Cryptanalytic Results on Trivium

Håvard Raddum

Department of Informatics, University of Bergen, N-5020 Bergen, Norway

Abstract. Trivium is a stream cipher submitted to the eSTREAM project in ECRYPT. It has a simple and elegant design and is very fast, and so far no weaknesses have been reported. In this paper we use a novel technique to try to solve a system of equations associated with Trivium. Due to the short key-length compared to the size of the internal state of Trivium (80 to 288 bits), we do not get an efficient attack on the full Trivium, but reduced versions corresponding to the design's 'basic construction', is broken by this approach.

1 Introduction

The eSTREAM project within ECRYPT issued a call for secure and fast stream ciphers in November 2004. The response from the cryptographic community has been good, with 34 proposals submitted. Some of these have been broken, and others again have exhibited weaknesses. Since eSTREAM allows "tweaks" (minor modifications to algorithm), several of the attacks have been made invalid after changing some of the specification components.

Trivium [1, 2] has remained unchanged since it was submitted, and at the time of writing there is only one paper [3] on the eSTREAM website that has any cryptanalysis of Trivium. The conclusion of this paper is that Trivium is secure against a linear sequential circuit approximation attack.

In this paper we set up systems of sparse equations describing the full Trivium and reduced versions, and try to solve them by using a new technique described in [4], but as yet unpublished in a proper journal or proceedings. By this approach we can show that the full Trivium is still not broken, but that reduced versions with two registers instead of three is broken significantly faster than exhaustive search. One point we would like to emphasize is that since our approach is algebraic in nature (solving equation systems) the attack requires very little known key-stream, as opposed to most other types of attacks that typically requires enormous amounts of known key-stream. This makes our kind of attack much more threatening in a real-world setting.

The paper is organized as follows. As our technique is unfamiliar to most, in Section 2 we will thoroughly describe the method of solving equation systems we use, with an example to help the reader appreciate the ideas behind this approach. In Section 3 we specify how the reduced versions of Trivium were made, and show how to build an associated sparse equation system from these. In Section 4 we present the results on the complexity of solving these systems, and conclusions are drawn in Section 5.

2 Constructing a graph from a sparse system of equations

In this section we will show how to build a graph from a system of equations. Throughout the whole paper we will only consider equations where the variables have values in $GF(2)$. We will describe the graph construction in the general case, where we assume we are given a system of m equations $E_1, \ldots, E_m$ in n variables $x_0, \ldots, x_{n-1}$. To help the reader get familiar with the methods we will use the following specific example system along the way:

$$\begin{aligned}
&E_1 : x_2x_0 + x_1x_0 + x_2 + x_0 = 0\\
&E_2 : x_3x_1x_0 + x_3x_1 + x_3x_0 + x_3 + x_1 + x_0 + 1 = 0\\
&E_3 : x_4x_0 = 0\\
&E_4 : x_4x_3x_1 + x_3x_1 + x_1 = 0\\
&E_5 : x_4x_2 + x_4 + x_2 + 1 = 0
\end{aligned}$$

2.1 Constructing the graph

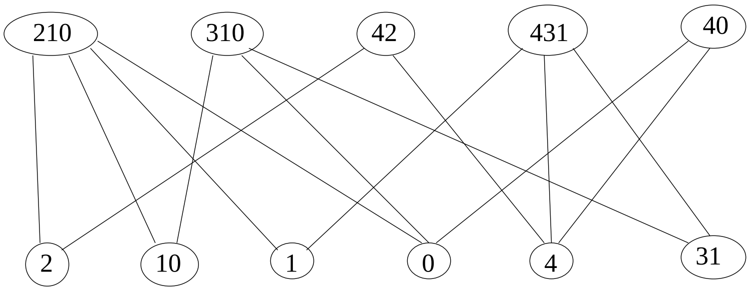
Each equation in our system will correspond to one vertex in the graph. This vertex will be labeled by the indices of the variables that make up the corresponding equation, where the indices are listed in decreasing order. In our example system, the vertex corresponding to E_1 will have the label (210). Each vertex in the graph will have a set of variables associated with it.

Definition 1. *A vertex in the graph is called a* **symbol**, *and the variables associated with symbol S is denoted by $X(S)$.*

We initialize the graph construction by creating one symbol for each equation. The rest of the graph is created according to the following procedure:

> For each pair of initial symbols S and T, let $u = X(S) \cap X(T)$. If there already exists a symbol Q with $u = X(Q)$, we draw edges from Q to S and T, if these edges do not already exist. If there is no symbol with label u, a symbol Q is created with $u = X(Q)$, and edges are drawn from Q to S and T.

There will be no edges between initial symbols, or between symbols created by intersecting variable sets of initial symbols, so the graph will be bipartite. Following this simple recipe, the graph associated with the example system will look like this:



2.2 Transferring the information in the equation system to the graph

We have shown how to make the vertices and the edges in the graph associated with a system of equations. To make the correspondence between the system of equations and the graph complete, we need to carry the information the system has about possible solutions into the graph. To do this we use the following definition.

Definition 2. *A* **configuration** *for an equation E is an assignment of values to the variables in E.*

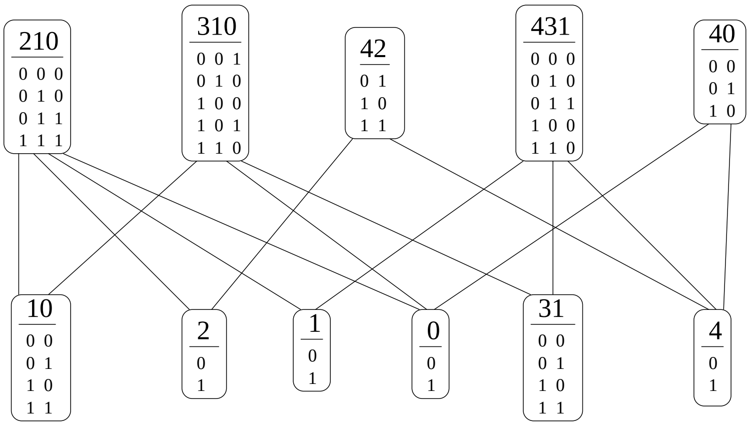
An equation E corresponds to a symbol S, and a configuration for E (equivalently, for S) can be represented as a bit-string of length $|X(S)|$. A bit in a specific position in the string contains the value for the variable located in the same position in $X(S)$.

We give every symbol corresponding to an equation a list of all configurations satisfying the associated equation, and denote the list of configurations for symbol S by $L(S)$. This will carry all information every equation has about the solution, into the graph. Each symbol not associated directly with an equation will also have a list of configurations. The list of configurations for one of these symbols, S, will initially consist of all bit-strings of length $|X(S)|$.

Note that producing the list of configurations for an equation containing r variables has complexity 2^r. We need (in general) to run through all 2^r strings of length r to find out which configurations that satisfy the equation, and which that do not. Also, the expected size of the configuration list of a boolean equation is 2^{r-1}.

This means we require our system of equation to be *sparse*, in the sense that no single equation consists of too many variables. Systems coming from ciphers tend to have this property, depending on how many extra variables that are introduced when creating the equations. As we shall see, the system we create from Trivium has 954 variables all together, but no single equation contains more than 6 of them, and no symbol S has more than 32 configurations in its $L(S)$.

Let us now compute the configuration lists of the equations of our example system, and add them into the graph. The reader may verify that the configuration lists of the five equations are the ones shown below. The symbols not corresponding to equations have full configuration lists.



2.3 Solving the system by message-passing

We now proceed to describe the core algorithm for solving the system of equations. Every solution of the system can be represented as an n-bit string, containing the values for $(x_{n-1}x_{n-2}\ldots x_0)$ that solves the system. Each equation in the system has to be satisfied with a solution, so for any symbol S associated with an equation, concatenating bits from a solution into a string for $X(S)$ will produce a configuration found in $L(S)$.

Our idea is to delete configurations that can not be part of a solution from the various $L(S)$. The goal is to remove all configurations from the lists, except for those that are part of a correct solution. If we are able to do this we can simply read the values of the remaining configurations in the symbols to get a solution of the system.

Symbols that are connected by edges can send messages about their configuration lists to each other. A symbol receiving a message may use that information to identify configurations that can not possibly be part of a solution. How this can be done is detailed below, using the following definition.

Definition 3. *Let S and T be two symbols such that $X(S) \subset X(T)$. A configuration for $X(T)$ is said to* **cover** *a configuration for $X(S)$ if the two are equal for all variables in $X(S)$.*

2.4 Sending messages downwards

Let the symbols S and T be connected with an edge, where T corresponds to an equation, so $X(S) \subset X(T)$. A message from T to S is a list of configurations for $X(S)$, containing exactly those configurations that are covered by at least one configuration from the list in T. These are the only possible values for the symbol S if T is to be satisfied.

When S receives the message, all configurations in $L(S)$ not found in the message will be deleted. These configurations can not be part of a solution since it is impossible to satisfy T with these values.

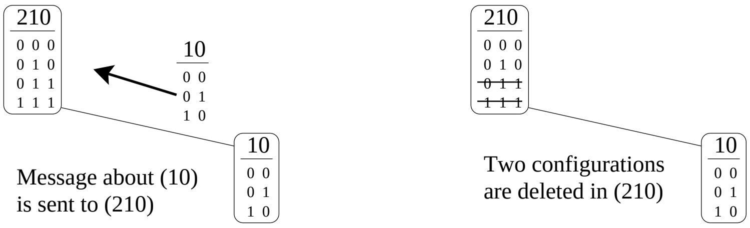
Example: The symbol (310) sends a message to (10):

2.5 Sending messages upwards

Let us still consider two symbols S, T, where $X(S) \subset X(T)$. The message S sends to T is simply $L(S)$.

When T receives the message, any configuration in $L(T)$ that does not cover any configuration in the message will be deleted. None of the configurations deleted can be part of a solution, since it is impossible to satisfy S with them.

Continuing the example above, we let (10) send a message to (210):



2.6 Making the graph agree

As can be seen from the examples above, the symbol (310), representing equation E_2, has communicated knowledge about the solution to (210) (equation E_1) through the symbol (10). This new information sets (210) in a position where it can send a message to (2), since $x_2 = 0$ is the only possibility if (210) is to be satisfied with its current configuration list. This shows that sending some messages can open up for others to be sent. In this way, we can get a chain-reaction of messages, deleting all configurations except for those corresponding to a solution.

The example system will be solved this way. We have got to the point where the symbol (2) knows its only possible value is 0. This message can be sent to (42), causing it to delete all configurations except for 10. This only leaves the possibility of the value 1 for the symbol (4), etc, etc. In the end, every symbol will only be left with one configuration, since the system has a unique solution $(x_4, x_3, x_2, x_1, x_0) = (1, 1, 0, 0, 0)$.

We call the process of sending useful messages, messages that actually cause deletion of configurations, the *Agreeing* algorithm. When the graph is left in a state where no more useful messages can be sent we say that all the symbols in the graph *agree*.

Of course, making the graph agree does not necessarily solve the associated equation system. If the amount of readily available information in the system is below some critical mass, the chain-reaction of sending useful messages will die out before we can see a solution, or it may be that the initial state of the graph does not allow any useful messages to be sent in the first place. The next subsections deal with two strategies for overcoming this problem.

2.7 Splitting

The first of the methods for re-starting the Agreeing algorithm we have called *Splitting*, and is quite simple. When the graph reaches an agreeing state with no visible solution what we do is the following: We focus on one symbol S, and split $L(S)$ in two parts, L_1 and L_2. We then replace $L(S)$ with L_1 or L_2 and start the Agreeing algorithm again. The correct configuration for S (assuming unique solution) is found on either L_1 or L_2, so what we are basically doing is guessing on which part that contains this configuration. If it becomes clear that the guess was wrong, we will run the Agreeing algorithm with the other list instead.

Guessing only once is generally not enough help for the Agreeing algorithm to solve the system. In general, the symbols will again come to an agreeing state with no apparent solution after the first guess. Then we need to guess again, run the Agreeing algorithm once more, and so on. Each time we guess, we are guessing one bit of information, since the correct configuration is found in either L_1 or L_2. One can see that guessing on the value of a specific variable is a particular case of splitting.

When will it become clear that a guess was wrong? When we make a wrong guess somewhere, we are deleting the correct configuration from a symbol's configuration list, making it impossible to find a solution. What happens, possibly after making more guesses, is that the Agreeing algorithm deletes all configurations in some $L(S)$. When this happens we will backtrack to the last guess made, and try the other possibility. If this also turns out to be a dead end, we will backtrack to the guess before that, and try the other possibility from that point, etc.

What we are doing is going through a binary search tree, looking for the solution of the system. When we follow the branch of the tree corresponding to the correct guesses we find a solution. The leaves of this tree will be the point where we either find a solution, or the points where some $L(S)$ becomes empty. The leaves will be at somewhat varying depths, and the complexity of this approach will be exponential in the average depth of the tree.

2.8 Gluing

The other main strategy for making the Agreeing algorithm start again we call *Gluing*. With this method we do not do any guessing, but instead merge two symbols into one.

Let symbols S and T be given, and let $X(S) = A \cup B$, $X(T) = B \cup C$ with $A \cap C = \emptyset$. We will make a new symbol Z, with $X(Z) = X(S) \cup X(T)$ and and define the list $L(Z)$ as follows. The list $L(Z)$ consists of all configurations (a, b, c), where b is a configuration for B, $(a, b) \in L(S)$, and $(b, c) \in L(T)$. In other words, $L(Z)$ consists of all configurations that cover one configuration in $L(S)$ and one configuration in $L(T)$. The symbol $Z = S \circ T$ is the result of gluing the symbols S and T, and the configuration corresponding to the solution of the system is found on $L(Z)$.

When our symbols reach an agreeing state, we can glue together symbols to create new symbols made from larger sets of variables, with more open information about the solution. When gluing S and T together we discard S and T, since all information in them are contained in $S \circ T$. After gluing together several pairs of symbols the new set of symbols will in general not be in an agreeing state. We can then construct a new graph from them, and start the Agreeing algorithm again.

The price to pay when gluing together symbols is longer configuration lists. Assuming S and T agree, the number of configurations in $S \circ T$ will be at least as big as $\max\{|L(S)|, |L(T)|\}$, and may be as big as $|L(S)| \cdot |L(T)|$, depending on the size of $X(S) \cap X(T)$. In practice we have to set a threshold and only glue together symbols that produce configuration lists with size below this threshold. This means we may run into cases where we can not afford any symbols to be glued.

3 Trivium and its reduced variants

Trivium has a simple and elegant design, making it a tempting candidate for cryptanalysis, but so far no weaknesses have been reported. It is also very fast, making it a strong candidate in eSTREAM as long as it is not broken.

Trivium operates on three registers of length $93, 84$ and 111 bits. These registers may also be regarded as concatenated into one register $(s_1, \ldots, s_{288})$ of 288 bits. The following pseudo-code copied from [2], is a compact but complete specification of how to generate one bit z of the key-stream:

$$
\begin{aligned}
t_1 &\leftarrow s_{66} + s_{93} \\
t_2 &\leftarrow s_{162} + s_{177} \\
t_3 &\leftarrow s_{243} + s_{288} \\
z &\leftarrow t_1 + t_2 + t_3 \\
t_1 &\leftarrow t_1 + s_{91} \cdot s_{92} + s_{171} \\
t_2 &\leftarrow t_2 + s_{175} \cdot s_{176} + s_{264} \\
t_3 &\leftarrow t_3 + s_{286} \cdot s_{287} + s_{69} \\
(s_1, s_2, \ldots, s_{93}) &\leftarrow (t_3, s_1, \ldots, s_{92}) \\
(s_{94}, s_{95}, \ldots, s_{177}) &\leftarrow (t_1, s_{94}, \ldots, s_{176}) \\
(s_{178}, s_{179}, \ldots, s_{288}) &\leftarrow (t_2, s_{178}, \ldots, s_{287})
\end{aligned}
$$

This process is repeated until enough key-stream bits have been produced.

Trivium takes an 80-bit key $(K_1, \ldots, K_{80})$ and an 80-bit IV $(IV_1, \ldots, IV_{80})$ for initializing the registers. This is done as follows:

$$
\begin{aligned}
(s_1, s_2, \ldots, s_{93}) &\leftarrow (K_1, \ldots, K_{80}, 0, \ldots, 0) \\
(s_{94}, s_{95}, \ldots, s_{177}) &\leftarrow (IV_1, \ldots, IV_{80}, 0, \ldots, 0) \\
(s_{178}, s_{179}, \ldots, s_{288}) &\leftarrow (0, \ldots, 0, 1, 1, 1)
\end{aligned}
$$

The registers are then clocked $4 \cdot 288$ times without producing any key-stream, and after this key-stream generation is ready to start.

3.1 System of equations representing Trivium

We regard the state of the registers at the time when key-stream generation is about to start as the initial state. We label these bits with the unknown variables $s_1, \ldots, s_{288}$, so at this point we have 288 unknowns and no equations. When clocking Trivium once, we introduce three new bits to the next state, which can be expressed as non-linear combinations of some of the bits from the initial state. In order to keep the future equations sparse enough, we let these three bits become new variables, and get the following equations from the first clocking:

$$
\begin{aligned}
s_{289} &= s_{66} + s_{91} \cdot s_{92} + s_{93} + s_{171} \\
s_{290} &= s_{162} + s_{175} \cdot s_{176} + s_{177} + s_{264} \\
s_{291} &= s_{243} + s_{286} \cdot s_{287} + s_{288} + s_{69}
\end{aligned}
$$

In addition to this we get one equation from the known key-stream bit z:

$$
z = s_{66} + s_{93} + s_{162} + s_{177} + s_{243} + s_{288}
$$

This procedure is repeated, so for each clocking of the cipher we get four equations and three new variables. After clocking the cipher 288 times we have 1152 equations in 1152 variables, so the knowledge of 288 consecutive key-stream bits should be enough to identify the initial state uniquely.

We can reduce the size of this system a little. Note that towards the end of generating the equations, the new variables we introduce will actually never be used in an equation given from the key-stream. The new bits entered into the registers are not used for key-stream generation before the cipher has been clocked 66 more times. This means that for the last 66 clockings, we can drop the three equations representing the new bits coming into the registers, and just make the equation from the key-stream bit without introducing new variables. This yields a system of 954 equations in 954 variables, where all equations consist of 6 variables and have a list of 32 possible configurations.

If we can solve this system, we find the initial state of the cipher at the time key-stream generation started, but we do not recover the key directly. It is a straightforward task to compute the previous state from a given state, so knowing the state of the cipher at some point we can clock Trivium backwards to reach the state where the key and IV were loaded. Hence solving our system is equivalent to finding the key.

3.2 Reduced variants

In [1, Fig. 4], the designers of Trivium describe what they call the 'basic construction' of the Trivium design. This is like the full Trivium, but with two registers instead of three. This smaller cipher comes in two slightly different versions. One where the key-stream bit produced is the sum of two bits from the internal state, and one where the key-stream bit is the sum of four bits from the state. We will call these variants for Bivium A and Bivium B, respectively.

We have made a specification of these two ciphers, trying to keep them as close as possible to the full Trivium. Both variants have two registers of length 93 and 84, and the state of these is given as $(s_1, \ldots, s_{177})$. The pseudo-code for clocking Bivium A once is then given as

$$\begin{aligned}
t_1 &\leftarrow s_{66} + s_{93} \\
t_2 &\leftarrow s_{162} + s_{177} \\
z &\leftarrow t_2 \\
t_1 &\leftarrow t_1 + s_{91} \cdot s_{92} + s_{171} \\
t_2 &\leftarrow t_2 + s_{175} \cdot s_{176} + s_{69} \\
(s_1, s_2, \ldots, s_{93}) &\leftarrow (t_2, s_1, \ldots, s_{92}) \\
(s_{94}, s_{95}, \ldots, s_{177}) &\leftarrow (t_1, s_{94}, \ldots, s_{176})
\end{aligned}$$

The specification of Bivium B is exactly the same, except that $z \leftarrow t_2$ is replaced by $z \leftarrow t_1 + t_2$.

For both variants, the key and IV loading is the same as in Trivium (the third register in Trivium, initialized with constants, is removed), and the ciphers are clocked $4 \cdot 177$ times before any key-stream is produced.

The equation systems representing Bivium A and B can be made the same way as for the full Trivium. Each clocking introduces two new variables and three new equations. After 177 clockings we get a system of 531 equations in 531 unknowns, but like for Trivium we can remove the last $2 \cdot 66$ equations representing new variables since these variables are not used in any key-stream equation. We then get systems of 399 equations in 399 unknowns.

The equations tying new variables to old ones are made up of 6 variables each, but the key-stream equations contain 4 variables each for Bivium B, and only 2 variables for Bivium A. This difference will turn out to be crucial for the complexities of solving these systems with the methods proposed in Section 2.

4 Solving the Equation Systems

Here we will report on the complexities of solving the equation systems representing Bivium A and B and Trivium by using the techniques described in Section 2.

4.1 Bivium A

Bivium A is the simplest of the three ciphers considered, and should have the equation system that is easiest to solve. Indeed, this is also the case. The depth of the search tree when running the Splitting algorithm is roughly 29. The depth of the tree can be smaller by gluing some of the symbols together before starting the Splitting algorithm. If we glue together as many pairs of symbols we can, allowing new symbols to have configuration lists of size up to 1024, we get a graph where only 150 symbols correspond to equations. The depth of the search tree is then about 21, and our program returned the correct initial state in about one day. Bivium A is very weak.

4.2 Bivium B

The only difference between Bivium A and B is that key-stream equations from Bivium B have four variables instead of two. This is enough to make the complexity of solving Bivium B's associated system of equations much higher than for Bivium A.

The best approach we found for solving this system is to glue together pairs of symbols yielding new symbols with configuration lists of size 32, and then apply the Splitting method.

The complexity of the Splitting method is $\mathcal{O}(2^d)$, where d is the depth of the tree we are searching. The leaves of this tree appears at slightly different depths, and to get the most accurate result we should look at several different parts of the tree when estimating its average depth. We counted the number of leaves in 8 random sub-trees with root node at depth 46 in the whole tree, and timed how long it took to search through this 2^{-46}-part of the whole search space. Let the number of leaves and the times recorded be l_i and t_i, $i = 1, \ldots, 8$. The depth of the whole tree can then be estimated as $d \approx 43 + \log_2(\sum_{i=1}^{8} l_i)$, and the time it takes to search through the whole tree can be estimated as $2^{43} \sum_{i=1}^{8} t_i$. The data recorded are summarized in the following table

Time visiting 2^{-46}-part (seconds)	# of leaves in sub-tree
$2^{10.2}$	81659
$2^{4.3}$	1159
$2^{9.6}$	50844
$2^{7.4}$	11386
$2^{10.2}$	81661
$2^{11.5}$	207045
$2^{6.1}$	4362
$2^{10.9}$	143648

From this we estimate the average depth of the tree to be 62.15, and that it will take 2^{56} seconds to search through the whole tree. The depth of the tree is significantly smaller than 80, which will be the depth of the "tree" when doing an exhaustive search for the key, so this looks like a valid attack on Bivium B. On the other hand it probably takes longer time to visit a leaf in the search tree of the Splitting algorithm, than it takes to try one key in Bivium B, so we have estimated how long time it will take to search through all 2^{80} keys in Bivium B.

It should be noted that trying one key in Bivium B is not without complexity, since we have to clock the cipher 708 times before we can start comparing the key-stream produced to see if the guess was right or wrong. Using our implementation of Bivium B we could search through 2^{24} keys in $2^{10.7}$ seconds, which means we can only search through $2^{69.3}$ keys in the same time it takes the Splitting algorithm to break the cipher. One may argue that there are faster implementations of Bivium B than we have used, but there is probably more room for optimization in our implementation of the Splitting algorithm since this software is more complex than the program guessing keys in Bivium B.

4.3 Trivium

We have not been able to break the full Trivium with our approach. The best result we have been able to get is when we glue together any pair of symbols yielding a new symbol with at most 1024 configurations. This gives a graph with only 477 symbols corresponding to equations, down from 954 in the original graph. After applying the Splitting algorithm on this graph we find that the depth of the search tree is about 164, which gives a complexity far higher than exhaustive search.

Classically, stream ciphers have often been designed with the initial state of the registers being the key. Ciphers employing this design have usually been broken. Newer stream ciphers, like Trivium, tend to have a user selected key shorter than the size of the internal state, and a key runup phase where the key-bits are mixed in a complex way into the registers before any key-stream is produced. The key runup phase makes things more difficult for the cryptanalyst. For example, creating sparse equations through the key runup phase will produce a lot of (more than 1500) new variables before any information is leaking out in the form of key-stream bits. In our approach we therefore have to regard each of the 288 bits in the state at the start of the key-stream generation as a variable, even though they are derived from only 80 unknown bits. From this viewpoint, finding 288 unknown bits with complexity $\mathcal{O}(2^{164})$ may not be so bad, it would make a strong attack if the key in Trivium had been the internal state at the start of the key-stream generation.

5 Conclusions

In this paper we have proposed a technique for solving equation systems that differs a lot from other known methods like Gröbner base computations and re-linearization. Our technique has been applied to equation systems representing Trivium and the two smaller versions from [1], in this paper called Bivium A and B.

The Agreeing method, helped by Splitting and Gluing performs rather well breaking both versions of Bivium, but with a large difference in complexity between the two. The full Trivium still resists our attack due to the short key compared to the size of the internal state.

References

1. C. De Cannière and B. Preneel. *TRIVIUM A Stream Cipher Construction Inspired by Block Cipher Design Principles*, http://www.ecrypt.eu.org/stream/papersdir/2006/021.pdf, 2005.
2. C. De Cannière and B. Preneel. *TRIVIUM Specifications*, http://www.ecrypt.eu.org/stream/ciphers/trivium/trivium.pdf, 2005.
3. S. Khazaei and M. Hassanzadeh. *Linear Sequential Circuit Approximation of the TRIVIUM Stream Cipher*, http://www.ecrypt.eu.org/stream/papersdir/063.pdf, September 2005.
4. H. Raddum and I. Semaev. *New Technique for Solving Sparse Equation Systems*, Internal ECRYPT webpage, https://www.cosic.esat.kuleuven.be/ecrypt/intern/STVL/, January 2006.