vector may create internal nodes with no incoming edges. We call these *orphan* nodes as they have no parents. After absorbing one linear dependency, all orphan nodes, and subgraphs only reachable through an orphan node, should be removed as part of the reduction procedure.

If there are several dependencies among the linear combinations in a BDD, we can easily find all and absorb them one after another. When all dependencies have been absorbed, we know that *any* remaining path in the BDD will give a consistent linear system of equations, which in turn is solved to find the values of the actual variables.



(a) Swapping l_0 and l_1.

(b) Adding l_0 to l_2.

(c) Adding $l_0 + l_2$ to l_3

(d) Removing **0**-level

Fig. 3: Absorbing one linear dependency.

3.2 Building and Solving Equation Systems as BDDs

It is rather straight-forward to build a system of equations representing a cryptographic primitive as a set of BDDs. For an encryption algorithm, the user-selected key bits become variables. We also look at the cipher blocks between the rounds of the primitive. We assign variable names to the bits between rounds, such that the input and output bits of all non-linear components can be written as linear combinations of variables.

For each non-linear component we then construct the corresponding BDD, like explained for S-boxes in Section 2.2. We replace the x_i and y_j with the linear combinations actually occuring in the cipher. After this we are left with a set of BDDs with linear combinations from the same pool of variables.

To proceed with finding a solution to the system we must join the BDDs together. There exist algorithms for joining two BDDs [17][12, p. 16], but they are somewhat complex, and assume single variables associated with the levels. We do it in a much simpler way:

– To join two BDDs, just replace the sink node of one with the source node of the other.

With this simple operation, we can easily string together some or all BDDs in a system and get fewer, or only one, BDD(s) in the set. If we join all BDDs together, finding a solution is equivalent to finding a path in the joined BDD that gives a right hand side yielding a consistent linear system. As can be expected, for interesting systems there will be many dependencies among the linear combinations in a fully joined BDD, so finding a path respecting all these dependencies is not trivial. We can, however, try to handle this problem with linear absorption, and if we can absorb all linear dependencies in the BDD we know that any remaining path will give a consistent linear system. The algorithm for solving a system of BDDs can then be summed up as:

1. Join BDDs.
2. Absorb all linear dependencies.
3. Select path and solve resulting system of linear equations.

3.3 Complexity

The first and third steps in the general solving algorithm are easy (assuming a modest number of BDDs and variables, which is the case even for full scale ciphers). Hence the second step must be hard if our cryptographic primitives are to remain secure. What we get when joining all BDDs together is a very long and very slim BDD, basically just a string of many small BDDs. The number of nodes at one level may double when adding or swapping two linear combinations, and after absorbing a whole linear dependency the total number of nodes may, in the worst case, double. This seems to lead to exponential growth in the number of dependencies absorbed, but in practice the number of nodes after absorbing a linear dependency is very far from doubling. Remember, the number of nodes may also *decrease* when applying a swap or an add operation.

If we expect a unique or only a few solutions, the BDD after absorbing all dependencies will have only one or a few paths. A BDD with only one path has only one node at each level, tied together with a string of 0- and 1-edges. Since all the systems we are interested in have very few solutions, we know that the number of nodes must decrease sharply before the last dependencies are absorbed. This means we will always reach some tipping point when absorbing dependencies, where the number of nodes in the BDD starts to decrease.

We take as our measure of complexity the largest number of nodes that a BDD contained during linear absorption. This is a measure of memory complexity, and is not equivalent to the time it takes to solve a system. On the other hand, there is no guessing involved in our solving method, and no operations that must be repeated an exponential number of times. Memory rather than time is the resource that constrains us. With 8GB of RAM it is hard to find a system where our solver runs for over an hour without either finishing or running out of memory. Therefore, we believe the largest number of nodes we had during the solving process is the most meaningful measure of complexity.

3.4 Solving strategies

When joining together many BDDs and absorbing all linear dependencies, the solving complexity depends heavily on the order the BDDs are joined. Finding the ordering of BDDs that

gives the minimum complexity is probably a hard problem. During our experiments we have not found a strategy for ordering that is universally best. However, we describe here some strategies for how to join and absorb with the aim to keep the complexity down.

Automatic Ordering. This is a default strategy, that can be applied to any system and does not require any deeper understanding of how the BDDs have been made. The procedure is to look for the subset of BDDs with the smallest total number of nodes, that still contains some dependencies. When this subset is found, we join these BDDs and absorb all dependencies. The number of BDDs in the set will then decrease by at least one, and we repeat the procedure until there is only one BDD left and all dependencies have been absorbed.

Divide-and-Conquer. This strategy takes the approach that it is always easy to join a *few* BDDs together and absorb all dependencies. To solve the system though, one sooner or later has to join all BDDs, and absorb all remaining dependencies. The assumption is that the true complexity of solving the system will only appear when all BDDs are joined. Thus we would like to have only a minimum of dependencies left when we are forced to join all BDDs together.

The divide-and-conquer strategy is to split the system in two (roughly) equally large halves, such that there are many dependencies within each half but only a few that use linear combinations from BDDs in both halves. We can then attack each half independently, trying to absorb all dependencies. If we succeed, we are left with one BDD with only independent linear combinations in each half. These can now be joined, and we absorb the relatively few remaining dependencies in the full BDD. When attacking one half, we use the Divide-and-Conquer technique recursively, treating the half as a complete system. The recursion stops when a "system" only contains one or two of the original BDDs.

Finding the optimal way to split a system in two equally sized parts seems to be a hard problem in general. However, knowledge of how the system has been constructed can help in this regard, as we will see with DES.

Finding Good Joining Order by Cryptanalysis. When we are trying to solve a system representing a cryptographic primitive, analysis of the primitive may help in deciding a good order of how to join the BDDs. The strategy is simply to decide on an order for the original BDDs, join all of them into one long BDD, and absorb all linear dependencies. The order of the BDDs should be such that each linear dependency only involves linear combinations on levels that are relatively close to each other. Absorbing each linear combination then becomes a somewhat local operation that only affects a small part of the BDD, keeping the complexity down.

4 Application of the Algebraic Attack Based on BDDs

This section describes practical aspects and results of the proposed attack on DES and Mini-AES as well as the time comparison of solving extended affine equivalence (EA) problem using Gröbner basis, CrytoMiniSat and BDD approaches.

4.1 A Practical Attack on Reduced DES

Previous results on solving DES systems can be found in [5] where the authors solve a 6-round version, and in [21] where DES with 6, 7 and 8 rounds are attacked. In all papers it was necessary to fix 20+ of the key bits to their correct values for the attacks to work, reducing the effective key size to at most 36 bits. The actual solving of equation systems was done by MiniSAT [22].

Our best result is that we can solve a 6-round system using 8 chosen plaintext/ciphertext pairs without fixing or guessing any variables. The average complexity is $2^{20.571}$ nodes. Solving the system for 6-round DES with 8 chosen plaintexts takes approximately one minute on a MacBook Air 2013 with 8GB of memory.

Constructing DES System of Equations. We assume the reader is familiar with the operations of DES [23]. The only non-linear part of DES is the application of the eight 6×4 S-boxes in each round. We assign variables to the output of the S-boxes in each round, except for the last two whose output can be expressed as linear conbinations of other variables and ciphertext bits. The key schedule of DES is linear, so we only need to assign variables to the 56 user-selected key bits. After this the input and output bits of each S-box can be expressed as linear combinations of variables, and we construct a BDD for each S-box as described in Section 2.2. Each BDD contains approximately 185 nodes after reduction, the eight different S-boxes vary slightly in size.

Solving Strategy. The solving strategy we found to work best for DES is Divide-and-Conquer. We then need to divide our system in two equally big halves, and the key schedule of DES gives a clear hint on how to do this: One half of the 56 key bits *only* appears in the inputs to S-boxes $1-4$, while the other half only appears in the S-boxes $5-8$. This applies to all rounds. For each round, we put the BDDs representing S-boxes $1-4$ into the set A_0, and the BDDs for S-boxes $5-8$ into the set B_0. Then we try to solve A_0 and B_0 independently, using Divide-and-Conquer again. For dividing A_0, we have found (by exhaustive search) that the best division is to group together the same two S-boxes from odd-numbered rounds, i.e. S-boxes 1 and 2, and the other S-boxes (3 and 4) from even-numbered rounds, into set A_1. The other BDDs from A_0 go into the set B_1. See Fig. 4 for a sketch of the division used. B_0 is re-divided similarly, and further divisions are done by exhaustive search on the fly.

Several Plaintexts. When using several plaintext/ciphertext pairs, we build one DES system for each. These systems will have the same 56 key-bit variables, but variables representing internal state will, in general, be different for each plaintext. However, if we carefully choose the difference between the plaintexts we can reuse a lot of internal variables across different systems. For producing up to eight different plaintexts, we vary only three bits in the left half. Then the input to the first round will be equal for each text, and the difference in the input to the second round will only be in three bits. These bits are chosen so they only affect one S-box each. In the experiments bits 1, 5 and 17 (numbering from 0 through 31) in the left half were used for generating differences, affecting S-boxes 1, 2 and 5 in the second round. Tracing differences further we find that we may reuse variables across systems as far as into the fourth round.

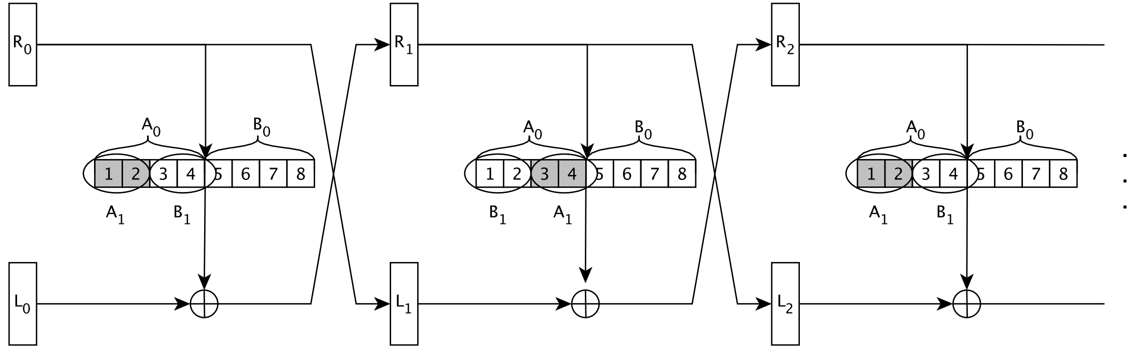


Fig. 4: Division used for Divide-and-Conquer strategy for DES.

We merge the different systems by joining all BDDs arising from the same S-box in the same round. As these share the same key variables, and often many other variables as well, there are many linear dependencies among the levels in the joined BDD. These dependencies are absorbed, and after this pre-processing we are left with a system of $8r$ BDDs representing an r-round version of DES, regardless of the number of chosen plainexts used.

Extracting Linear Equations. When using more than four plainetxts in the experiments, we observed that the heaviest step while solving did not occur when joining the sets A_0 and B_0, but rather when joining A_1 and B_1. After all dependencies in A_0 had been absorbed the resulting BDD was very slim, with many levels only containing one node.

If a level only has outgoing b-edges (which is often the case with one-node levels) we know that the associated linear combination is equal to b, ($b \in \{0, 1\}$). We can use this linear equation to eliminate a variable from the system. Extracting as many linear equations as possible and eliminating variables after all dependencies in A_0 had been absorbed, it became trivial to absorb all dependencies in B_0, and the full system.

Results for DES Experiments. We have solved systems representing DES for $4, 5$ and 6 rounds, using $1 - 8$ plaintext/ciphertext pairs. For each choice of rounds and number of plaintexts we randomly generated and solved 100 systems, recording their complexities. The minimum, maximum and average (in bold) complexities observed are summarized in Table 1.

There are rather large variations in complexities inside most cells in Table 1. In each cell, the key and one plaintext were chosen at random for each of the 100 instances, and some choices give much lower complexity than others. We can not explain the differences, and have not been able to identify which choices lead to low solving complexity.

4.2 A Practical Attack on Scaled-Down Version of AES

There are many scaled-down versions of the AES cipher (MiniAES). The first one that follows Rinjdael's description was proposed in [24]. A few years later Cid et al. analyzed many small AES variants in [7], and Elizabeth Kleiman tried to attack MiniAES in her master and doctoral theses [15, 25]. Also, an equation system representing MiniAES was solved in [16].

Table 1: Complexities for solving reduced-round DES-systems. Each cell shows the minimum, **average** and maximum complexity observed over 100 instances.

rounds \ # texts	1	2	3	4	5	6	7	8
4	$2^{22.651}$ $\mathbf{2^{22.715}}$ $2^{22.770}$	$2^{10.800}$ $\mathbf{2^{14.506}}$ $2^{17.473}$	$2^{9.281}$ $\mathbf{2^{10.606}}$ $2^{13.006}$	$2^{9.585}$ $\mathbf{2^{10.257}}$ $2^{12.029}$	$2^{9.748}$ $\mathbf{2^{9.805}}$ $2^{9.892}$	$2^{9.976}$ $\mathbf{2^{10.070}}$ $2^{10.412}$	$2^{10.103}$ $\mathbf{2^{10.203}}$ $2^{10.978}$	$2^{10.283}$ $\mathbf{2^{10.381}}$ $2^{10.446}$
5		$2^{19.472}$ $\mathbf{2^{22.110}}$ $2^{23.805}$	$2^{13.831}$ $\mathbf{2^{16.455}}$ $2^{19.329}$	$2^{11.440}$ $\mathbf{2^{13.526}}$ $2^{15.618}$	$2^{12.126}$ $\mathbf{2^{13.995}}$ $2^{16.633}$	$2^{12.289}$ $\mathbf{2^{14.212}}$ $2^{16.758}$	$2^{12.583}$ $\mathbf{2^{14.410}}$ $2^{16.882}$	$2^{12.749}$ $\mathbf{2^{14.704}}$ $2^{17.414}$
6						$2^{24.506}$ $\mathbf{2^{24.929}}$ $2^{25.352}$	$2^{22.206}$ $\mathbf{2^{22.779}}$ $2^{24.324}$	$2^{19.932}$ $\mathbf{2^{20.571}}$ $2^{21.915}$

Description of MiniAES. The constants of our studied encryption model are the block length (16 bits) and the key size (16 bits). This scaled-down cipher corresponds to AES-128 [26]. In contrast to original AES, the mini version is a nibble (4 bits) oriented cipher with the state represented as a 2×2 matrix. The round function consists of four routines: AddroundKey (AK_k), SubBytes (SB), ShiftRows (SR) and MixColumns (MC). The encryption algorithm can be described as

$$E_K(M) = \prod_{i=1}^{r} (AK_{k_i} \circ MC \circ SR \circ SB) \circ AK_{k_0}(M),$$

where r is the number of rounds. The substitution and MDS matrix was taken from [24].

Constructing System of BDDs for MiniAES. Unlike DES, MiniAES has non-linear components in the key schedule. We assign variables to the output of the S-boxes in each round, except for the last one. Additionally, we have to add 8 extra variables for each round key, except the first one with 16 bits of the user-selected key. Then the number of BDDs and variables is equal to $2r + 4r = 6r$ and $(16 + 8r) + 16(r - 1) = 24r$, respectively.

Solving Strategy. The best strategy we found for solving the MiniAES systems was to determine a good order of BDDs by cryptanalysis, join all BDDs and do a full absorption of all dependencies.

The order was found by carefully studying which variables that appear in each BDD, both from the key schedule and the encryption function. Variables from the cipher state only appear in two consecutive rounds. It is therefore clear that the four BDDs from the same round should be joined close to each other, and also close to the two BDDs from the key schedule producing the round key used for the round. The BDDs were put together in groups of six this way, following the rounds of the cipher.

The order of the BDDs in each group was determined by looking at which variables that appear in each individual BDD. Two BDDs that share many variables should be adjacent in the final order. After doing this for the four, five and six round versions it became clear that

a pattern emerged for the joining order. This pattern was followed for the higher number of rounds.

Results for MiniAES Experiments. The complexities for solving MiniAES systems for various rounds are summed up in Table 2. Of course, MiniAES only has a 16-bit key and can be broken very fast using for instance CryptoMiniSAT, which essentially does a very intelligent brute force on the key [27, p. 250-251]. In [16], the authors create the polynomials for the ciphertext bits using only the user-selected key bits as variables. This results in 16 polynomials in 16 variables containing approximately 2^{15} terms each, that can be solved using PolyBoRi. Our algorithm does not take advantage of the short key, and should be compared to the earlier attacks in [7, 15] that also does not exploit the short key.

For each number of rounds we solved 10 different instances, using 1 known plaintext/ciphertext pair. We were not able to reduce the complexities by using more pairs. The observed complexities for one particular number of rounds did not vary, so the minimum, maximum and average complexities are all the same. We have also changed the S-box and the MixColumn matrix to see if other choices affected the complexity, but we found the complexity remains the same for all variants tried.

It has been observed before that for one plaintext/ciphertext pair in scaled-down AES versions there may be more than one key that encrypts the given plaintext into the given ciphertext. This was shown in our experiments as well, often we had two, three or even four solutions to our systems.

Table 2: Complexities of solving MiniAES systems.

Rounds	4	5	6	7	8	9	10
Complexity	$2^{22.404}$	$2^{23.051}$	$2^{23.440}$	$2^{24.154}$	$2^{24.217}$	$2^{24.862}$	$2^{24.961}$

4.3 Problem of determining EA-equivalence

To get a good comparison against other solvers we have chosen the problem of EA-equivalence [29]. This problem is interesting in cryptography, and it can be solved via non-linear systems of equations. There are no special variables in these systems, like key bits, so we get a fair comparison between CryptoMiniSAT, Gröbner bases and the BDD method.

Two functions are EA-equivalent if the following equation holds for all $\mathbf{x} \in GF(2)^n$

$$F(x) = M_1 \cdot G(M_2 \cdot x \oplus V_2) \oplus M_3 \cdot x \oplus V_1, \tag{1}$$

where elements of $\{M_1, M_2, M_3, V_1, V_2\}$ have dimensions $\{m \times m, n \times n, m \times n, m, n\}$ and M_1 and M_2 are non-singular [29]. For simplicity, we set $n = m$. The EA-equivalence problem can then be formulated as follows:

For given functions $F, G : GF(2)^n \mapsto GF(2)^n$ find M_1, M_2, M_3, V_1, V_2 such that (1) holds or show that such vectors and matrices do not exist.

The variables in the system to be solved are the entries in M_1, M_2, M_3, V_1 and V_2, so the number of variables is $3n^2 + 2n$. The maximum number of equations and a system's degree can be calculated theoretically for given F and G [30]. However, for $n \leq 6$ the system can always

be made quadratic by introducing the matrix $M_3' = M_1^{-1} \cdot M_3$ and the vector $V_1' = M_1^{-1} \cdot V_1$ and add them as additional equations and variables.

For $n = 4$ and $n = 5$ the problem of EA-equivalence is tractable, and the complexity comparison of Gröbner basis (GB), CryptoMiniSat (SAT) and proposed approach (BDD) is presented in Table 3 for five different instances. Unlike the two other methods, CryptoMiniSat only finds one solution by default. Therefore, we have used the option "n = +infinity" in CryptoMiniSat to force it to find all solutions and get a fair comparison. For GB we tested with several options for ordering, with or without using "faugere" option (turning linear algebra on or off), and with or without parallelization. The smallest observed times we could achieve are reported below.

Table 3: Time complexity for solving EA-equivalence problem

#	n	Number of solutions	Seconds used to solve		
			BDD	GB	SAT
1	4	2	$2^{4.05}$	$2^{1.30}$	$2^{13.71}$
2	4	60	$2^{4.86}$	†	$2^{16.77}$
3	4	2	$2^{3.92}$	$2^{1.01}$	$2^{12.08}$
4	5	1	$2^{10.20}$	$2^{11.43}$	-
5	5	155	$2^{10.48}$	†	-

In Table 3 "†" means that the Gröbner bases solver implemented in Sage crashes after several minutes with out-of-memory message, and "–" means that CryptoMiniSat spent more than 78 hours (2^{18} seconds) without finding a solution [28].

5 Conclusions

In this paper we have explained an approach to build and solve equation systems using binary decision diagrams and reported on experiments with the method. The best previous results on algebraic attacks against DES use guessing at least 20 of the key bits. We improved on these results using BDDs, breaking six rounds of DES without guessing any variables.

For MiniAES we have also received results which are better than the previous algebraic attacks described in [7] and [15]. According to our experiments a system representing 10 rounds of MiniAES can be solved in 45 minutes on a ordinary computer using the BDD approach. However, attacks exploiting the short key have lower complexities. At the same time, the BDD method is shown to be advantageous compared to Gröbner basis and CryptoMiniSat on solving the EA-equivalence problem.

These experiments indicate the BDD approach can compete with other methods in applications which require solving the non-linear equation systems. There are several open questions to address in future research. Does there exist a generic algorithm giving an order of BDDs that yield low complexity when applying linear absorption? Is it possible to analytically estimate the complexity of solving a BDD system of equations, or do we have to actually run the solver to find out? Which ciphers are most vulnerable against this type of algebraic attacks? We hope the potential of BDDs in cryptanalysis will be thoroughly examined in future research.

References

1. Shannon, C.E.,*Communication Theory of Secrecy Systems*, Bell system technical journal 28, no. 4, pp. 656–715 (1949).
2. Raddum, H., Semaev, I., *Solving multiple right hand sides linear equations*, Designs, Codes and Cryptography, Volume 49, no. 1-3, pp. 147–160, Springer (2008).
3. Courtois, N.T., *Fast algebraic attacks on stream ciphers with linear feedback*, CRYPTO 2003, LNCS 2729, pp. 176–194, Springer (2003).
4. Courtois, N.T., Bard, G.V., Wagner, D., *Algebraic and slide attacks on KeeLoq*, Fast Software Encryption 2008, LNCS 5086, pp. 97–115, Springer (2008).
5. Courtois, N.T., Bard, G.V., *Algebraic cryptanalysis of the data encryption standard*, Cryptography and Coding, LNCS 4887, pp. 152–169, Springer (2007).
6. Helleseth, T., Rønjom, S., *Simplifying algebraic attacks with univariate analysis*, Information Theory and Applications Workshop (ITA), pp. 1–7, IEEE (2011).
7. Cid, C., Murphy, S, Robshaw, M.JB., *Small scale variants of the AES*, Fast Software Encryption 2005, LNCS 3557, pp. 145–162, Springer (2005).
8. Bard, G.V., *Algebraic cryptanalysis*, Springer (2009).
9. Albrecht, M., *Algorithmic Algebraic Techniques and Their Application to Block Cipher Cryptanalysis*, University of London, Thesis (2010). http://www.sagemath.org/files/thesis/albrecht-thesis-2010.pdf.
10. Daum, M., *Cryptanalysis of Hash functions of the MD4-family*, Ruhr University Bochum, Thesis (2005). http://www.cits.ruhr-uni-bochum.de/imperia/md/content/magnus/dissmd4.pdf
11. Lee, C.Y., *Representation of Switching Circuits by Binary-Decision Programs*, Bell Systems Technical Journal 38, pp. 985–999 (1959).
12. Knuth, D.E., *The Art of Computer Programming Volume 4, Fascicle 1: Bitwise tricks and techniques; Binary Decision Diagrams*, Pearson Education India (2009).
13. Krause, M., *BDD-based cryptanalysis of keystream generators*, EUROCRYPT 2002, LNCS 2332, pp. 222–237, Springer (2002).
14. Stegemann, D., *Extended BDD-based cryptanalysis of keystream generators*, Selected Areas in Cryptography 2007, LNCS 4876, pp. 17–35, Springer (2007).
15. Kleiman, E., *High Performance Computing techniques for attacking reduced version of AES using XL and XSL methods*, Graduate Theses and Dissertations (2010) http://lib.dr.iastate.edu/etd/11473
16. Bulygin, S., Brickenstein, M., *Obtaining and Solving Systems of Equations in Key Variables Only for the Small Variants of AES*, Mathematics in Computer Science 3(2), pp. 185-200, BirkhÃďuser-Verlag (2010)
17. Bryant, R.E., *Graph-based algorithms for boolean function manipulation*, IEEE Transactions on Computers, Volume 35, no. 8, pp. 677–691 (1986).
18. Schilling, T, Raddum, H., *Solving Compressed Right Hand Side Equation Systems with Linear Absorption*, SETA 2012, LNCS 7280, pp. 291–302, Springer (2012).
19. Rudell, R., *Dynamic Variable Ordering for Ordered Binary Decision Diagrams*, 1993 IEEE/ACM International Conference on Computer-aided Design, vol. 12, pp. 42–47 (1993).
20. Bollig, B., Wegener, I., *Improving the variable ordering of OBDDs is NP-complete*, IEEE Transactions on Computers, Volume 45, no. 9, 993-1002 (1996).
21. Perret, J.C F.L., Spaenlehauer, P.J., *Algebraic Differential Cryptanalysis of DES*, Western European Workshop on Research in Cryptology (2009). http://www.pjspaenlehauer.net/data/papers/DESweworc.pdf
22. Eén, N., Sörensson, N., *An extensible SAT-solver*, Theory and Applications of Satisfiability Testing, LNCS 2919, pp. 502–518, Springer (2004).
23. *Data Encryption Standard*, Federal Information Processing Standards Publication (FIPS PUB) 46, National Bureau of Standards, Washington, DC (1977).
24. Phan, R. C.-W., *Mini advanced encryption standard (Mini-AES): A testbed for cryptanalysis students*, Cryptologia, Volume 26, no. 4, pp. 283–306, Taylor & Francis (2002).
25. Kleiman, E., *The XL and XSL attacks on Baby Rijndael*, Master Thesis, Iowa State University (2005). https://orion.math.iastate.edu/dept/thesisarchive/MS/EKleimanMSSS05.pdf
26. *Announcing the Advanced Encryption Standard (AES)*, Federal Information Processing Standards Publication 197, United States National Institute of Standards and Technology (NIST) (2001).
27. Soos, M. , Nohl, K., Castelluccia C., *Extending SAT Solvers to Cryptographic Problems*, SAT 2009, LNCS 5584, pp. 244 – 257, Springer (2009)

28. Stein, W., et al.: Sage mathematics software (version 6.2). The Sage Development Team (2014). `http://www.sagemath.org`.
29. Budaghyan, L., Kazymyrov, O.: Verification of restricted EA-equivalence for vectorial Boolean functions. In Özbudak, F., Rodríguez-Henríquez, F. (eds.), Arithmetic of Finite Fields, vol. 7369 of Lecture Notes in Computer Science, pp. 108–118. Springer Berlin Heidelberg, 2012.
30. Eilertsen, A. M., Kazymyrov, O., Kazymyrova, V., Storetvedt, M.: A Sage library for analysis of nonlinear binary mapping. In Pre-proceedings of Central European Conference on Cryptology (CECC14), pp. 69–78. 2014.