Attacking Bivium Using SAT Solvers

Tobias Eibach, Enrico Pilz, and Gunnar Völkel

University of Ulm, Institute of Theoretical Computer Science,
James-Franck-Ring 27, 89069 Ulm, Germany
{tobias.eibach,enrico.pilz,gunnar.voelkel}@uni-ulm.de

Abstract. In this paper we present experimental results of an application of SAT solvers in current cryptography. Trivium is a very promising stream cipher candidate in the final phase of the eSTREAM project. We use the fastest industrial SAT solvers to attack a reduced version of Trivium – called Bivium. Our experimental attack time using the SAT solver is the best attack time that we are aware of, it is faster than the following attacks: exhaustive search, a BDD based attack, a graph theoretic approach and an attack based on Gröbner bases. The attack recovers the internal state of the cipher by first setting up an equation system describing the internal state, then transforming it into CNF and then solving it. When one implements this attack, several questions have to be answered and several parameters have to be optimised.

Key words: SAT Solver, Application, Cryptography, Stream Cipher, Rsat, eSTREAM, Bivium, Trivium, BDD, Gröbner Base

1 Introduction

Stream ciphers are used in many applications like GSM, UMTS, RFID, Bluetooth and online encryption of big amounts of data in general. The eSTREAM project ([1]) was started in October 2004 to find a new stream cipher, after the NESSIE project ([2]) ended in 2003 without recommending one. Starting with a call for candidates the project is organised into several phases and in each phase weak candidates dropped out. The final stream cipher candidates should be fast and cryptographically secure. All eSTREAM candidates are divided into two categories: hardware-oriented and software-oriented ciphers. The eSTREAM project is now in the last phase with only few candidates left in each category. One of the hardware-oriented ciphers is Trivium, introduced in [3]. Until now no attacks have been successfully applied to Trivium – i.e. it has not been possible to prove a running time faster than exhaustive search. In [4] a reduced version of the cipher has been introduced: Bivium (initially called Bivium B). The intention is to find attacks on Bivium and then extend them to Trivium. In this paper we focus on Bivium, however this "algebraic attack" concept is generic and can also be applied to other stream ciphers.

Stream ciphers are symmetric cryptographic primitives – the communicating parties already share a secret key. Like most stream ciphers, Trivium can be described as a finite automaton whose initial state is derived from the secret key and a public known initialisation vector (IV) by filling the register with the key and the IV and then making a few transitions to "shuffle" the internal state. After this initialisation phase the cipher starts to produce output bits (the "keystream"). Trivium and Bivium produce one keystream bit with every clock (step) of the cipher. To encrypt a message m, one uses the XOR function to add the message- and keystream-bits bitwise, e.g: $m_i \oplus z_i = c_i$ for $i = 1, 2, 3, ...$ The receiver produces the same keystream z and also adds it bitwise to decrypt the message.

We use the common and realistic attack scenario, that we know a part of the keystream z (from a known-plaintext attack) and try to reconstruct the internal state of the cipher from it. If we are successful, we can clock the cipher backwards to reconstruct the secret key. More importantly, we can clock the cipher forward to produce the whole keystream and thereby decrypt the whole message.

In order to use a SAT solver to reconstruct the internal state, we first set up an equation system, given by the Bivium definition and the observed keystream. The solution of the equation system is the internal state of the cipher. Then we transform the equation system into a CNF formula, by using the truth table and Quine McCluskey algorithm. Finally we use the fastest complete SAT solvers of the SAT competition 2007 ([5]) in the industrial category to solve the CNF. Doing this, there are several parameters to be optimised and several questions that have to be answered.

We are aware of several attack concepts that can be applied to Bivium as well. We implemented 3 of them and we quote the results of the remaining ones to have a complete comparison of the SAT solver speed to the other attack speeds. According to our experiments, the SAT solver attack is by far the fastest attack type. It is faster than our exhaustive search on the key/IV-setup, also faster than our attack based on BDDs (binary decision diagrams) and faster than our attack based on Gröbner bases. It is also faster than the attack times that we found in other papers: an attack based on a graph-theoretic approach ([4]), a guess-and-determine strategy ([6]) and an attack based on the birthday paradoxon ([7]).

Most attack concepts like the one based on SAT solvers, BDDs or Gröbner bases use heuristical algorithms that are theoretically not well understood – at least there is a significant gap between the proven bounds on the running times and the actual running times. This is the reason why experiments are needed. We run our experiments on a multi-processor system. The important process is computed on a fast 2 GHz CPU with 2GB of memory to avoid interference with other processes. We address several questions and optimisations and try to answer them isolatedly.

1.1 About this Paper

In Section 2 we describe the stream cipher Bivium. In Section 3 we describe, how we use SAT solvers to recover the internal state of Bivium. In Section 4 we show, how we optimised the attack and which results we found. In Section 5 we describe briefly our attack on Bivium using BDDs. In Section 6 we briefly describe our attack using Gröbner bases. Finally in Section 7 we discuss and compare our experimental results, we also compare them to results that have been published so far and give an outlook in Section 8.

2 Description of Bivium

We focus our attack on Bivium, a stream cipher with an internal state of 177 bits (that can be seen as 2 registers, of size 93 and 84 bits) and a key size of 80 bits. The internal state of Bivium is initialized with the secret key, the initialization vector (IV) and zeros. Then the cipher is clocked $4 \cdot 177$ times and then starts producing keystream bits, according to the scheme given in Figure 1. As one can see from the figure, Bivium is obviously a reduced version of Trivium, as it uses only 2 registers instead of 3. The update-functions of the 3 internal registers are non-linear, as each involves one AND gate. The output-function is linear as it just combines 4 (for Bivium) or 6 (for Trivium) bits of the internal state by a XOR gate. One also notices the very low amount of gates used. On the one hand this leads to a low power consumption of the cipher and a fast implementation, but on the other hand the equation system describing the cipher will be sparse and thereby can be rather efficiently converted to a CNF (conjunctive normal form) formula (see also [8]).

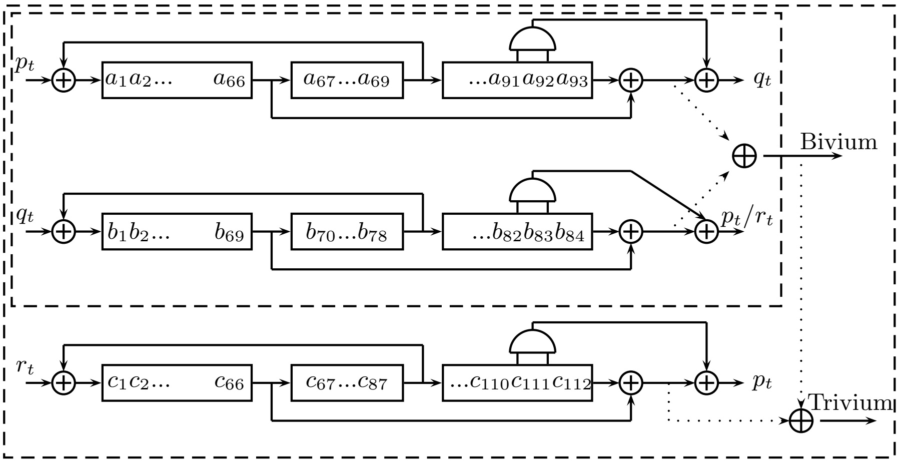

Fig. 1. Trivium and Bivium scheme

The size of the secret key (used to initialise the cipher) is just 80 bits. However, so far the most efficient way to attack the key directly is exhaustive search (see Section 7). For the SAT solver attack, we decided not to attack the key directly. We try to reconstruct the internal state (177 bit) from a part of the keystream. Of course it is a disadvantage to have a search space of 177 bits instead of the 80 bits. However the equation system in the 80 key-bits gets far too difficult through the key/IV-setup phase. Below we give the pseudocode of Bivium. For a more detailed description please see [4] for the description of Bivium or [3] for Trivium.

Bivium pseudocode

```
1: for i = 1,2,3,... do
2:   t[1] := s[66] + s[93]
3:   t[2] := s[162] + s[177]
4:   z[i] := t[1] + t[2]
5:   t[1] := t[1] + s[91] * s[92] + s[171]
6:   t[2] := t[2] + s[175] * s[176] + s[69]
7:   (s[1],s[2],...,s[93]) := (t[2],s[1],...,s[92])
8:   (s[94],s[95],...,s[177]) := (t[1],s[94],...,s[176])
9: end do
```

$s[1]...s[177]$ denote the internal state of the cipher and $z[i]$ ($i = 1, 2, 3, ...$) denotes the output of the cipher. $t[1]$ and $t[2]$ are temporary variables.

3 Describing the Attack

The concept of using SAT solvers for attacking stream ciphers has been proposed in [8] and [9]. Before this, there have already been other remarkable applications of SAT solvers in cryptography. The actual running time of a SAT solver can be hardly estimated. Here experiments are needed to determine the running time on Bivium instances.

First we generate an equation system describing the internal state of the cipher. By clocking the cipher we get one new equation in every step from line 4 of the pseudo code that connects the (known) output to the internal state. In line 5 and 6 we can decide whether we want to introduce 2 new variables for t[1] and t[2] or whether we use the given construction of t[1] and t[2] in line 7 and 8. The first option obviously increases the number of variables and equations but keeps the equations short and the degree low while the second increases the degree and length of the equations. It turned out that introducing two new variables for t[1] and t[2] has many advantages: The equation system produced has only 2 types of equations – one with 4 and one with 5 variables, which can be converted into CNF without producing too many clauses. Also we believe that this way the “structure” of the initial problem can be maintained for the SAT solver while algebraic operations on the equation system might reduce the link between the original structure and the structure in the CNF formula.

Having the equation system we now transform every single equation into a CNF formula that equals 1 in case the equation is fulfilled and 0 otherwise. By combining all formulas with AND we get an equivalent Boolean formula.

Transforming an algebraic equation into CNF can be done by looking at the truth table to construct the clauses and then the formula can be minimized by using the Quine McCluskey algorithm. Transforming an equation with n variables produces about 2^{n-1} clauses. Actually 2^{n-1} is the maximum, but the actual number is close to this, as the equation system consists mainly of XOR operations and has only few AND operations. The fact that the number of clauses grows exponentially in n means that at a certain point we have to introduce new variables that substitute several old variables, to keep the number of variables in each equation small. In [8] a "cutting number" of 6 variables per equation is suggested. This means that if an equation has more than 6 variables one should substitute half of them by a new variable, leading to 2 new equations with about half the size and fewer clauses in the final CNF. By introducing 2 new variables in every step we do not have to consider this cutting number rule, as our equations do not get bigger than 5 variables.

Having the Boolean formula we have to transform it into the DIMACS format that serves as input to the SAT solver program and is just a convention ([10]). Before we let the SAT solver solve the formula we have to reduce the complexity of the instance by guessing some variables. Here we have to decide how many variables we guess and which ones. If we decide to guess m variables the expected number of runs of the SAT solver will be approximately 2^{m-1} so the total expected running time will be 2^{m-1} times the average running time for one instance (if we guess the variables in a way so that we do not guess the same assignment twice).

We skipped one more consideration above: We have to decide how many equations we want to produce for our equation system. We need at least 177 equations – but more equations would make the problem instance overdefined and could thereby speedup the solving time. Our experiments showed that less than 180 used keystream bits are not enough (the attack was very slow or did not return the internal state that we were looking for). We did not see a difference in the range of 180 to 300 used keystream bits, just a slight increase in the running times above 250 bits used. So we decided to use 200 keystream bits to set up our equation system. Also as noted in [4] it is not necessary to add new variables to the equation system, if they do not get "connected" to the keystream (the last introduced 66 variables for the first register and the last introduced 69 variables for the second register do not get connected to the keystream). This way the number of variables can be reduced.

4 Experimental Results of the SAT Attack

In this section we present the experimental results of our implementation of the attack using SAT solvers. All times are given in seconds and are averaged over 100 instances. In Table 1 we compare several SAT solvers to find out which one would solve our kind of instances fastest. In the second column we guess 40 values of the internal state, in the next column 45 and then 50. We used the guessing strategy "Ending2" (see below). The fastest SAT solver is Rsat combined with the SatElite preprocessor (version 2.01), followed by MiniSAT (version 2/070721) – available at [11] and [12].

Table 1. Comparing SAT solvers

	guess 40	guess 45	guess 50
Rsat & SatElite	46.10	3.32	0.26
MiniSat	67.32	5.06	0.36
Picosat	103.96	5.78	0.42
Rsat	229.09	11.49	0.79
Zchaff	735.08	17.36	0.78

Rsat (with SatElite) and MiniSat were also the two fastest solvers in the SAT competition 2007 in the UNSAT industrial category (as we are guessing m bits to reduce the complexity, the outcome is “UNSAT” in all runs except one). Consequently the following experiments have been done using Rsat (with SatElite). In Table 2 we compare several guessing strategies. Inspired by [6] we used the strategy “ThreeFour” that is guessing 3 variables in a way to directly compute a fourth variable. This way it is possible to start the SAT solver with 64 variables guessed – at the same cost as guessing 48 independent variables. However it turned out that guessing the last 48 variables of the second register (“Ending2”) helps more and leads to a faster average running time. We also tried “Ending1” that is guessing the end of the first register and “Ending-halved” that is guessing the endings of both registers. We did the same for the beginning positions of the registers. We also tried to guess 3 random sets of variables that show quite a variation. One possible explanation why the “Ending2” strategy gives the fastest running time might be, that many of the most frequent occurring variables are in the “Ending2” set.

Table 2. Comparing different guessing strategies

strategy	time
Beginning1	3246
Beginning2	21.27
Beginning-halved	2712
Ending1	3.94
Ending2	0.116
Ending-halved	0.718
ThreeFour	0.275
Random1	1.144
Random2	51.988
Random3	17.993

In Figure 2 we determine the optimal number of variables to guess. The x-axis shows the number of variables guessed using the “Ending2” strategy. The y-axis shows the expected running time (scaled by 10^{-10}) of the whole attack. The dark curve shows the case that we guess randomly (e.g. with high probability incorrect, leading to an UNSAT result of the SAT solver) and the dotted curve shows the running time if we guess the variables correctly. This however will happen just once in our attack scenario. One should expect SAT solvers to be better than guessing a variable randomly at the cost of multiplying the running time with 2. This is – at a certain point – not the case, so that we measure a minimum at guessing 45 variables with an expected running time of 1637E10 seconds.

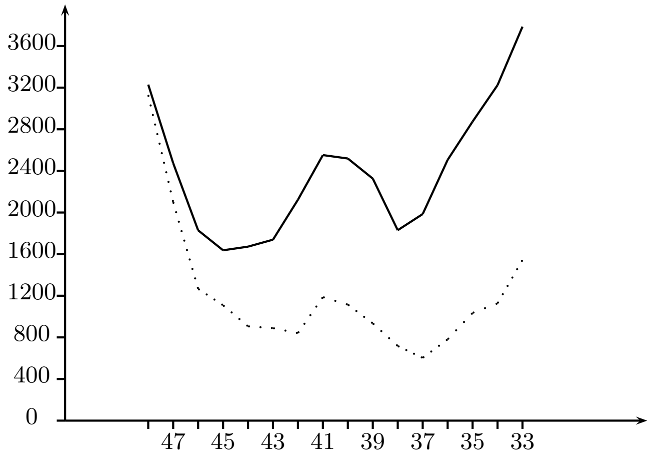

Fig. 2. Optimal guessing number

In Figure 3 we determine the correlation between the Hamming-weight of the internal state and the time needed to solve the CNF. When we guess the 36 variables correctly there is a huge correlation (dotted curve). This however happens only once in the attack scenario and we do not see this correlation if we guess randomly (dark curve). The x-axis shows the Hamming weight of the internal state and the y-axis the time needed to solve one instance by guessing 36 variables with the “Ending2” strategy. We average the running time here over just 50 instances.

Following the idea of the attack described in [6], we made one experiment, whether certain parts of the keystream are easier to attack. This however was not the case. In all the experiments, we observed a huge variation in the running times – up to a factor of 20. This is why we averaged the running times over 100 instances.

One way to optimise the attack is to guess the m variables earlier than in the final CNF formula. This allows us to do some further simplifications in the equations. However this work has to be done for every guess and not just once. We did not see a big influence on the running times here, but it allows

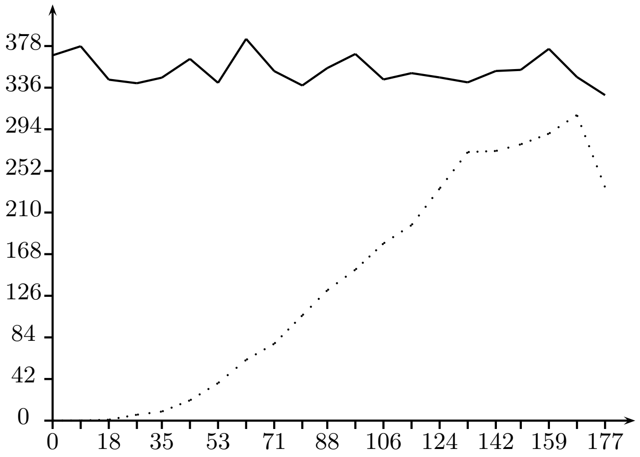

Fig. 3. Influence of the Hamming weight

us to give a more realistic table on the CNF statistics. So in Table 3 we give the averaged numbers of the instance size that we get if we guess m bits in the equation system and do some fast simplifications there (roughly: expand equations, remove duplicates and remove duplicate monomials, where possible and substitute to 4-CNF).

Table 3. Number of guessed variables vs. number of clauses and variables

Nb. guessed	Nb. clauses	Nb. variables
0	9909.0	2677.00
31	5631.0	1543.50
35	5200.2	1427.70
39	4933.8	1361.50
43	4679.4	1293.50
45	**4560.4**	**1265.10**
47	4463.6	1236.80
51	4277.4	1189.45

In [8] it is further suggested to use Gaussian elimination to reduce the number of variables in each equation. This did not help in our experiments.

5 An Attack Based on BDDs

We also implemented an attack on Bivium that is based on BDDs. We explain the idea of this attack roughly, present our attack times and in Section 7 we compare them to the SAT solver attack times.

A BDD is a way to represent a Boolean formula as a directed graph, with the variables at the nodes, the values of the variables at the outgoing edges and the corresponding function values at the leaves. There are several fast operations on BDDs, especially if we delimit them to OBDDs. These are BDDs with an order on the variables, so every variable can only be read once and in every path the reading order is the same. In this case, the following operations are efficient: count the remaining number of paths leading to 1 ("sat-count"), minimize the BDD for the given order and combine two BDDs. For an introduction please see [13].

The idea of the BDD attack has been published in [14], more details are published in [15] and improvements in [16] and [17]. The BDD attack uses one BDD to efficiently represent all possible internal states of Bivium at a given time. The main BDD that we construct in step i represents the characteristical function of the set of all possible internal states after i observed keystream bits. To construct the main BDD we describe the 2 internal update-functions and the output-function of Bivium as 3 BDDs for every keystream bit. We combine all those BDDs to get the main BDD. We initialize the main BDD with the constant 1-BDD that represents the fact that all internal states are possible (as we have not read a single keystream bit). Then, for every keystream bit, we combine the 3 new BDDs with the main BDD using the AND operation. This reduces in every step the number of possible internal states. After the minimal amount of keystream bits the BDD represents only the one internal state that we wanted to recover.

Also in this attack there are several questions to be answered and parameters to be optimised. First we had to decide which BDD library to use. We only achieved a fast implementation using the CUDD library ([18]). CUDD uses a hash-table of initially fixed size to speed up the operations on the BDDs. The size of this hash-table is critical. In our experiments we got good running times with about 1GB of memory, less memory led to swapping and thereby to much worse running-times. The running time of the BDD operations depends mainly on the size of the BDD (number of nodes) and the width of the BDD (the size of the maximal level of the BDD). There is no particular order in which the single BDDs have to be combined. One can start by combining only the BDDs describing the output-function and then at a later point add the BDDs of the update-function. The optimisation target here is to keep the main BDD as small as possible.

The CUDD library does not support an explicit minimisation operation, but the resulting BDD of an AND operation is already minimised (for a fixed variable order). The reordering operation tries to further minimize the BDD by reordering the variables. This operation requires most of the time in the BDD attack and is performed every few steps (based on a heuristic). The reordering operation is also just a heuristic and does usually not find the best ordering of the variables. However it significantly reduces the size of the BDD (up to a factor of 10) and makes future operations faster.

5.1 Experimental Results of the BDD Attack

The variance in the running times is much lower for BDDs than for SAT solvers. So we averaged our experimental results over just 10 runs for every instance. After identifying the CUDD library as the best BDD library for our purpose we continued the same way as for the SAT solver attack. We tried several guessing strategies, to find out how the complexity of the problem can be reduced most efficiently. Again it was most useful to guess the bits close to the "end" of Bivium (the bits close to the output). With a small difference: it is best to guess half of the bits at the end of the first register and half of the bits at the end of the second register ("Ending-halved").

In Figure 4 we determine the optimal number of variables to guess in the BDD attack. The x-axis shows the number of variables that we guess (using the "Ending-halved" strategy). The y-axis shows the expected running time in seconds scaled by 10^{-17}. It is optimal to guess 55 variables, with an expected running-time of 4.22E17 seconds. As for the SAT attack there is a point from which on it is better to guess more variables at a cost of "factor 2" in the running time, than using the BDD construction.

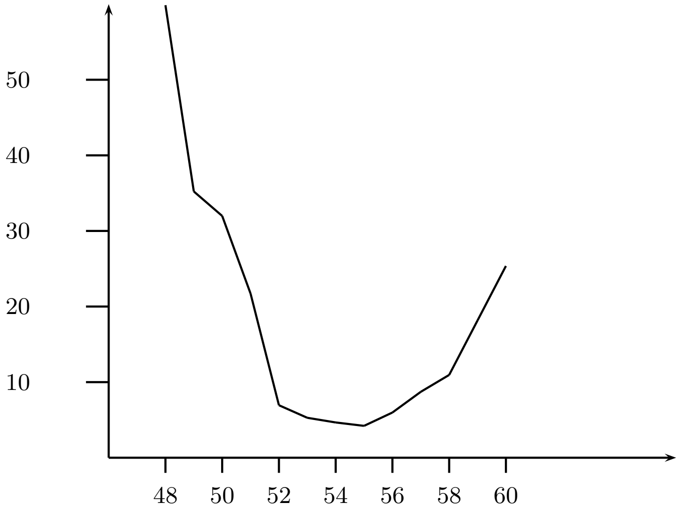

Fig. 4. Optimal guessing number for the BDD attack

6 An Attack Based on Gröbner Bases

Gröbner bases are the most common and usually fastest way to solve a system of (non-linear) equations. They are used in algebra software like Mathematica, Maple, MAGMA and Singular ([20]). We use the algebra software SAGE ([19]) that integrates the Singular software, as it is free, offers several algorithms to compute a Gröbner base and also Singular is well known for its fast and optimised Gröbner base algorithms.

A Gröbner base G for a given set of polynomials F by definition generates the same ideal as F and has certain nice properties that allow fast solutions to many problems that are hard for F. This is especially true for the solution of the equation system: the Gröbner base G has the same solution(s) as F but due to the "elimination property" the solution(s) can be derived easily. The "elimination property" implies that one of the polynomials is univariate – meaning that it depends on just one of the variables – and so this variable can be computed directly. When we substitute this variable in the other equations we get again an equation in just one variable and so on. This can be seen as a generalised form of the Gaussian elimination (please see [21] for an introduction).

First we set up the Bivium equation system – in contrast to the SAT solver attack, it is much better not to introduce any new variables, so we set up the equation system in just the 177 unknown internal bits. We set up 200 equations and then we guess several variables to reduce the complexity of one instance. We use different algorithms of the Singular packet to construct the Gröbner base, this step took most of the time. If we guess the variables incorrectly (most likely) the Gröbner base will be just the 1-polynomial and we guess again. If we guess correctly, we will be able to derive the solution (the internal state) very efficiently. In contrast to the SAT solver approach here the running time when guessing incorrectly is much faster than when guessing correctly.

Table 4. Comparing guessing numbers and Gröbner algorithms

Nb. guessed	std	slimgb
64	0.2852	0.1887
62	0.2308	0.2264
60	0.1824	**0.1823**
58	8.1856	82.019
56	267.35	1114.3

In Table 4 we give the averaged times for the 2 fastest Gröbner base algorithms in the Singular packet (out of 5 algorithms) for solving one instance. We used the "Ending-halved" guessing strategy for which it is optimal to guess 60 variables (randomly), resulting in an expected running time of 1.051E17 seconds. The "std" algorithm used from 100MB to 5 GB of memory – depending on the number of variables guessed. The "slimgb" algorithm used just up to 400 MB.

There are many different algorithms to compute a Gröbner base for a given equation system and a given ordering on the monomials. Two well known algorithms are the F4 and F5 algorithm by Faugère. In the Singular portfolio of Gröbner base algorithms the "slimgb" algorithm was slightly the fastest in our experiments. It is optimised to keep coefficients small and polynomials short on the computation and this pays off in computation time and memory usage ([22]).

It is partly based on ideas of the F4 algorithm by Faugère. It also offers a direct access to adjust the algorithm to special problem classes through its weighted length computation. However we did not exploit this parameter.

A reduced Gröbner base is unique for any given ideal and monomial ordering. However there are huge differences when switching from one order to another. Usually the "fastest" ordering is "Graded Reverse Lex Order", meaning that a monomial has the higher order if the sum of its degrees is smaller (in case the degrees sum up to the same sum, one further distinguishes by a lexicographical order).

The expected running time of 1.051E17 seconds is surprising to us, as in [8] the authors say "if Magma or Singular do not crash, then they tend to be faster" (than SAT solvers). However in our experiments the SAT solvers are about 6400-times faster – and both implementations are stable.

7 Discussing and Comparing the Results

The expected running time of the SAT solver attack is 1.64E13 seconds, while the BDD attack takes 4.22E17 seconds and the attack using Gröbner bases 1.051E17 seconds. The other main differences are that on the one hand the SAT solver attack is faster, probably because it combines several sophisticated search heuristics, but at the cost that the SAT solver can almost only be used as a black box. While on the other hand the BDD attack uses heuristics only in the reordering algorithm. Also the BDD in construction, can already be interpreted in every step, as it represents all possible internal states of the cipher at this point. Also there are some theoretical bounds for the BDD attack, as published in [17]. We used the Gröbner bases attack also almost only as a black box, however the theory offers more insight and optimisation as sketched above.

Another resource to compare the 3 attack concepts on, is the amount of memory required: SAT solvers require almost no memory, while BDDs use up to 1 GB and Gröbner bases up to 400 MB. All three attacks can be easily parallelised due to the guessing loop. If we had several CPUs we would just divide the guessing-space among them, leading to parallelisation without overhead, as no communication between the processes is required. The amount of keystream needed is very low for all attacks (about 200 bits).

The influence of randomness is much higher in the SAT solver attack, while the BDD attack only uses randomness for guessing the bits in the beginning. This also supports the fact that the variance in the running times of SAT solvers is much higher (up to a factor of 20) while the variance for BDD running times is below a factor of 2. The variance for the Gröbner base approach is also rather low, especially when computing the same instance twice there is almost no variance in the running times.

For all three attack concepts we had to decide, which part of the internal state we wanted to guess. It turned out that for SAT solvers and BDDs it is most useful to guess the internal bits close to the end of the 2 registers. The difference is that it is optimal for the SAT solvers to only guess the end of the second register and for the BDDs, it is optimal to share the guessed bits between the ends of both registers. Looking at the equation system describing Bivium, we notice that the variables close to the output occure rather frequent (those of the second register a little more than those of the first). This might be one reason, why these guessing strategies helped most.

We also implemented an exhaustive search on the keysetup, this resulted in an expected running time of 1.5E17 seconds.

7.1 Comparing Against Other Attacks

We want to cite some more results that we are aware of, but have not implemented ourselves: In [4] Raddum proposed to solve the equation system by a graph-theoretic approach, resulting in a running time of about $2^{56} \approx$ 7.2E16 seconds. The attack published in [6] gives a running time of about $c \cdot 2^{36.1}$, for $c \approx 2^{14}$, leading to $2^{50.1} \approx$ 1.2E15 seconds. In [9] the SAT solver attack has also been implemented – we were not able to reconstruct the results, so we just quote them: when guessing 34 variables the average running time using MiniSAT is given as $2^{8.7}$, leading to an expected running time of $2^{33} \cdot 2^{8.7} = 2^{41.7} \approx$ 3.57E12 seconds. A classic time-memory trade-off technique based on the birthday paradoxon ([7]) gives a theoretical running time of $O(\sqrt{2^{177}})$ (also for the amount of memory needed).

8 Outlook

The attack concepts based on SAT solvers, BDDs and Gröbner bases are generic, so one could run many more experiments to get cryptoanalytic results also on other stream ciphers, not just for Bivium. Of course one open question is how to extend the attack to Trivium. Also we believe that there is much potential for optimisation left. For example one could try to guess the variables not just independently with probability one-half 0 or 1. We expect further speedups for the SAT attack, by just applying the latest SAT solvers that will lead to faster running times.

While SAT solvers are much faster and need almost no memory the huge disadvantage is that they can almost only be used as a black box. This might be a starting-point for further research. Also we hope to show by this paper that there is a very interesting benchmark class for industrial SAT solvers. An expected running time of about 533,000 years for the attack of course does not break Bivium and especially not the harder cipher Trivium. However parallelisation is possible without overhead and combined with further improvements this attack concept could become practicable.

References

1. eSTREAM: *eSTREAM – The ECRYPT Stream Cipher Project.* `http://www.ecrypt.eu.org/stream/`

2. NESSIE: *NESSIE – New European Schemes for Signatures, Integrity and Encryption.* `https://www.cosic.esat.kuleuven.be/nessie/`
3. C. De Cannière, B. Preneel: *TRIVIUM – a stream cipher construction inspired by block cipher design principles.* eSTREAM, ECRYPT Stream Cipher Project, Report 2005/030, 2005. `http://www.ecrypt.eu.org/stream/trivium.html`
4. H. Raddum: *Cryptanalytic results on TRIVIUM.* eSTREAM, ECRYPT Stream Cipher Project, Report 2006/039, 2006. `http://www.ecrypt.eu.org/stream`
5. D. Le Berre, L. Simon: *Special Volume on the SAT 2005 competitions and evaluations.* Journal of Satisfiability (JSAT), March 2006. `http://www.satcompetition.org/`
6. A. Maximov, A. Biryukov: *Two Trivial Attacks on Trivium.* Selected Areas in Cryptography 2007, pp. 36-55, 2007.
7. A. Biryukov, A. Shamir: *Cryptoanalytic time/memory/data tradeoffs for stream ciphers.* Proceedings of ASIACRYPT 2000, LNCS 1976, pp. 1-13, 2000.
8. G. Bard, N. Courtois, C. Jefferson: *Efficient Methods for Conversion and Solution of Sparse Systems of Low-Degree Multivariate Polynomials over GF(2) via SAT-Solvers.* Cryptology ePrint Archiv, Report 2007/024, 2007.
9. C. McDonald, C. Charnes, J. Pieprzyk: *Attacking Bivium with MiniSat.* Cryptology ePrint Archive, Report 2007/040, 2007.
10. DIMACS specification: `http://www.satlib.org/Benchmarks/SAT/satformat.ps`
11. K. Pipatsrisawat, A. Darwiche: *RSat 2.0: SAT Solver Description.* Technical report D153. Automated Reasoning Group, Computer Science Department, University of California, Los Angeles, 2007. `http://reasoning.cs.ucla.edu/rsat/`
12. Niklas Een, Niklas Sorensson: *MiniSat – A SAT Solver with Conflict-Clause Minimization.* Proc. Theory and Applications of Satisfiability Testing (SAT'05), 2005.. `http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/MiniSat.html`
13. I. Wegener: *Branching Programs and Binary Decision Diagrams.* SIAM Monographs on Discrete Mathematics and Applications. SIAM, Philadelphia, PA, 2000.
14. M. Krause: *BDD-Based Cryptanalysis of Keystream Generators.* Proceedings of EUROCRYPT 2002, LNCS 2332, pp. 239-237, 2002.
15. M. Krause: *OBDD-Based Cryptanalysis of Oblivious Keystream Generators.* Theory of Computing Systems 40(1), pp. 101-121, 2007.
16. M. Krause, D. Stegemann: *Reducing the space complexity of BDD-based attacks on keystream generators.* Proceedings of FSE 2006, LNCS 4047, pp. 163-178, 2006.
17. D. Stegemann: *Extended BDD-based Cryptanalysis of Keystream Generators.* Proceedings of SAC 2007, LNCS 4876, pp. 17-35, 2007.
18. F. Somenzi: *CUDD*, version 2.4.1, University of Colorado, `http://vlsi.colorado.edu/~fabio/CUDD/`
19. W. Stein: *Sage Mathematics Software (Version 2.9.2)*, The SAGE Group, 2007, `http://www.sagemath.org`.
20. G.-M. Greuel, G. Pfister, H. Schönemann: *Singular* 3.0.4. A Computer Algebra System for Polynomial Computations. Centre for Computer Algebra, University of Kaiserslautern, 2007, `http://www.singular.uni-kl.de/`
21. B. Buchberger: *Gröbner Bases: A Short Introduction for System Theorists.* Proceedings of EUROCAST 2001, LNCS 2178, pp. 1-14, 2001.
22. M. Brickenstein: *Slimgb: Gröbner Bases with Slim Polynomials.* Reports on Computer Algebra 35, ZCA, University of Kaiserslautern, 2005.