Extending SAT Solvers to Cryptographic Problems

Mate Soos[†], Karsten Nohl[‡], and Claude Castelluccia[†]

[†]INRIA Rhône-Alpes, [‡]University of Virginia

Abstract. Cryptography ensures the confidentiality and authenticity of information but often relies on unproven assumptions. SAT solvers are a powerful tool to test the hardness of certain problems and have successfully been used to test hardness assumptions. This paper extends a SAT solver to efficiently work on cryptographic problems. The paper further illustrates how SAT solvers process cryptographic functions using automatically generated visualizations, introduces techniques for simplifying the solving process by modifying cipher representations, and demonstrates the feasibility of the approach by solving three stream ciphers.

To optimize a SAT solver for cryptographic problems, we extended the solver's input language to support the XOR operation that is common in cryptography. To better understand the inner workings of the adapted solver and to identify bottlenecks, we visualize its execution. Finally, to improve the solving time significantly, we remove these bottlenecks by altering the function representation and by pre-parsing the resulting system of equations.

The main contribution of this paper is a new approach to solving cryptographic problems by adapting both the problem description and the solver synchronously instead of tweaking just one of them. Using these techniques, we were able to solve a well-researched stream cipher 2^6 times faster than was previously possible.

1 Introduction

Cryptographic functions are at the base of computer security with encryption ciphers ensuring confidentiality and authenticity. Despite their importance, many practical cryptographic functions rely on unproven assumptions about the complexity of their underlying mathematical problems. When these assumptions are found to be incorrect, new theoretical and practical attacks are constructed that sharpen the understanding of a specific problem and advance the evolution of cryptography in general. SAT solvers have been shown to be a powerful tool in testing mathematical assumptions. In this paper, we extend SAT solvers to better work in the environment of cryptography.

Previous work on solving cryptographic problems with SAT solvers has concentrated on the best mathematical representation of ciphers [1]. To further improve the potential of SAT solvers, we adapted a SAT solver to better suit cryptographic problems and then manipulated the representation of some cryptographic

problems to best fit this modified solver. We refined SAT solvers to understand the XOR operation, which is common in cryptography, besides functions in the conjunctive normal form (CNF) that is native to many SAT solvers. We further added dynamic behavior analysis to more thoroughly understand the workings of SAT solvers on cryptographic primitives.

To show the effectiveness of our approach, we solved a few different ciphers. The first two targets, Crypto-1 [2] and HiTag2 [3], are weak stream ciphers, widely used in electronic payment, access control and car immobilizers. Our third target, Bivium [4], is a simplified version of the eSTREAM standard stream cipher Trivium [5] known for its simple description. Solving these ciphers with an unmodified SAT solver and with only basic improvements to their CNF representation reveals the secret state within 170 hours for Crypto-1, a week for HiTag2 and in $2^{42.7}$ s [6] for Bivium. With our adapted SAT solver and tuned cipher description techniques, the average attack time on a desktop PC drops to 40 seconds for Crypto-1, 6.5 hours for HiTag2 and $2^{36.5}$ s for Bivium.

Contributions

We optimize a standard SAT solver for cryptographic problems in Sect. 3. The SAT solver now handles XOR operations natively to faster solve cryptographic problems and the solver's execution is visualized to allow insight into its inner workings. Based on these improvements, guidelines are derived on how to convert ciphers to a description that can be quickly solved in Sect. 4. Finally, three ciphers are solved using the adapted SAT solver faster than was previously possible with other SAT solver-based techniques, in Sect. 5.

2 Background

Our results build on research in stream ciphers, SAT solvers, and algebraic cryptanalysis. This section presents the current state of research in these areas and indicates where they connect.

2.1 Stream ciphers

A stream cipher is a symmetric cryptographic function that allows two parties to communicate privately when they share a secret key. Stream ciphers produce a stream of pseudorandom bits (the *keystream*) given a secret *key* and a non-secret random initialization vector (IV). This key stream is XORed with a message prior to sending and again XORed after receiving so that the message cannot be read while in transit.

The stream ciphers discussed in this paper are based on one or more shift registers with linear or non-linear feedback function as well as a filter function that maps the register states to keystream bits. Stream ciphers have two phases: an *initialization phase* followed by a *keystream generation phase*. During initialization, key and IV are typically mixed to become the initial state by shifting the registers

while feeding in a combination of the feedback function and the filter function. During keystream generation, the registers are shifted and their feedback function is applied, while the keystream is generated from the state using the filter function.

2.2 SAT solvers

Satisfiability solvers are programs that employ highly optimized mathematical algorithms to decide whether a set of constraints have a solution. This paper only discusses one widely-used constraint set, the conjunctive normal form (CNF). In CNF, each element in the constraints, a or $\bar{a}$, is called a literal. A clause is a disjunction (`or`-ing) of literals. CNF is a conjunction (`and`-ing) of clauses. Hence, the constraints are presented to the SAT solver as an "and of ors".

SAT solvers are mostly used in electronic design automation (EDA), though they are also used in a growing number of other domains. State-of-the-art solvers have been extended or adopted to meet the specific characteristics of different problem domains, for example temporal induction in [7].

Modern SAT solvers that are based on the DPLL algorithm [8] evolved from GRASP [9] which introduced learning, and later from from Chaff [10] which introduced watched literals and dynamic variable ordering. Solvers that employ these techniques are called *conflict-driven* SAT solvers. In this paper we extend MiniSat [11], a conflict-driven SAT solver designed for researchers to adapt it to different domains.

MiniSat employs a backtracking-based, depth-first search algorithm to find a satisfying variable assignment for a system of clauses. The algorithm branches on a variable by guessing it to `true` or `false` and examining whether the value of other variables depends on this guess. The affected variables are then assigned and the search continues until no more assignments can be made. During this period, called *propagation*, a clause may be found that cannot be satisfied anymore. If such a *conflict* is encountered, a *learned clause* is generated that captures the wrong guesses that lead to the conflict. The topmost guess allowed by the learned clause is then reversed and the algorithm continues. The learned clauses trim the search tree and guide the algorithm in choosing the best next guess. Eventually, either a satisfiable assignment is found or the search tree is exhausted, meaning that no solution exists.

2.3 Algebraic Cryptanalysis

Algebraic cryptanalysis is a family of attacks that exploits insufficient complexity in ciphers. These attacks have successfully been applied to break a number of ciphers secure against other forms of cryptanalysis. In algebraic attacks, equations are constructed that express the output bits of a cipher in terms of its inputs, or its state. These equations are then solved and reasoned about with either dedicated equation solvers such as the F5 algorithm [12], or standard SAT solvers.

The first SAT-based cryptanalysis was by Massacci et al. [13], experimenting with the Data Encryption Standard (DES) using DPLL-based SAT solvers. More recent work by Courtois and Bard has produced attacks against KeeLoq [1]

and stream ciphers with linear feedback [14]. Algebraic cryptanalysis has also been used on modern stream ciphers, such as the reduced version of Trivium, Bivium [4].

3 Adapting the SAT solver

To take full advantage of the power of SAT solving we adapted and optimized MiniSat, a state-of-the-art DPLL-based SAT solver, for algebraic cryptanalysis. We further added visualization to the solver to help identify bottlenecks and improve the solving by altering the problem representation. Among the many choices for modern SAT solvers, we chose MiniSat for its competitive performance, code availability, and a design that specifically encourages extensions to its input language.

3.1 XOR support

Cryptographic building blocks such as filter and feedback functions lead to equations with many XORs. These XOR constraints, when converted to CNF representation without further elaboration, grow exponentially in size. This is because the XOR constraint's Karnaugh table contains 2^{len-1} minterms, and hence needs 2^{len-1} clauses to describe in CNF.

To circumvent this limitation, previous research extended the Satz solver to reason about 2- and 3-long XOR constraints, which they called equivalency reasoning [15]. For MiniSat, previous research [1, Sect. 6.4] cut up the XOR function into groups of smaller XORs, each setting an additional variable. The full XOR was then represented as a XOR of the additional variables.

While cutting up XORs allows MiniSat to work on long XOR chains, this approach forces the solver to watch and examine many clauses for variable changes, when in fact only one XOR constraint should be watched. To mitigate this limitation, we implemented the XOR constraint natively into MiniSat. Each XOR constraint is represented by a single *xor-clause*. A xor-clause behaves as a regular clause towards all unchanged parts of the solver: it dynamically changes appearance when propagating or causing a conflict by appearing as a different regular clause depending on the current assignment of variables.

For example, the xor-clause $a \oplus b \oplus c$ represents all the regular clauses

$$a \vee \overline{b} \vee \overline{c} \quad (1) \qquad \overline{a} \vee \overline{b} \vee c \quad (2)$$
$$a \vee b \vee c \quad (3) \qquad \overline{a} \vee b \vee \overline{c} \quad (4)$$

and if, for example $a =$ `true` and $b =$ `true`, then it changes its appearance to the regular clause (2), and causes the propagation $c =$ `true` just as its regular representation would. If, however, $a =$ `false`, $b =$ `true` and $c =$ `true`, the xor-clause changes its appearance to regular clause (1) and causes a conflict just as its regular representation would.

Generating a conflicting or propagating clause from a xor-clause is done as follows. All variables that are assigned to `false` are included as-is, and all

variables that are assigned to `true` are included in a negated form. If propagating, the single unassigned variable is also included, its negation depending on the values of the other variables in the xor-clause.

Solving cryptographic functions is accelerated considerably by integrating xor-clauses into MiniSat. For the stream ciphers Crypto-1 and Grain solving is up to twice as fast with xor-clauses and memory usage is decreased by at least an order of magnitude.

Besides speeding up the solving, native XOR support leads to more concise input file and internal data structures, which simplify analyzing the dynamic behavior of the solver. Lastly, xor-clauses enable a straightforward implementation of Gaussian elimination into MiniSat as explained in the next section.

3.2 Gaussian elimination

Gaussian elimination is an efficient algorithm for solving systems of linear equations. Since each xor-clause is a linear equation, we can use this algorithm to solve the system of equations described by the xor-clauses. Some linear problems with as many as 100 variables can be trivially solved with Gaussian elimination but take an excessive amount of time when solved with SAT solvers. This phenomenon is due to the fact that SAT solvers solve by guessing variables and determining if there is any equation that gives a result given the current assignments. If the set of linear equations is dense (i.e. all equations contain many variables), almost all variables need to be guessed before any equation gives a result. Thus, for a system with 100 variables, it is not uncommon that 80 variables need to be guessed before any equation gives a result, i.e. the search space is on the order of 2^{80}. When using Gaussian elimination, on the other hand, the same problem can be solved in less than 2^{20} operations.

Since Gaussian elimination and the DPLL algorithm (used in MiniSat) are optimal for different parts of cryptographic problems, the best results are achieved by switching between the two. To benefit from Gaussian elimination during solving, whenever the SAT solver cannot perform any further propagations and would need to guess a variable, we ask the Gaussian elimination if there is any information it could extract from the xor-clauses.

Execution of the Gaussian elimination gives one of the following results: either it finds nothing, or it finds that a variable can be propagated by a combination of xor-clauses, or it finds that given the current assignments, the system of equations is unsatisfiable. In the two latter cases, the solver needs the actual xor-clause, which when evaluated with the variable assignments, gives a unit or empty clause, respectively. This actual xor-clause is important, as it signals the solver what variable was propagated by what clause (in case of a propagation), or what clause caused the conflict (in case of a conflict). To calculate the actual xor-clauses, we keep two matrixes: one updated with the current assignments, and one that mirrors the other only with its row-swap and row-xor operations. Whenever there is a propagation or conflict indicated by the first matrix, the second matrix is

used to generate the actual xor-clause. For example, if the two matrixes are:

xor-clauses with $v8$ assigned to `true`

$v10$	$v8$	$v9$	$v12$	aug
1	–	1	1	1
0	–	1	1	1
0	–	0	1	0
0	–	0	0	0

actual xor-clauses

$v10$	$v8$	$v9$	$v12$	const
1	1	1	1	0
0	0	1	1	1
0	1	0	1	1
0	1	0	0	1

then the second to last row of the first matrix indicates propagation of $v12 =$ `false`. The actual xor-clause can be read from the second matrix: it is $v8 \oplus v12 \oplus 1$. The matrix used by the Gaussian elimination algorithm is upper triangular, but the matrix containing the actual xor-clauses is only upper triangular for the columns representing variables that are not assigned.

Including Gaussian elimination into MiniSat is based on the idea of *SAT Modulo Theories* (SMT). An SMT instance is a generalization of a Boolean SAT instance in which various sets of variables are replaced by predicates from a variety of underlying theories. Naturally, SMT formulas provide a much richer modeling language than is possible with Boolean SAT formulas. In essence, xor-clauses enrich MiniSat's language, which the Gaussian elimination can understand and reason about, tightly integrating its conclusions into the DPLL algorithm of MiniSat.

A trade-off parameter for the Gaussian elimination is the cut-off depth until which it is worthwhile to execute the algorithm. Cutting off branches at the top reduces the search space more than cutting at the bottom, but it takes approximately the same time to execute the algorithm. However, if the cut-off depth is too shallow, the constant overhead is more than the benefit, but if too deep, the dynamic overhead is more than the benefit. In the end, we made the cut-off depth configurable, and ran tests to decide for each cipher which depth gave the most benefit.

To save time, the matrix is incrementally normalized as the solver travels down the search tree and assignments are made. We save the matrix at every search depth, and in case the solver has to jump back (due to a conflict), we re-load the matrix from the state saved at that depth.

Using Gaussian elimination, solving Bivium and Trivium is faster by 1-5% if we restrict the search depth to between 1 and 8, depending on the number of guessed bits. For other instances derived from other ciphers, Gaussian elimination does not appear to decrease the overall solving time. A comparative figure for Bivium, showing the speed of solving and the explored search space versus the depth until which the algorithm was active is present in Fig. 1. It is apparent from the graphs that using Gaussian elimination reduces the explored search space (in the example, by up to 83%), but the algorithm takes more and more time to execute as the cut-off depth is increased.

Apart from the marginal speedup that Gaussian elimination brings, it is a useful tool for multiple other reasons. First of all, it demonstrates that SAT

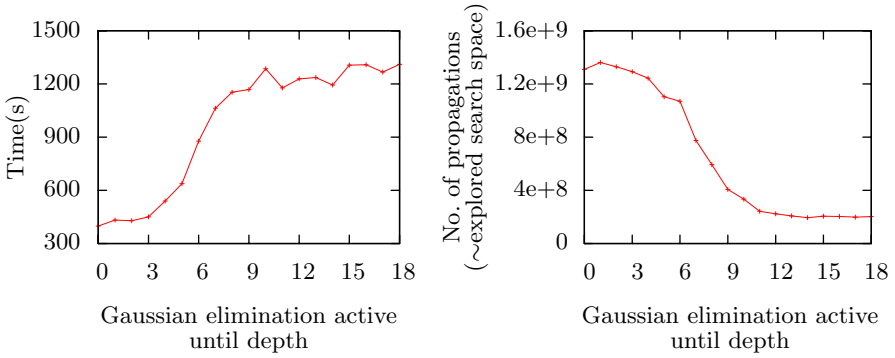

Fig. 1. Comparison between the time and the number of propagations (∼explored search space), relative to the depth until which the Gaussian elimination was active. Each point in the graphs represent 2000 random examples of the Bivium cipher, given 56 randomly guessed state bits

solvers ignore certain characteristics of the problem they are dealing with, and by exploiting these properties the search space could be significantly reduced. Secondly, the combined solver works much faster on problems that have large parts that can benefit from Gaussian elimination. Lastly, our implementation of Gaussian elimination can likely be improved upon leading to greater speedups.

3.3 Dynamic behavior analysis

The dynamic behavior of SAT solvers is hard to follow since branching and propagation occur far too many times to be traceable by hand. Understanding the solver's dynamic behavior, however, is essential for estimating a cipher's complexity and for improving the solver's performance.

To better understand how MiniSat reaches solutions, we implemented search tree tracing into the solver. The output of our MiniSat trace extension can be analyzed visually and statistically. Visualizing the operation of DPLL-based SAT solvers was introduced in [16], which our implementation augments in multiple ways. Our extension allows for variables to be named and for clauses to be grouped, which is useful when multiple clauses are used to represent one logical entity (e.g. a feedback function). The calculated statistics include the type of most conflicted clauses (e.g. filter functions), the average number of propagations per search tree branch, etc. An example search tree of the Crypto-1 cipher is in Fig. 2. The visualization allowed us, for instance, to identify the regularly placed filter function taps of Crypto-1 as its largest weakness over an improved variant found in HiTag2 tags.

It is clear from the variable branching statistics that during solving, the most important variables are picked automatically by MiniSat, which are always the state or key bits. By examining smaller search graphs and the statistics on the most conflicted clause groups, we further found that once the important

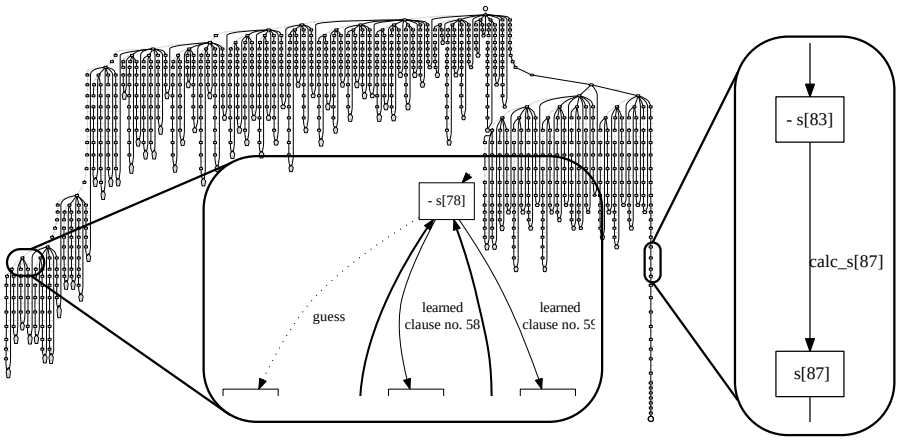

Fig. 2. Graphviz visualization of an example search for the Crypto-1 cipher's states. The tree is read from left to right, top to bottom: the left- and bottommost pentagon is the first conflict clause, the right- and bottommost circle is the satisfying assignment.

variables have been guessed, the results of these assignments are propagated to the equations representing the known keystream bits, and if they do not match, a conflict occurs, a guess is reversed, and the algorithm starts again.

The solver's strategy is therefore similar to a brute force search in which all key or state bits are tried. If one or more keystream bits can be evaluated without knowing all state bits, the SAT solver will evaluate them, and if the equations do not work out, stop the computation there, effectively doing partial evaluation. Furthermore, clauses are learnt during the search, which later prune the search tree, helping to perform partial evaluation.

The lessons learnt from search-tree tracing are as follows. It is best not to include long initialization sequences (such as that used by Grain) in the equations since after initialization all keystream bits depend on all key bits. This forces the solver to calculate a large part of the cipher in an ineffective way, as its description and subsequent evaluation in the solver is more complicated than the way the cipher was originally meant to be calculated.

A stream cipher is considered broken if its state can be determined at any point during keystream generation. Therefore, instead of making initialization part the problem, its state at a suitable point should be treated as the unknown, as this is the only possible way to take advantage of the partial evaluation property of SAT solvers. Although this state is larger than the key for all modern ciphers, it is relatively easy to solve a large part of it, as the keystream bits depend much more directly on the selected state's bits.

4 Adapting the cipher representation

Finding the best representation of stream ciphers in regular and xor-clauses is a crucial step in breaking a cipher with SAT solvers [1, Sect. 8]. For the techniques

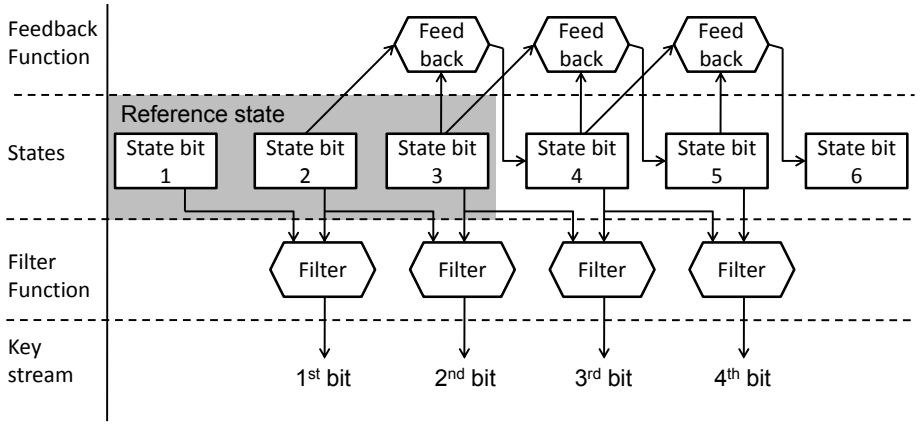

Fig. 3. Logical circuit representation of an example stream cipher: The cipher has a 3-bit shift register, whose filter function depends on the first two bits in the register, and whose feedback function depends on the last two bits in the register.

in this paper, a cipher is described as a logical circuit with functions, variables, the known keystream, and known inputs.

4.1 Logical circuit representation

In the logical circuit representation used in our approach, the unknown is the reference state's bits, and functions are expressed in regular and xor-clauses, using variables as input. An example logical circuit for a 3-bit state stream cipher is given in Fig. 3. In the figure, the cipher produces four keystream bits, and the shift register is shifted three times, using the feedback function. Functions are shown as hexagons, variables as simple boxes, and the reference state is marked in gray.

The *depth* of a keystream bit is the number of distinct functions (resp. hexagons) traversed on the way from the keystream bit to the reference state bits. For example, on Fig. 3. the 1^{st} keystream bit's depth is one, while the 4^{th} keystream bit's is four. Since the solver guesses the reference state's bits, the depth of the circuit indicates the number of functions that must be evaluated by the solver to realize that a wrong guess was made for a given keystream bit. Therefore, the shallower the overall depth of the circuit, the faster the solving. The difficulty hidden behind the functions (resp. hexagons) is also relevant, as when traversed, these must be calculated. If the number and length of clauses representing these hexagons are large, the solver is slowed down considerably. Finally, the number of reference state bits each keystream bit depends on plays an important role during solving, as a large part of these must be guessed before evaluation can take place. This *dependency number* can be calculated by simply

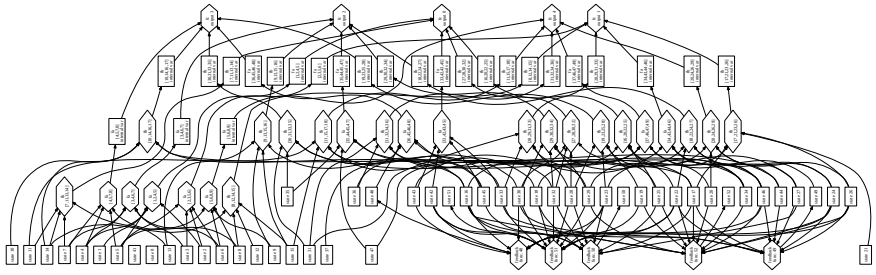

Fig. 4. Clause- and variable-dependency graph of HiTag2. Clause groups are represented as hexagons, and variables as boxes. The known keystream bits are the 5 final filter functions at the top, and the feedback functions are the 5 hexagons at the bottom right.

traversing the graph in a breath-first search fashion from the keystream up. The lower this number, the faster the solving.

The difficulty hidden behind the functions (resp. hexagons) is also relevant, as they must be calculated whenever traversed. If the number and length of clauses representing these hexagons are large, the solver is slowed down considerably. Finally, the number of reference state bits each keystream bit depends on plays an important role during solving, as a large part of these must be guessed before evaluation can take place. This *dependency number* can be calculated by simply traversing the graph in a breath-first search fashion from the keystream up. The lower this number, the faster the solving.

To summarize, when attempting to represent a cipher, the depth of the resulting logical circuit, the number of reference state bit dependencies and the complexity of the traversed functions' representations must all be optimized to maximize solving speed.

4.2 Generating the logical circuit representation

To evaluate the effectiveness of different representations of the same stream cipher, we extended MiniSat by a tool that generates the logical circuit's description. Given some additional information in the input language, the circuit is visualized with `Graphviz` or statistically analyzed to calculate keystream bit depths and state-bit dependencies. In the generated circuit, just as in the search tree, clauses are grouped into logical elements (such as a filter function), and variables are named (such as reference state bit). An example visualization of HiTag2's logical circuit representation is in Fig. 4.

Having the logical circuit representation allowed us to implement a dependency-tree walker that removes functions whose output does not contribute to any keystream bits, e.g. the last feedback function in Fig. 3. The method used is in essence the same that is used in electric circuit design to remove unused elements, applied to the domain of SAT-based cryptanalysis. Removing useless functions gives only a minor speedup of about 1%, however, unnecessary functions

no longer show up on the dynamic behavior analysis statistics, which helps in understanding the inner workings of the solver.

4.3 Optimizing the representation of LFSRs

Most stream ciphers contain one or more linear feedback shift registers (LFSR). For these ciphers, the state bits not in the reference state can be either be deduced by continually applying the forward and backward feedback functions or be directly calculated from the reference state's bits. This latter option increases the interdependency of the resulting equations, which helps the solver generate learned clauses that are useful for a larger part of the search tree. These learnt clauses are then used later to avoid useless branches of the search tree, reducing the overall search time.

To generate r keystream bits, r distinct states are needed since generating the n-th keystream bit requires the filter function to be applied to the n-th state. For the solving to be fast, we need to choose the reference state that generates the least complex logical circuit representation. In particular, we must minimize both the average depth and the reference state bit dependencies. According to our experience, this optimal reference state is usually near the $r/2$-th state. As an example, if we had taken state 2 (i.e. state bits 2 to 4) as reference in Fig. 3., the overall depth of the circuit would have been reduced.

4.4 Optimizing the representation of non-linear functions

For efficient solving, the number of clauses, the average clause length, and the number of variables should all be low, but often there exists a trade-off between the three properties.

As an example, the simple $\mathbb{GF}(2)$ polynomial

$$x_1 \oplus x_1x_2 \oplus x_2x_3 \oplus x_1x_3$$

has a Karnaugh table presentation in CNF of

$$\bar{x}_1 \vee \bar{x}_3 \qquad \bar{x}_2 \vee x_3 \qquad x_1 \vee x_2$$

However, the same polynomial can be represented with each non-single monomial expressed as a function, setting additional variables $i_1 \ldots i_3$. The polynomial then becomes

$$x_1 \oplus i_1 \oplus i_2 \oplus i_3$$

Using this representation, the number of clauses increases to 3×3 regular + 1 xor-clause, and the average clause length increases to 4.14. Three extra variables also need to be added, diluting the possible learnt clauses with extra variables, thus reducing the effectiveness of learning.

The trade-offs between the two representation methods are complex; from our experience with Grain, Trivium, Crypto-1 and other ciphers, we find that the

Table 1. Running times for solving Crypto-1, HiTag2, and Bivium

	Vanilla MiniSat	Karnaugh optimization	Karnaugh and xor-clause optimizations
Crypto-1	500 s	72 s	40 s
HiTag2	$2^{17.8}$ *s*	2^{15} s	$2^{14.5}$ s
Bivium	$2^{36.7}$ s	$2^{36.7}$ s	$2^{36.5}$ s

Karnaugh-table representation works well for functions that contain few (up to 5-6) variables and where these variables are often repeated in many monomials. For instance, solving HiTag2 and Crypto-1 are both sped up by a factor of up to 9x using the Karnaugh table representation.

When a polynomial can be broken up into sub-functions that do not share variables among themselves, such as the polynomials representing the filter functions of Crypto-1 and HiTag2, then these sub-functions must be modelled separately. This increases the overall depth of the resulting logical circuit, however, the complexity of the individual functions traversed during solving is much lower, which is crucial for the solver.

5 Implemented Attacks

The extended SAT solver can solve many stream ciphers. Attacks against three ciphers have been implemented that are faster than any previous SAT solver-based attacks. The first two targets, Crypto-1 and its relative HiTag2, are a weak ciphers used in contactless cards and car immobilizers. The third target, Bivium, is a simplified version of Trivium, a modern cipher standardized through the eSTREAM competition. The solving times for Crypto-1, HiTag2, and Bivium are in Table 1, and their detailed discussion is below.

5.1 Crypto-1 and HiTag2

The Crypto-1 stream cipher [2] is implemented on the NXP Mifare Classic card, which is widely used for micropayment in public transport and for building access control. The cipher was designed to have a particularly small hardware footprint consisting of an 48-bit LFSR and a network of small binary functions that form the filter function. HiTag2 [3], used in car immobilizers, is a relative of Crypto-1, and shares its structure but uses different feedback and filter functions.

The security of Crypto-1 has already been broken using MiniSat by Courtois et al. [17]. They did not publish the details of their attack but only stated that secret keys can be found within 200 seconds on average on a PC given 56 bits of keystream. Their attack, however, modifies the equations describing the cipher by mathematical means, which makes their techniques mostly orthogonal to ours. With our method, solving Crypto-1 using 56 bits of known keystream takes 40 s, while solving HiTag2 given the same number of keystream bits takes $2^{14.5}$ s.

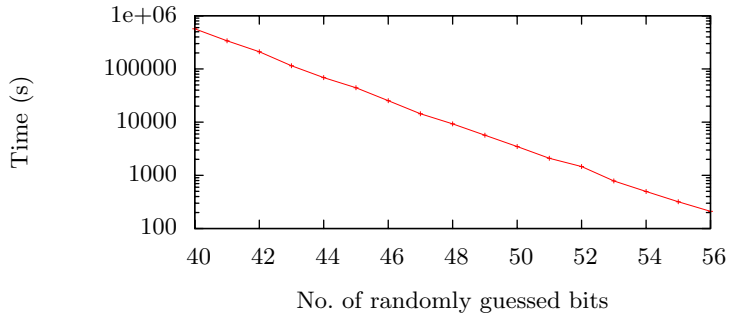

Fig. 5. Solving the Bivium cipher 1000 times, with randomly selected and assigned guess bits. The time to solve is exponential in the number of guess bits.

5.2 Bivium

The Bivium stream cipher [4], is a reduced version of the original Trivium cipher, and is intended to be used solely as a research tool to analyze the original cipher. The papers that have been published on this cipher [4, 6] improve on each other's results, the best of which is solving in $2^{42.7}$ s on a desktop machine.

Bivium can be solved by describing it in MiniSat using the enhancements and insight presented in this paper. To let the solver finish within reasonable time, we randomly guessed some randomly picked reference state bits and did a thousand different runs for each configuration. With this approach the time to solve is exponential in the number of guessed state bits, as illustrated in Fig. 5. Due to the large amount of random runs for each point, we can safely extrapolate the graph, giving the result that solving Bivium's state given 177 keystream bits takes about $2^{36.5}$ s.

To generate this result, Gaussian elimination was turned off, as it proved to slow down the solver if less than 58 reference state bits were guessed – for more than 58 guessed bits however, Gaussian elimination with cut-off depth 8 gave an average 5% speedup.

6 Conclusions

SAT solvers are a powerful tool in the analysis of mathematical assumptions, including cryptographic hardness and complexity assumptions. The full potential of SAT solving can only be achieved by matching the problem description to the solver language. For cryptographic ciphers, matching the solver and the problem requires extensive changes to the solver itself. We implemented several steps towards a specialized SAT solver for cryptography including native support for the XOR operation, Gaussian elimination, and logical circuit generation.

The extended solver solves problems from its target domain, simple and complex stream ciphers, faster than any other known SAT-solver based techniques.

The Crypto-1 cipher is solved in 40 seconds, HiTag2 in $2^{14.5}$ s, while Bivium takes $2^{36.5}$ s, 2^6 times less than the previous best SAT solver-based attack [6]. Stream ciphers can be strengthened against the attacks presented in this paper through the use of larger states, more complex feedback functions, and through longer initialization phases.

References

1. Bard, G.V.: Algorithms for the solution of polynomial and linear systems of equations over finite fields, with an application to the cryptanalysis of KeeLoq. Technical report, University of Maryland Dissertation (April 2008)
2. Garcia, F.D., et al.: Dismantling MIFARE Classic. In Jajodia, S., López, J., eds.: ESORICS. Volume 5283 of LNCS., Springer (2008) 97–114
3. Nohl, K.: Description of HiTag2. Webpage `http://cryptolib.com/ciphers/hitag2/`.
4. Raddum, H.: Cryptanalytic results on Trivium. Technical Report 2006/039, ECRYPT Stream Cipher Project (2006)
5. Cannière, C.D.: Trivium: A stream cipher construction inspired by block cipher design principles. In: ISC. Volume 4176 of LNCS., Springer (2006) 171–186
6. McDonald, C., Charnes, C., Pieprzyk, J.: Attacking Bivium with Minisat. Technical Report 2007/040, ECRYPT Stream Cipher Project (2007)
7. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. In: Proc. of Intl. Workshop on Bounded Model Checking. Volume 89 of ENTCS. (2003)
8. Davis, M., Putnam, H.: A computing procedure for quantification theory. J. ACM **7**(3) (1960) 201–215
9. Marques, J.P., Karem, S., Sakallah, A.: Conflict analysis in search algorithms for propositional satisfiability. In: Proc. of the IEEE Intl. Conf. on Tools with Artificial Intelligence. (1996)
10. Malik, S., Zhao, Y., Madigan, C.F., Zhang, L., Moskewicz, M.W.: Chaff: Engineering an efficient SAT solver. Design Automation Conference (2001) 530–535
11. Eén, N., Sörensson, N.: An extensible SAT-solver. In Giunchiglia, E., Tacchella, A., eds.: SAT. Volume 2919 of LNCS., Springer (2003) 502–518
12. Faugère, J.C.: A new efficient algorithm for computing Gröbner bases without reduction to zero (F5). In: ISSAC '02, ACM (2002) 75–83
13. Massacci, F., Marraro, L.: Logical cryptanalysis as a SAT-problem: Encoding and analysis. In Journal of Automated Reasoning **24** (2000) 165–203
14. Courtois, N.T.: Fast algebraic attacks on stream ciphers with linear feedback. In: CRYPTO 2003. Volume 2729/2003 of LNCS., Springer (2003) 176–194
15. Li, C.M.: Equivalency reasoning to solve a class of hard SAT problems. In: Information Processing Letters. (1999) 76–1
16. Sinz, C.: Visualizing SAT instances and runs of the DPLL algorithm. J. Autom. Reason. **39**(2) (2007) 219–243
17. Courtois, N.T., Nohl, K., O'Neil, S.: Algebraic attacks on the Crypto-1 stream cipher in Mifare Classic and Oyster cards. Technical Report 2008/166, Cryptology ePrint Archive (2008)