Enhanced Gaussian Elimination in DPLL-based SAT Solvers

Mate Soos*
UPMC LIP6, PLANETE team INRIA, SALSA team INRIA
Paris, France
soos_mate@gmail.com

Abstract

When cryptographical problems are treated in SAT solvers, they often contain large set of XOR constraints. Treating these XOR constraints through on-the-fly Gaussian elimination during solving has been shown to be a viable approach by Soos et al. We describe various enhancements to this scheme which increase the performance and mostly eliminate the need for manual tuning of parameters. With these enhancements, we were able achieve speedups of up to 29% on the Bivium and up to 45% on the Trivium ciphers, contrary to the 1-5% speedup achieved by the original scheme.

1 Introduction

SAT solvers have recently been enjoying a boom in the application front: more and more applications can and do make use of SAT solvers to accomplish tasks ranging from the fairly trivial to the very complex. In the particular use-case of cryptography, SAT solvers have become an important tool to analyse and break encryption mechanisms [3]. In the case of cryptographical application, SAT solvers are often faced with problems that encode relatively large amount of XOR constraints. The presence of these XOR constraints were exploited by Soos et al. [16] using on-the-fly Gaussian elimination, gaining a reported 1-5% speedup.

In this paper we extend CryptoMiniSat [14], a SAT solver based on MiniSat [5], with a much-improved Gaussian elimination, following the footsteps outlined in [16]. We tested the efficiency of the algorithm given four independent optimisations we added relative to [16] on stream cipher-based problems generated using the Grain-of-Salt tool [15]. The Bivium [12], Trivium [1] and HiTag2 [11] ciphers were used as benchmarks, and the improved Gaussian elimination routine speeded up solving Bivium by up to 29%, Trivium by up to 45% and HiTag2 by up to 5%, in contrast to the relatively minor speedups reported in the original paper.

During the SAT-Race of 2010 a newer version of CryptoMiniSat was running, v2.5.0. This version mainly differed from v2.4.2, described in this paper, by having less problems with clause-subsumption and having an improved binary clause reasoning engine. Even though CryptoMiniSat 2.5.0 was capable of using Gaussian Elimination, this feature was turned off for the duration of the Race.

Contributions

The rest of this paper is structured as follows. We give some background on DPLL-based SAT solvers, XOR constraint handling and Gaussian elimination in Sect. 2. We describe our method to remove columns and rows from the matrixes without affecting the power of the algorithm in Sect. 3. Then, we describe our improvements to the basic Gaussian elimination routine in the

*The author was supported by the RFID-AP Project of ANR, project number ANR-07-SESU-009

context of SAT solvers in Sect. 4. We describe our data structures in Sect. 5 and in Sect. 6 we present how to reduce the workload on the elimination routine by using multiple independent matrixes. In Sect. 7 we present the heuristic we used to turn on and off Gaussian elimination during search. Finally, in Sect. 8 we present our results, and in Sect. 9 we conclude this paper.

2 Background

In this section we give a short description of DPLL-based SAT solvers, describe how XOR constraints have been handled and preprocessed in SAT solvers, and describe how Gaussian elimination has been integrated into the DPLL procedure.

2.1 SAT solvers

Satisfiability solvers are complex mathematical algorithms used to decide whether a set of constraints have a solution or not. This paper only discusses the well-known conjunctive normal form (CNF) constraint type. The CNF formula φ on n binary variables $x_1, \ldots, x_n$, is a conjunction (`and`-ing) of m clauses $\omega_1, \ldots, \omega_m$ each of which is the disjunction (`or`-ing) of literals, where a literal is the occurrence of a variable e.g. x_1 or its complement, $\neg x_1$.

In this paper, we focus on solvers that use the DPLL algorithm. The DPLL procedure is a backtracking, depth-first search algorithm that tries to find a variable assignment that satisfies a system of clauses. The algorithm branches on a variable by assigning it to `true` or `false` and examining whether the value of other variables depend on this branching. If they do, the affected variables are assigned to the indicated value and the search continues until no more assignments can be made. During this period, called *propagation*, a clause may become unsatisfiable, as all of its literals have been assigned to `false`. If such a *conflict* is encountered, a *learnt clause* is generated that captures the wrong variable assignments leading to the conflict. The topmost branching allowed by the learnt clause is reversed and the algorithm starts again. The learnt clauses trim the search tree, reducing the overall time to finish the search. Eventually, either a satisfiable assignment is found or the search tree is exhausted without a solution being found and the problem is determined to be unsatisfiable.

2.2 XOR constraints in SAT solvers

The XOR constraint, which can be described as $\bigoplus_i x_i = \texttt{true/false}$ is quite common in multiple areas where SAT solvers are used. For instance, it is used extensively in software verification for bit-vector arithmetic and for binary function description in cryptography.

XOR constraints can be found with relative simplicity in any problem described in the CNF format. Heule [7, Sect. 6.2.1] describes an algorithm that sorts the clauses according to the variables they contain, and counts the number of negations for each specific variable set. If 2^{n-1} different negation combinations occur for a given variable set of size n, then the 2^{n-1} clauses are converted to a single XOR constraint. For example:

$$\left.\begin{aligned} x_1 \vee x_2 \vee \neg x_3 &= \texttt{true} \\ x_1 \vee \neg x_2 \vee x_3 &= \texttt{true} \\ \neg x_1 \vee x_2 \vee x_3 &= \texttt{true} \\ \neg x_1 \vee \neg x_2 \vee \neg x_3 &= \texttt{true} \end{aligned}\right\} \Leftrightarrow x_1 \oplus x_2 \oplus x_3 = \texttt{false}$$

Executing such an algorithm is relatively fast. From a practical point of view, using a computer with an Intel Core i7 processor, searching for XOR constraints takes on the order of a couple of seconds at most for the majority of application-domain problems of the 2009 SAT Competition.

Previous research [8, 16] has also extended DPLL-based SAT solvers to natively work with XOR constraints. The two-watch literal scheme of Chaff [10] has been extended to work on XOR constraints as a 2-variable watch scheme by Soos et al. in [16].

Preprocessing of XOR constraints has been, however, an active topic only in the context of solvers tuned to solve crafted DIMACS problems. Accordingly, the published literature on this topic have mostly improved the `march` family of solvers. The papers by Warner and van Maaren [17], Heule and van Maaren [8] and the thesis of Heule [7] present algorithms such as *local substitution*, *global substitution* and *dependent variable removal* to preprocess XOR constraints. The algorithms *local-* and *global substitution* shorten XOR constraints by selectively XOR-ing them together, and *dependent variable removal* removes XOR constraints in which variables appear that appear nowhere else.

2.3 Gaussian elimination in SAT solvers

Many application-domain problems contain sub-problems that are known to be very difficult to solve with the DPLL procedure, whilst being efficiently solvable with Gaussian elimination. Previous work focused on the DIMACS 32-bit parity problem set `par32`, one of ten difficult propositional reasoning challenges suggested by AT&T researchers in 1997 [13]. The work by Li [9] extracted 2- and 3-long XOR constraints from the parity problems and at each depth of the search tree, took two 3-long XOR constraints, XOR-ed them together to obtain binary XOR constraints during the search, which they used to replace variables with one another. This can be thought of as a simplified form of on-the-fly Gaussian elimination. A more successful approach was that by Chen [2], which extracted the 2- and 3-long XOR constraints, and in a preprocessing step, XOR-ed them together as long as they contained common variables. Once the longest possible XOR constraints have been found, they were subjected to Gaussian elimination, to extract as much information as possible. Finally, the remaining part of the problem was solved with the SAT solver WalkSAT.

Tightly integrating Gaussian elimination into DPLL-based SAT solvers was shown to be efficient in cryptographic scenarios by Soos et al. in [16]. The close integration of these radically different solving mechanisms was achieved by calling Gaussian elimination before every point the DPLL procedure branches on a variable. If the Gaussian elimination finds any truths (be them propagation(s) or a conflict), these truths are taken into account by the DPLL procedure, and the correct action is taken: either further propagations are carried out, or the conflict analysis routine is used to analyse the conflict returned by the Gaussian elimination procedure. In spite of this tight integration, the technique only lead to a 1-5% speedup on solving stream ciphers Bivium and Trivium.

Soos et al. [16] build the matrix on which to carry out Gaussian elimination by including every XOR chain in the problem as a row in the matrix. The matrix has exactly one more columns than the number of variables in all XOR-s: the last column is a special extra column called the *augmented column*, storing the invertedness of the XOR, i.e. if the XOR represented by the row should evaluate to `true` or `false`. The resulting matrix is duplicated into two independent, but closely tied matrixes. The first matrix is an assigned matrix, *A-matrix* and the second is a non-assigned matrix, *N-matrix*. The A-matrix is always updated with the current assignment of variables, and is kept upper-triangular, while the N-matrix is never updated with variable assignments, but follows the row ordering and row XOR-ing of the A-matrix. This is

A-matrix with x_3 assigned to `true`

x_1	x_2	x_3	x_4	x_5	aug
1	1	–	1	1	0
0	**1**	–	**0**	**0**	**0**
0	0	–	1	1	1
0	**0**	–	**0**	**1**	**0**
0	0	–	0	0	0

N-matrix

x_1	x_2	x_3	x_4	x_5	aug
1	1	0	1	1	0
0	**1**	**1**	**0**	**0**	**1**
0	0	0	1	1	1
0	**0**	**1**	**0**	**1**	**1**
0	0	1	0	0	1

Figure 1: An example A- and N-matrix pair. The A-matrix indicates propagation of $x_2 =$ `false` and $x_5 =$ `false`. The XOR constraint causing these propagations are in the N-matrix: $x_2 \oplus x_3 =$ `true` and $x_3 + x_5 =$ `true`, respectively. The A-matrix is kept upper triangular, while the N-matrix is kept in a state that any XOR constraint in it is a combination of the original problem's XOR constraints.

advantageous, since from the A-matrix any new assignments and/or conflicts can be read easily, while the same row in the N-matrix stores the combination of the original XOR-s that, when evaluated with the current assignments, lead to the propagation or conflict. An example setup is present in Fig. 1.

3 Row and column elimination by XOR

Warners and van Maaren [17] describe a technique to search and eliminate variables called *dependent* by the authors. Such variables are present in only one XOR constraint, and are not present in any clause. For instance, if variable x_1 is only present in the XOR constraint $x_1 \oplus x_2 \oplus x_3 =$ `true`, then this XOR constraint can be removed from the solver, along with the variable x_1. When the solving has finished, and the solution is SAT, the value of x_1 is calculated such as to satisfy the removed XOR constraint.

The removal of such dependent variables and their associated XOR constraints is a very useful preprocessing step for Gaussian elimination, as it removes a row and a column without lowering the deductive power of the procedure. We extended the technique of dependent variable removal to work on two XOR constraints. For example, if variable x_1 is not present in any constraints other than

$$x_1 \oplus x_{10} \oplus x_{11} = \texttt{true} \tag{1}$$
$$x_1 \oplus x_{20} \oplus x_{21} = \texttt{true} \tag{2}$$

then by XOR-ing these XOR constraints together, we obtain

$$(1) \oplus (2) = x_{10} \oplus x_{11} \oplus x_{20} \oplus x_{21} = \texttt{false}$$

which no longer contains x_1. The original constraints can then be removed, and the new XOR constraint inserted into the working set. Therefore the number of active XOR constraints is lowered by one, lowering the number of rows in the matrix used by the Gaussian elimination routine, and also removing a column (associated with the eliminated variable) from the matrix. When solving has finished and the solution is SAT, the value of the eliminated variable (x_1 in the example) is calculated using one of the original XOR constraints.

Even though this technique leads to an increase in the size of the resulting XOR constraints, this is not of concern. Firstly, modern processors pull in cache lines of 64 bytes at once, which should be sufficient to pull in the larger XOR constraints, as according to our experience, the original XOR constraints rarely contain more than 5 variables. Secondly, if the original XOR constraints were ever to cause a conflict or a propagation, the dependent variable had to be assigned and propagated: without it, both original XOR constraints would have an unassigned variable. By XOR-ing these two XOR constraints together, we can avoid this otherwise necessary assignment and propagation.

Allowing the final XOR constraint to be larger than any of its constituents differentiates this technique from *global substitution*, described in [7, Sect 6.2.3]. There, the author required the size of the resulting XOR constraint to be smaller than at least one of its constituents, which is not a requirement in our case.

The above detailed technique, which we call *Row and column elimination by XOR* (or RCX for short) is applied once as a preprocessor at the beginning of solving. The algorithm works in a similar fashion as the iterative variable elimination procedure of SatELite [4]. Variable occurrence list for all xor clauses is calculated, and an array is stored to indicate if a variable is present in any regular clause. Initially, all variables that don't appear in any regular clause are in a list `touched`. All variables in the touched list are checked whether their occurrence size is one or two. If it is one, the variable is dependent and is removed along with the xor clause. If the occurrence size is two, the XOR of the two xor clauses is added as a new xor clause, the original two xor clauses are removed, along with the variable. The removed xor clauses' variables are always touched if they don't appear in any regular clause. The algorithm finishes when the touched list is empty. An illustration of this algorithm is present in **Function** `Conglomerate`.

4 Reducing the size of the working set

There is extensive literature on Gaussian elimination, as it is one of the cornerstones of mathematics, and thus features as the base element of many complex algorithms. Although there is much to be learnt from these Gaussian elimination algorithms, they are unfortunately not directly applicable to SAT solving. There are mainly two reasons for this. Firstly, none of these algorithms are made to work on two matrixes simultaneously. However, if only one matrix (the A-matrix) is used, the reasons for the propagation of a literal or of a conflict will not be available to the solver, and thus conflict analysis will be seriously hindered. More importantly, these algorithms cannot deal with small changes (deltas) made to a matrix that has already been Gaussian eliminated. Exploiting these small deltas is an important way to speed up Gaussian elimination in the context of SAT solving, as we will see in this section.

By restricting Gaussian elimination to the part of the matrix where assignments have taken place, the complexity of the algorithm can be substantially lowered. Essentially, the Gaussian algorithm only has to update the matrix starting from the leftmost column that has been changed (i.e. its variable assigned) since the last time Gaussian elimination has been called, see Fig. 2(a). This means it is advantageous to order the columns in a way that the leftmost column is updated at the top of the search tree, and as the search progresses, the columns updated are progressively on the right of each other. Unfortunately, perfect ordering cannot be achieved, as it is very difficult to decide which variable will be propagated by which setting of variables. We used the approximate (and quite fragile) ordering given by the `order_heap` heap structure in CryptoMiniSat. If the variable activities don't change much during restart (which is typical of cryptographic instances, as the problem-type decider in CryptoMiniSat demonstrates), the variables with higher activity are most likely to be branched upon first, so we put them in the

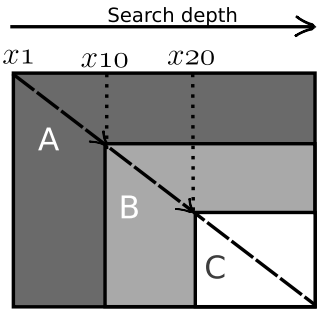
Function Conglomerate(φ, ψ, numVars) This function removes dependent variables as well as XOR-s two xor clauses that contain a common variable that appears only in those two xor clauses. The function removeXorClause not only removes and saves the xor clause for later calculation, it also updates the occurrence lists, and touches all variables in it for which inRegularClause[Var] is `false`. Similarly, addXorClause not only adds the clause, but also updates the occurrence lists, and touches all variables in it for which inRegularClause[Var] is `false`.

```
   Input: clauses φ, xor clauses ψ, variables numVars
 1 inRegularClause.resize (numVars, false);
 2 foreach var in φ do inRegularClause[var] ← true;
 3 populateOccurList (occur, ψ);
 4 touched.resize (numVars, false);
 5 for var ← 1 to numVars do
 6 |   if inRegularClause[var] = false then
 7 |   |   touched[var] ← true;
 8 |   |   touchedList.push (var);
 9 |   end
10 end
11 foreach var in touchedList do
12 |   if size(occur[var]) = 1 then
13 |   |   removeXorClause (occur[var][1]);
14 |   |   removeVar (var);
15 |   end
16 |   if size(occur[var]) = 2 then
17 |   |   addXorClause (occur[var][1] ⊕ occur[var][2]);
18 |   |   removeXorClause (occur[var][1]);
19 |   |   removeXorClause (occur[var][2]);
20 |   |   removeVar (var);
21 |   end
22 end
```

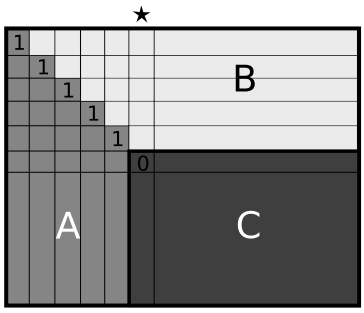
leftmost column, and the rest of the variables (resp. columns) were put progressively on the right of each other.

Another optimisation we applied relies on the observation that all rows above the row that contains the bottommost "1" in the leftmost updated column can also be mostly disregarded by the Gaussian elimination routine, see Fig. 2(b). None of these rows need to be XOR-ed with any other rows, since their leading "1" has been left unchanged, as this leading "1" is the bottommost "1" in that column. Since these rows can only change in that some of their (non-leading) "1"-s could have been zeroed out (due to assignments of columns), they can only cause propagations. To quickly calculate the range of rows that need to be checked for propagations, we keep an array that stores the row having the bottommost "1" in each column. This array is very cheap to keep updated during the Gaussian elimination procedure, and so poses almost no overhead, but saves significant time.

The combination of these two optimisations leads to a progressively smaller matrix as the search gets deeper in the search tree. The figures (a) and (b) in Fig. 2 illustrate this. This property is very advantageous as the majority of the time the DPLL algorithm is deep in the



(a) In the figure, variable x_1 was assigned first and so the entirety of the matrix, marked with "A", is treated. Later in the search, the leftmost assigned column was x_{10}, reducing the active part of the matrix to "B". Finally, very deep in the search tree, the leftmost assigned column was x_{20}, with the active part of the matrix shrunk to "C".



(b) In the figure, the variable (resp. column) marked with a star is the leftmost column that has been assigned. The part of the matrix marked with "A" is all zeros. The part of the matrix marked with "B" could have had some of its "1"-s zeroed, and needs to be checked if any of its rows contain all zeros — if so, that row only contains the leading "1", and causes a propagation. Gaussian elimination only needs to treat the part of the matrix marked with "C".

Figure 2: Illustration of the continual shrinking of the active part of the matrix as the search gets progressively deeper into the search tree.

search tree, and that is where the active part of the matrix is expected to be the smallest, and consequently the Gaussian elimination to be the fastest.

5 Using a dense matrix representation

A well-known optimisation for matrixes which are sparse, a quality that holds for our matrixes, is to store them in a special data structure to minimise overhead [6]. However, according to our experience, if Gaussian elimination is to bring any benefits, the matrixes must also be very small, otherwise using them does not bring an overall speedup. Since sparse matrix representation usually only brings benefits if the matrix is relatively large, we rejected it after some negative experiments. Using a sparse matrix representation would also necessitate the switching between a sparse and dense matrix representation every time a certain limit of density is reached, which is a non-trivial problem, as the N-matrix usually becomes dense much faster than the A-matrix (note that it is impossible to find a pivot that is optimal for both), and using two different data structures and algorithms for the two different matrixes poses problems in terms of programmability, instruction cache misses, etc.

For the reasons presented above, we store both A- and N-matrixes in a dense, bit-packed format except for the augmented column, which is stored in a non-packed format. There are three advantages of this data structure. Firstly, it becomes easy to check the augmented column's value, which is checked often. Secondly, XOR-ing rows can be done 32, 64 or even more bits at a time, depending on the available instruction set of the processor (32/64-bit, MMX, SSE, etc.). Thirdly, the bit-packed format means rows need less memory space to store, which is important for speed of storage and retrieval of matrixes. Fast storage and retrieval allows to store the matrix more often, thus saving time by avoiding to re-do eliminations already carried out at higher decision levels.

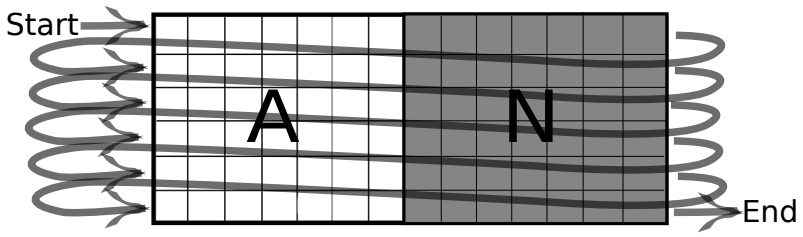


Figure 3: Illustration of the interlaced memory representation of A- and N-matrix's contents. Essentially, the two matrixes are represented in the memory as one bigger matrix, where the A-matrix is on the left, and the N-matrix is on the right.

5.1 Pairing the dense memory layout of the A- and N-matrixes

We tuned the memory layout of the two matrixes such as to minimise dataflow interruptions. We store A- and N-matrixes' rows in an interlaced fashion in a memory array, where the rows follow each other in the memory: $A[0], N[0], \dots A[n-1], N[n-1]$, where $A[0]$ is the first row of the A-matrix, $N[0]$ is the first row of the N-matrix, and n is the number of rows of both matrixes. A visual representation of this storage structure is present in Fig. 3.

This storage structure is advantageous, since when the two rows need to be swapped or XOR-ed, they need to be swapped/XOR-ed in both matrixes, and the memory swap/XOR operation can work on two $2m$-long memory areas instead of working on four m-long memory areas. For example, if two m-long rows x and y need to be swapped, they originally had to be swapped as

$$A[x][0] \dots A[x][m] \leftrightarrow A[y][0] \dots A[y][m]$$
$$N[x][0] \dots N[x][m] \leftrightarrow N[y][0] \dots N[y][m]$$

whereas now the memory areas

$$A[x][0] \dots A[x][m] \text{ and } N[x][0] \dots N[x][m]$$
$$A[y][0] \dots A[y][m] \text{ and } N[y][0] \dots N[y][m]$$

are next to each other, and can be read and written in one go. Since row swapping and XOR-ing accounts for 1/3rd of the total time spent in Gaussian elimination, this is an important optimisation.

6 Multiple A- and N-matrix sets

We do not necessarily put all XOR constraints into one A- and N-matrix pair. Instead, we build a graph from the XOR constraints where each variable is represented by a vertex and there is an edge between two vertexes when both variables are present in at least one XOR constraint. We then search for connected components in the resulting graph, and each connected component is treated as a separate A- and N-matrix pair.

This method does not reduce the algorithmic power of Gaussian elimination: if these matrixes were in one larger matrix, and the columns and rows of the matrix were suitably ordered such as to obtain a matrix that resembles the left-hand side of Fig. 4, then running Gaussian elimination on such a matrix would lead to exactly the same result as that on running Gaussian elimination on two separate matrixes, as illustrated on the right-hand side of the same figure.

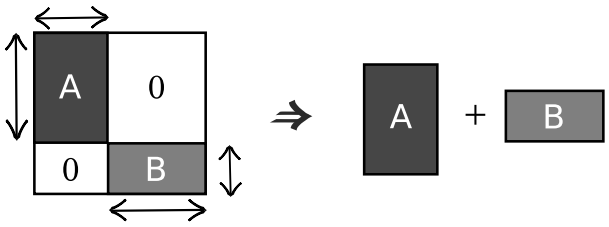


Figure 4: A matrix with two separate components. On the left-hand side, the columns (resp. variables) have been sorted such that variables that appear in the same component are preferably next to each other. Our implementation of the Gaussian elimination does not permutate columns, so it is easy to visually verify that the parts indicated with "0"-s would stay zero for every iteration of the Gaussian elimination. Therefore, if the two components are separated into different matrixes as it is present on the right-hand side, the output of the Gaussian elimination will not change. However, since the difficulty of Gaussian elimination is polynomial in the size of the matrix treated, it is preferable to isolate the sub-matrixes and treat them separately.

The advantage of having separate matrixes is the resulting reduced complexity of performing Gaussian elimination. The algorithmic complexity of performing Gaussian elimination on an $n \times m$ matrix is roughly $O(nm^2)$. If for instance two equally-sized separate matrixes are discovered and separated, the complexity is reduced from cnm^2 (where c is a suitable constant) to $2c(n/2)(m/2)^2 = cnm^2/4$, i.e. to the quarter of the original complexity. Most practical problems have many separate matrixes, representing different parts of the problem. For instance, stream ciphers typically have at least 2 independent shift registers (e.g. Grain), and all instances of the DIMACS 32-bit parity problem set contain two independent matrixes.

7 Auto turn-off heuristics

Though Gaussian elimination can be very powerful, it does not always perform very well for every restart of the solver. This is because some restarts explore a part of the search space that is not closely connected to the XOR constraints. In this case, Gaussian elimination cannot help much, and it is best to turn it off, such that it does not slow down the solver.

To achieve this turn-off, we implemented the following heuristics. Firstly, Gaussian elimination is used only if the problem is determined to be of cryptographic nature — this is automatic for CryptoMiniSat 2.4.2, and depends on the stability of variable activities between restarts, and the percentage of xor clauses of the instance. Secondly, for each restart (carried out by CryptoMiniSat in a geometrical sequence in case of cryptographic problems), the number of propagations and conflicts caused by the Gaussian elimination is measured during the first 100 conflicts. We then calculate the heuristic cut-off $\text{CUTOFF} = 2\text{NUMGAUSSCONFL} + \text{NUMGAUSSPROP} - 0.05\text{NUMGAUSSCALLED}$ where NUMGAUSSCALLED is the number of times Gaussian elim. was called, NUMGAUSSCONFL is the number of times Gaussian elim. caused a conflict, and NUMGAUSSPROP is the number of times Gaussian elim. caused a propagation. If $\text{CUTOFF} < 0$, we decide that it is best to turn off Gaussian elimination for the duration of the restart, otherwise, we carry on with Gaussian elimination during the whole restart. The heuristic cut-off was derived experimentally, though since we did not have the necessary computing resources to experiment with all combinations, it remains an educated guess.

Table 1: Time to solve the non-compressed parity learning problem set `par32` by EqSatz, XORSAT, and CryptoMiniSat with Gaussian elimination. The auto turn-off had to be deactivated for these problems, as it prematurely turned off Gaussian elimination. All times are scaled to a Pentium 4@2.66MHz computer speed.

Instance	#var	#clause	EqSatz	XORSAT	CryptoMS with Gauss	Matrixes found
`par32-1`	3176	10277	181.5 s	0.094 s	50.2 s	$615 \times 647, 61 \times 123$
`par32-2`	3176	10253	44.0 s	0.422 s	2.6 s	$593 \times 625, 61 \times 123$
`par32-3`	3176	10279	2062.9 s	3.235 s	18.1 s	$614 \times 646, 61 \times 123$
`par32-4`	3176	10313	170.6 s	0.219 s	9.9 s	$615 \times 647, 61 \times 123$
`par32-5`	3176	10325	2844.6 s	2.922 s	10.0 s	$628 \times 660, 61 \times 123$

8 Results

To test the effectiveness of our improved Gaussian elimination routine, including all the new enhancements detailed in Sections 3–7, we used CryptoMiniSat 2.4.2 [14] and problems generated by the Grain-of-Salt tool [15] for ciphers Bivium, HiTag2, Trivium and Grain. The timing results for all ciphers with Gaussian elimination enabled and disabled are are present in Table 2. The table shows that Gaussian elimination lead to a significant, up to 45% speedup for Trivium, up to 29% speedup for Bivium, and up to 5.5% speedup for HiTag2. The solving of Grain was slowed down by by up to 23% through the usage of Gaussian elimination, but it seemed to perform better as the problem difficulty increased.

During the many thousands of problems solved to generate Table 2, Gaussian elimination only provided a handful of unitary xor clauses. From this we can conclude that essentially all speedups were achieved through the on-the-fly nature of the algorithm, rather than its possible use as a preprocessor to find unitary xor clauses.

We have also tested how RCX and Gaussian elimination performed in all on/off combinations with the Bivium cipher. The results are present in Table 3. The slowest solving was without RCX or Gauss, the next fastest was RCX without Gauss, then Gauss without RCX, and finally, the fastest was RCX and Gauss combined. RCX alone helped around 5-15%, and in combination with Gauss, it helped an extra 3-8%.

For comparison, we also tested our Gaussian elimination routine on the non-compressed parity learning problem set `par32`[1] that EqSatz [9] and XORSAT [2] aimed to solve. The solving times for our and these solvers are in Table 1. Our method performed better than EqSatz, but XORSAT left it behind in speed. This difference can be attributed to the fact that XORSAT used WalkSAT to solve the non-XOR part of the problem, which was better adapted to this task than the DPLL solving backbone used by CryptoMiniSat.

In the original article by Soos et al. [16], stopping the Gaussian elimination at a precise search depth was deemed necessary, but in our case, we could mostly do away with this burden. We set the maximum depth to 100 in the case of Bivium and Trivium ciphers, and 30 in case of HiTag2 and Grain ciphers. Both of these values were very crude approximations, and we believe that a more fine-tuned value would probably have lead to better results.

On problems that didn't benefit from Gaussian elimination, we have found that the slowdown caused by Gaussian elimination is problem-dependent. If while solving Gaussian elimination

[1] Problems that identify a 32-bit parity function given (potentially noisy) I/O samples of the function. Generator by Rob Schapire and Haym Hirsh

Table 2: Table showing the average time (in seconds) for solving different stream ciphers, each ran 100 times with random key and IV, given a number of randomly selected state bits (as help bits) each set randomly to `true` or `false`.

Bivium												
no. help bits	56	55	54	53	52	51	50	49	48	47	46	45
RCX	0.35	0.65	0.89	1.30	2.36	5.76	8.87	14.75	35.68	79.83	104.90	193.98
Gauss+RCX	0.31	0.52	0.69	0.90	1.85	3.81	6.20	9.55	20.86	35.25	75.68	137.44

Trivium					
no. help bits	157	156	155	154	153
RCX	66.57	86.42	146.17	261.75	472.27
Gauss+RCX	40.57	68.16	84.13	146.35	259.07

HiTag2							
no. help bits	15	14	13	12	11	10	9
RCX	4.78	11.73	30.70	76.44	233.61	719.86	1666.99
Gauss+RCX	4.76	11.64	29.03	77.19	220.64	701.46	1636.77

Grain				
no. help bits	109	108	107	106
RCX	168.51	291.29	540.14	1123.08
Gauss+RCX	193.09	359.58	608.47	1133.75

Table 3: Time to solve (in seconds) for the state of different random Bivium problem instances (random IV and key), given randomly set state bits (help bits), similarly to what is present in Table 2. This table shows different combinations of the RCX preprocessing algorithms and the on-the-fly Gaussian elimination turned on and off.

Bivium							
no. help bits	56	55	54	53	52	51	50
no RCX + no Gauss	0.40	0.69	1.26	1.38	2.19	6.25	10.40
RCX + no Gauss	0.35	0.65	0.89	1.30	2.36	5.76	8.87
no RCX + Gauss	0.34	0.55	0.91	1.06	1.89	3.87	7.76
RCX + Gauss	0.31	0.52	0.69	0.90	1.85	3.81	6.20
Num vars removed on avg. by **Function** `Conglomerate`	36.42	36.27	36.42	37.30	37.07	38.32	37.94

regularly finds new truths, but the matrix is too large, the slowdown can be on the order of 20% — an example for this is the Grain cipher. However, if the matrix is small, the auto turn-off heuristic acts quickly enough, and the slowdown is on the order of 1%.

9 Conclusions

We have described a multitude of ways to improve on-the-fly Gaussian elimination in the context of SAT solvers, originally put forward by Soos et al. [16]. Without these advancements, the speed increase of Gaussian elimination was only 1-5%, but with our enhancements, the speedups were up to 45% for the Trivium, up to 29% for the Bivium, and up to 5% for the HiTag2 ciphers.

We believe that in the long term, a successful mix of SAT solvers and higher-level algorithms such as Gaussian elimination could potentially be the best way to solve complex challenges arising from a multitude of different problem domains. Also, future research on the effect of Gaussian elimination could further enhance the heuristics controlling its activation, including auto turn-off heuristics and cut-off depth, possibly leading to even more speedups.

Acknowledgements

Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (`https://www.grid5000.fr`).

References

[1] Cannière, C. D. Trivium: A stream cipher construction inspired by block cipher design principles. In *ISC* (2006), S. K. Katsikas and et al, Eds., vol. 4176 of *LNCS*, Springer, pp. 171–186.

[2] Chen, J.-C. XORSAT: An efficient algorithm for the DIMACS 32-bit parity problem. *CoRR abs/cs/0703006* (2007).

[3] Courtois, N. T., and Bard, G. V. Algebraic cryptanalysis of the Data Encryption Standard. Tech. Rep. 2006/402, IACR E-print, November 2006.

[4] Eén, N., and Biere, A. Effective preprocessing in SAT through variable and clause elimination. In *SAT* (2005), F. Bacchus and T. Walsh, Eds., vol. 3569 of *Lecture Notes in Computer Science*, Springer, pp. 61–75.

[5] Eén, N., and Sörensson, N. An extensible SAT-solver. In *SAT* (2003), E. Giunchiglia and A. Tacchella, Eds., vol. 2919 of *LNCS*, Springer, pp. 502–518.

[6] Gilbert, J. R., Moler, C., and Schreiber, R. Sparse matrices in MATLAB: Design and implementation. *SIAM J. Matrix Anal. Appl 13* (1991), 333–356.

[7] Heule, M. march: Towards a lookahead sat solver for general purposes. Tech. rep., Technische Universiteit Delft, February 2004.

[8] Heule, M., and van Maaren, H. Aligning CNF- and equivalence-reasoning. In *SAT (Selected Papers* (2004), H. H. Hoos and D. G. Mitchell, Eds., vol. 3542 of *Lecture Notes in Computer Science*, Springer, pp. 145–156.

[9] Li, C. M. Equivalent literal propagation in the DLL procedure. *Discrete Applied Mathematics 130*, 2 (2003), 251–276.

[10] Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., and Malik, S. Chaff: Engineering an efficient SAT solver. In *DAC* (2001), ACM, pp. 530–535.

[11] Nohl, K. Description of HiTag2. Press release, March 2008. `http://cryptolib.com/ciphers/hitag2/`.

[12] Raddum, H. Cryptanalytic results on Trivium. Tech. Rep. 2006/039, ECRYPT Stream Cipher Project, 2006.

[13] Selman, B., Kautz, H., and McAllester, D. Ten challenges in propositional reasoning and search. In *IJCAI'97: Proceedings of the 15th international joint conference on Artifical intelligence* (San Francisco, CA, USA, 1997), Morgan Kaufmann Publishers Inc., pp. 50–54.

[14] Soos, M. CryptoMiniSat — a SAT solver for cryptographic problems, 2009. `http://planete.inrialpes.fr/~soos/CryptoMiniSat2/`.

[15] Soos, M. Grain of Salt — an automatic system to generate cnfs from stream cipher descriptions, 2009. `http://planete.inrialpes.fr/~soos/GrainOfSalt/`.

[16] Soos, M., Nohl, K., and Castelluccia, C. Extending SAT solvers to cryptographic problems. In *SAT* (2009), O. Kullmann, Ed., vol. 5584 of *Lecture Notes in Computer Science*, Springer, pp. 244–257.

[17] Warners, J. P., and van Maaren, H. A two-phase algorithm for solving a class of hard satisfiability problems. *Oper. Res. Lett. 23*, 3-5 (1998), 81–88.