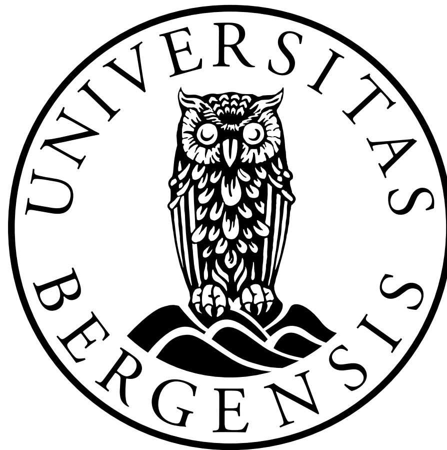
Towards Efficient Algorithms in Algebraic Cryptanalysis

Thorsten Ernst Schilling



Dissertation for the degree of Philosophiae Doctor (PhD)

The Selmer Center
Department of Informatics
University of Bergen

May 2012

Acknowledgements

First and foremost I would like to thank my supervisor Håvard Raddum for his time and dedication. During the almost four years we have worked together, he has offered me invaluable assistance and guidance.

Further, I would like to thank Pavol Zajac for the joint work we did during the Slovak-Norway collaboration in cryptology.

I enjoyed my time at the Selmer Center much thanks to the friendly atmosphere there. While I would like to thank the whole Department of Informatics at the University of Bergen, a special mention goes out to my good colleagues Tor Erling Bjørstad, Michal Hojsik, Joakim Grahl Knudsen and Somaye Yari.

During my time at the University of Washington in the spring of 2012, Peter Horak proved to be an excellent host and made my stay there very comfortable.

My friends, among them Carlos Schaffner, Torbjørn Lium, Johannes Rack, with whom I enjoyed many interesting discussions and who have had to endure me and my idiosyncrasies while in Norway, deserve an honorable mention. In the same vein, a big thank you to the families Bonness-Meyer and Havskov for good food and companionship.

Of course, I'd like to direct a big DANKE to my family, i.e., my parents, brother, grandfather (and the dog) for all the support these last 30 years.

Last but not least, I would like to thank my girlfriend Laila not only for four happy years together, but also for contributing at least 100 commas to this thesis.

List of Papers

1. Thorsten E. Schilling, Pavol Zajac, *Phase Transition in a System of Random Sparse Boolean Equations*, Tatra Mountains Mathematical Publications **45**, 1–13, 2010.
2. Thorsten E. Schilling, Håvard Raddum, *Solving Equation Systems by Agreeing and Learning*, Springer LNCS **6087**, 151–165, 2010.
3. Thorsten E. Schilling, Håvard Raddum, *Analysis of Trivium Using Compressed Right Hand Side Equations*, Springer LNCS **7259**, 18–32, 2012.
4. Thorsten E. Schilling, Håvard Raddum, *Solving Compressed Right Hand Side Equation Systems with Linear Absorption*, Sequences and their Applications, Waterloo, to appear in Springer LNCS 2012.

Contents

Acknowledgements **i**

List of Papers **iii**

1 Introduction **1**

2 Background **3**
- 2.1 Complexity Theory . . . 3
  - 2.1.1 Computational Problems . . . 3
  - 2.1.2 Landau Notation . . . 4
  - 2.1.3 P vs. NP . . . 4
  - 2.1.4 SAT Problem . . . 5
- 2.2 Graph Theory . . . 6
  - 2.2.1 Graph Problems . . . 8
- 2.3 Modern Cryptography . . . 8
- 2.4 Algebraic Cryptanalysis . . . 10

3 Computational Methods in Cryptanalysis **13**
- 3.1 Brute-Force Attack . . . 13
- 3.2 Algebraic Methods . . . 14
  - 3.2.1 Linearization . . . 15
  - 3.2.2 Gröbner Bases . . . 15
- 3.3 Boolean Constraint Propagation . . . 16
  - 3.3.1 DPLL Family Solvers . . . 17
- 3.4 Gluing & Agreeing . . . 19
  - 3.4.1 Gluing . . . 19
  - 3.4.2 Agreeing . . . 21
  - 3.4.3 MRHS . . . 22
- 3.5 Other Methods . . . 24

4 Introduction to the Papers **27**

5 Scientific Results **29**
- 5.1 Phase Transition in a System of Random Sparse Boolean Equations . . 31
- 5.2 Solving Equation Systems by Agreeing and Learning . . . 45
- 5.3 Analysis of Trivium Using Compressed Right Hand Side Equations . . 63

5.4 Solving Compressed Right Hand Side Equation Systems with Linear Absorption 81

6 Other Results **95**
6.1 Independent Set Reduction 95
6.1.1 Reduction 96
6.1.2 Graph Structure 97
6.1.3 IS-Solving Algorithm 99
6.1.4 Experimental Results 100
6.1.5 Conclusion & Further Work 100
6.2 Algorithm Unification 101
6.2.1 Syllog Preprocessing 102
6.2.2 Conclusion & Further Work 104

List of Figures

2.1 Complexity classes **P** and **NP** in case of $\mathbf{P} \neq \mathbf{NP}$. 5
2.2 Directed graph. 7
2.3 Undirected graph with edge labels. 7
2.4 Graph with independent set highlighted. 8
2.5 Attack complexities AES. 10

3.1 Expected Complexities of Agreeing/Agreeing-Gluing. 20

6.1 2-Step Reduction to IS-problem 98
6.2 Syllogism pockets for $x_i^{(\alpha)} \Rightarrow x_j^{(\beta)} \Rightarrow x_k^{(\gamma)}$. 103

Chapter 1

Introduction

Hiding information is an ancient art that dates back to before 1900 BCE. Throughout the centuries, it has evolved from rudimentary techniques of hiding simple messages to the sophisticated algorithms that are used to secure, authenticate and validate digital information today.

In the very early days of cryptography it was often not more than a distraction or curiosity for the literate. Strictly speaking, writing information down could be considered a kind of cryptosystem, since the illiterate would not able to decipher the message contained. But as soon as more than the communicating parties share a common language, written or spoken, and they want to conceal the content of their communication a more sophisticated method is needed. What would be considered a game children play in school today was serious business around 100 BCE – 44 BCE: the *Caesar cipher*. The substitution of one letter with another by shifting the alphabet (e.g., write B for A, C for B, etc.) to conceal written communication was a method used, e.g., by Caesar to share military information.

It is not known how secure this form of encryption was at the time, but our current knowledge about the history of cryptography indicates that it probably took until the 14th century to develop more sophisticated methods, like using different substitution alphabets.

One early important contribution to the field of cryptography was what is still known today as *Kerckhoffs' principle*. In 1883 Auguste Kerckhoffs stated in a journal article several design principles of ciphers. We say that the only secret information of a ciphersystem should be the key and that the system must be able to fall into the hands of the enemy without inconvenience. This means that if everything about the cipher, except the key, is known to an adversary, it should remain secure. In most cases this maxim is still followed today.

During the First and Second World Wars and the Cold War, the parties involved used all kinds of cryptographic techniques to hide their intentions. A great deal of the development of what is the foundation of today's cryptography was achieved in this time. The name Alan Turing especially stands out. Not only did Alan Turing play a central role laying the foundation for the theory of computation and algorithms, he also was involved in breaking the *Enigma* cipher. The latter earned the allies a crucial tactical advantage.

During this time, applications which required intensive computations and the increasing availability of computing devices also caused a rise in the interest for algo-

rithms. And also today the amount and complexity of data which is collected and processed by all kinds of algorithms is still growing.

Now, in the age of digital communication electronic devices which receive, store and transmit information are prevalent at our workplaces and in our homes. Today, examples of cryptographic applications and the influence of algorithms can be found in almost all aspects of daily life. From mobile phone communication to electronic payment systems, ticketing systems and even clothing labels.

Because of the huge impact it has on our daily life, understanding cryptographic methods is important. By analyzing its design and searching for weaknesses we try to achieve security.

In this thesis we introduce several techniques to represent and solve equation systems derived from ciphers. We formulate algorithms to investigate if we can practically solve such systems and do this to understand how hard it is to break a cipher.

While this chapter gives a generally understandable introduction and explains the motivation behind this work we will in Chapter 2 introduce some key concepts necessary to understand the central scientific contribution of this thesis. Chapter 3 then will explain comparable techniques which are closely related to the main contributions in this work. Chapter 4 outlines the scientific results which are presented in Chapter 5. Two other additional results are then presented in Chapter 6 which concludes the thesis.

Chapter 2

Background

In this chapter, necessary key concepts and terms are introduced. The sections should give a very brief overview about the fields of computer sciences and cryptography which are used in order to clarify the environment of the contributions in this thesis.

2.1 Complexity Theory

Computational complexity theory seeks to determine the inherent difficulty of computational problems. Difficulty can be the time needed to find an answer to a stated computational problem or the space needed to calculate it. This section introduces some central aspects which are used throughout the thesis.

2.1.1 Computational Problems

The objects investigated in complexity theory are *computational problems*. They can informally be categorized into two general classes:

1. Functional problems
2. Decision problems

The first class contains all those problems which to a given *question* (*input*) demand a certain *answer* (*output*). One example of a functional problem is the *factorization problem*. Here the input is a number r and the output is a list of the prime factors of r.

A decision problem, on the other hand, can only have the output YES or NO. An example for a decision problem is the *primality problem*, i.e., the question if a given number r is a prime. The input is the number r and the only two possible answers are YES or NO; often denoted as the binary states 1 and 0, *accept* and *reject* or TRUE and FALSE.

Decision problems can therefore be easily expressed as sets of some *formal language*[1]. An *algorithm* seeking to solve a specific decision problem is then nothing else than a set of instructions to test for the membership in this language. The *primality problem* can for example be represented by the set

$$D = \{x \in \mathbb{N} \mid x \text{ is prime.}\}$$

[1] We omit the definition of an alphabet of a formal language and assume all words of the language to be encoded in binary.

as a formal language [48]. An algorithm seeking to solve the problem for a certain input y, i.e., test if y is prime, can then be viewed as a set of instructions for testing if $y \in D$. Such an algorithm is also called a *decision procedure*. If a problem can be solved, i.e., if there exists some decision procedure for it, the decision problem is called *decidable*.

One reason why decision problems are of special interest is that they are limited in their output, can be formulated as a formal language, and are easy to study. Furthermore, it turns out that every functional problem whose output is polynomially bounded in size has an equivalent decision procedure with which it can be solved with a polynomial overhead for the reduction [24]. One can find an example of this in section 2.1.4.

2.1.2 Landau Notation

Computational problems are not only investigated for their decidability, but also for how difficult it is to decide them. That means how much resources are needed in order to decide them. The basis of the estimation is the concept of the *Turing machine*. A Turing machine is a hypothetical computing machine [30, 48] and can compute the answer to a certain decision problem with a certain amount of *steps*. A step can be seen as an unspecified unit of time.

Usually, one is interested in how many steps some algorithm $\mathfrak{A}$ takes to output YES or NO in relation to the length $n = |p|$ of some input p. Since the exact function is in most cases not really interesting or too complicated, we make use of upper bounds in the Landau notation.

Definition 1 (Landau Upper Bound). *For two functions f, g and $x, x_0, c \in \mathbb{R}$ we say $f \in \mathcal{O}(g)$ if there exists a $x_0 > 0$ such that for all $x > x_0$ it holds that*

$$|f(x)| \leq c \cdot |g(x)|$$

for some $c > 0$.

For example, we can now say that some algorithm $\mathfrak{A}$ needs $\mathcal{O}(n^2)$ steps to decide problem D. This means that there exists some length of input n_0 such that all inputs with length $n > n_0$ can be decided with at most $c \cdot n^2$ steps. This notation dates back to Edmund Landau [31]. Along with other asymptotical bounds, it predates computational complexity and is used in almost all aspects of computer sciences to express the computational complexity of a problem.

One sometimes speaks about *polynomial time* and by this refer to an amount of $\mathcal{O}(f)$ steps where f is a polynomial.

2.1.3 P vs. NP

A *complexity class* is defined as a collection of problems that have one or more characteristics. The most fundamental – and in practical applications likely the most interesting – complexity classes are **P** and **NP**.

The complexity class **P** contains all languages for which there exists a *deterministic* Turing machine such that the language can be decided in polynomial time. Examples for problems in this class are the *greatest common divisor problem* or the *shortest*

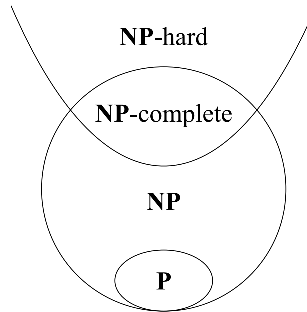


Figure 2.1: Complexity classes **P** and **NP** in case of $\mathbf{P} \neq \mathbf{NP}$.

path problem in a graph. In general, problems in this class are regarded as feasible or tractable. That usually means that larger instances can be evaluated due to the relatively moderate increase in running time, or are expected to be more easily manageable when the size of the problem instances increases.

On the other hand the complexity class **NP** contains all languages for which there exists a *non-deterministic* Turing machine such that the language can be decided in polynomial time. A non-deterministic Turing machine might be interpreted as one that *branches* at run-time into many copies and evaluates all possible computational paths in parallel [46]. Each computational path then has at most a polynomial number of steps until a YES or NO is produced, and all paths are evaluated simultaneously.

This entails certain facts. First, it is true that **P** is a subset of **NP**. Second, all problems in **NP** have a proof that can be verified in deterministic polynomial time. This yields that all non-deterministic computations can be simulated on a deterministic Turing machine, or any deterministic (Turing complete) computer for that matter. The main problem is that this may require an exponential time overhead since all computational paths of the non-deterministic Turing machine must be evaluated sequentially.

Since non-deterministic computers are so far believed to remain a thought experiment, our only tools to tackle problems in **NP** are deterministic devices. Due to the previously mentioned possible exponential overhead in time, problems from **NP** are often called infeasible. Therefore the question if $\mathbf{P} \neq \mathbf{NP}$ is crucial.

If all problems in **NP** can be *reduced* to one specific problem, we call this problem **NP**-hard. By reducing a problem instance p of some problem D to a problem D' we mean to compute some function $f : D \to D'$, s.t., $p \in D$ if and only if $f(p) \in D'$. In case this problem D' is also itself a member of **NP**, it is called **NP**-complete. Such problems are known to exist [28].

In case it would turn out that $\mathbf{P} = \mathbf{NP}$, every problem in **NP** would suddenly become feasible. This is widely regarded as highly unlikely, but not proven to be impossible. It is probably one of the most important questions in the computer sciences, and its solution would have a fundamental impact in other sciences.

2.1.4 SAT Problem

The *Boolean satisfiability problem*, or short *SAT* problem, was the first problem to be shown to be **NP**-complete [28]. That is, every problem in the complexity class **NP** can

be reduced in polynomial time onto an instance of *SAT* and by accepting or rejecting this instance find an answer to the original problem.

Finding a solution to the *SAT* problem is one alternative technique to the main contributions in this thesis. It is also an important topic in complexity theory, so the problem is outlined in the following.

A *Boolean formula* ϕ consists of literals of variables $x_0, x_1, \ldots, x_{n-1}$ using the logical operators $\wedge$ (AND), $\vee$ (OR) and parentheses. A *literal* is either a variable (x_i) or a negation of a variable ($\overline{x_i}$). The Boolean formula $\phi_{x_i=r}$ is the Boolean formula ϕ with all occurrences of the variable x_i fixed to the value r.

Definition 2 (Boolean satisfiability problem (*SAT*))**.** *Given a Boolean formula ϕ in variables $x_0, x_1, \ldots, x_{n-1}$ does there exists an assignment of* TRUE *and* FALSE *values (*1 *and* 0 *values) to the variables which makes ϕ evaluate to* TRUE *(*1*)?*

This problem is clearly a decision problem and an algorithm solving it does not obviously entail an algorithm yielding the assignments for the variables $x_0, x_1, \ldots, x_{n-1}$. However, the output of the possible function problem (the values of $x_0, x_1, \ldots, x_{n-1}$) is polynomially bounded in the size of the input. As mentioned at the end of section 2.1.1, we can derive an algorithm which solves the function problem with only a polynomial overhead utilizing the algorithm solving the decision problem.

Assume that algorithm $\mathfrak{A}$ can solve the decision version in time $\mathcal{O}(f)$ for some instance ϕ. If $\mathfrak{A}$ outputs YES and we would like to know what the assignment for $x_0, x_1, \ldots, x_{n-1}$ is we can first run $\mathfrak{A}$ on $\phi_{x_0=1}$. If $\mathfrak{A}(\phi_{x_0=1})$ answers YES we know that $x_0 = 1$, otherwise $x_0 = 0$. We can then fix the newly learned value of x_0 in ϕ and proceed for x_1 and so on. This has to be done at most n times, and therefore the running time for the function problem would be at worst $\mathcal{O}(n \cdot f)$. If the decision version for the *SAT* problem runs in polynomial time, the functional problem will, too.

Thus, if it would be possible to find an algorithm which solves the decision version of the *SAT* problem in deterministic polynomial time, we could find the assignment for $x_0, x_1, \ldots, x_{n-1}$ as well.

Restricted Versions There exist more restricted versions or special classes of the *SAT* problem which are still **NP**-complete. The most common in practical applications is the restriction in which the Boolean formula has to be in *conjunctive normal form* or CNF. That means that the input has to be in the form

$$\phi = \bigwedge_i \left(\bigvee_j x_{ij}^{(r_{ij})} \right) \tag{2.1}$$

where $r_{ij} \in \{0, 1\}$, $x^{(1)} = x$ and $x^{(0)} = \bar{x}$. The conjunction of literals, e.g., $(a \vee b \vee c)$ is then called a *clause*. Most modern solvers for the *SAT* problem operate on inputs in the form (2.1) as discussed in section 3.3.

2.2 Graph Theory

The first paper in what is today regarded as the discipline of graph theory was published by Leonard Euler [19]. In his paper he analyses the famous problem of the *Seven*

Bridges of Königsberg. By dismissing the real shape of the city of Königsberg and only considering the start and end points of the bridges, he introduced the formal concept of the graph.

This thesis utilizes graph theory mostly as a tool to describe specific algorithms and to illustrate related problems. In Chapter 6 we will show how to use graph theory, and one of its problems as a central tool to solve the problem of finding the solution to a non-linear equation system over a finite field.

In the following we will give the most common definition of two types of graphs along with some other useful related terms.

Definition 3 (Directed and undirected Graph)**.** *A graph is a pair $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. The set of points $\mathcal{V}$ is called vertices. The set $\mathcal{E}$ contains pairs of elements in $\mathcal{V}$ and is called edges. Edges may be either directed or undirected (depending on the graph type) and connect vertices. If not clear from the context, directed edges are denoted by (v_i, v_j) and undirected by v_iv_j for two vertices $v_i, v_j \in \mathcal{V}$. The directed edge (v_i, v_j) points towards v_j, i.e., expresses a connection from v_i to v_j, but not in the other direction. The edge v_iv_j expresses a connection in both directions.*

In some cases it might be useful to assign a *label* to an edge e (directed or undirected). If not stated otherwise we denote the label by $\omega(e)$.

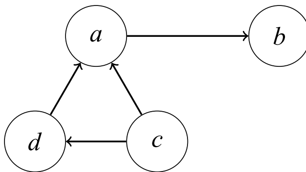


Figure 2.2: Directed graph.

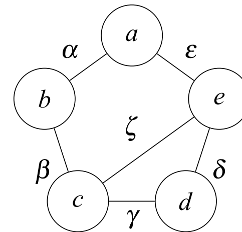


Figure 2.3: Undirected graph with edge labels.

Example 1 (Two graphs). The directed graph in Figure 2.2 contains four vertices, namely $\mathcal{V} = \{a, b, c, d\}$ and four directed edges $\mathcal{E} = \{(a, b), (c, a), (c, d), (d, a)\}$. The undirected graph in Figure 2.3 additionally contains some edge labels, e.g., $\omega(ce) = \zeta$.

The *complete undirected graph K_r* is a graph $\mathcal{G}$ with r vertices where every vertex is connected to every other with an edge. A complete subgraph of size r is also called a *clique*. If $r = 3$ then the complete subgraph is called *triangle*. The term *anti-triangle* (used in Chapter 6) denotes a set of 3 vertices, none of which are connected.

A *subgraph* $\mathcal{G}'$ of a graph $\mathcal{G}$, also denoted as $\mathcal{G}' \subseteq \mathcal{G}$, is any graph $\mathcal{G}' = (\mathcal{V}', \mathcal{E}')$ for which $\mathcal{V}' \subseteq \mathcal{V}$ and $\mathcal{E}' \subseteq \mathcal{E}$. We say that a subgraph $\mathcal{G}'$ is *induced* by a subset $V \subseteq \mathcal{V}$ if $\mathcal{G}' = (V, E)$, $V \subseteq \mathcal{V}$ and $E = \{uv \in \mathcal{E} \mid u \in V \text{ and } v \in V\}$. A subgraph *induced* by V on $\mathcal{G}$ can be also denoted as $\mathcal{G}[V]$.

A *path* in a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is any sequence of vertices $p = v_i, \ldots, v_j, v_{j+1}, \ldots, v_k$ such that for every consecutive pair of vertices v_j, v_{j+1} there exists an edge (v_j, v_{j+1}) in $\mathcal{E}$ if $\mathcal{G}$ is directed, or, for the undirected case, either $v_jv_{j+1} \in \mathcal{E}$ or $v_{j+1}v_j \in \mathcal{E}$.

A *connected component* of an undirected graph $\mathcal{G}$ is any subgraph $\mathcal{G}'$ in which every vertex v_i can reach every other vertex v_j by some path p.

A *cut-set* C between two sets of vertices A and B is here defined as the union of all edges connecting A and B. I.e., $C = \{uv \in E \mid u \in A \text{ and } v \in B\}$.

2.2.1 Graph Problems

Given the above terms and definitions, we can describe several problems in graph theory. One problem which will be discussed in connection to the central topic of the thesis is the problem of the *maximum independent set*. It is well known to be *NP*-complete [46] and thus unlikely to have an efficient algorithm that provides a solution.

Definition 4 (Independent Set)**.** *Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, an independent set is any subset of vertices $\mathcal{I} \subseteq \mathcal{V}$ where for all $u, v \in \mathcal{I}$ it is true that $uv \notin \mathcal{E}$.*

The problem of the maximum independent set is then to find the biggest subset of vertices in $\mathcal{G}$ which constitute an independent set by Definition 4. We let $\alpha(\mathcal{G})$ denote the size of the biggest independent set in $\mathcal{G}$.

Example 2 (Independent Set). In figure 2.4, a graph with one of the possible maximum independent sets colored orange. In this case $\alpha(\mathcal{G}) = 2$.

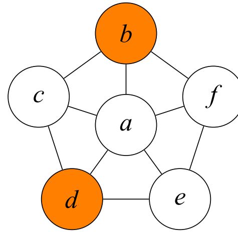


Figure 2.4: Graph with independent set highlighted.

2.3 Modern Cryptography

Formally any cryptosystem is a quintuple $(M, C, K, \mathcal{E}, \mathcal{D})$. Here M is called the *message* space (also *plaintext* or *cleartext* space), C is the *ciphertext* space and K is the *key* space. Then $\mathcal{E} = \{E_k \mid k \in K\}$ is a set of functions $E_k : M \to C$, the so called *encryption* functions. Furthermore $\mathcal{D} = \{D_k \mid k \in K\}$ is a set of functions $D_k : C \to M$, the *decryption* functions. For each $e \in K$, there exists a $d \in K$ such that for each $m \in M$ it holds that

$$D_d(E_e(m)) = m. \tag{2.2}$$

A cryptosystem is said to be *symmetric* if $d = e$ and *asymmetric* if $d \neq e$. Since we assume in the following that all operations of the cipher are carried out by some computer, we restrict M, C and K to be sets of binary strings of finite length.

The convention is that if the cryptosystem generates so called *stream bits* which are combined directly with message bits (usually *XORed*), it is called a *stream* cipher. If the cipher works on groups of bits it is called a *block* cipher.

Modern Age Cryptography Modern and ancient cryptography served the same purpose: Hiding messages and make information available only for authorized entities. The history of cryptography pre-dates modern computer sciences by around 4000 years [27], while modern cryptography started evolving mainly in the beginning of the 20th century.

One of the periods which saw the greatest evolution in cryptography was probably the Second World War. Advances in technology and a new kind of warfare led to a high demand of secret communication. The most prominent cryptographic device of the Second World War is the German *Enigma*. Like many cryptographic devices at that time it was an electro-mechanical *rotor machine*. At the time it was thought that it was impossible to break the Enigma, but a team of mathematicians, among them Alan Turing, were able to do so in the end [27]. In this context and time the work of Claude Elwood Shannon (1916-2001) – sometimes dubbed "the father of information theory" – has to be mentioned since it is to a great extend responsible for the direction modern cryptography would take. His works *Communication Theory of Secrecy Systems* [51] and *The Mathematical Theory of Communication* [52] laid the foundations for modern information theory.

After the Second World War most of the work on ciphers was carried out in secret. Cryptography found new applications in the Cold War and the increasingly important electronic communication was just one of the fields that benefited from it. Several ciphers which could be executed by hand were developed in order to be used by spies in the field.

The introduction of the micro-chip, the civilian use of computers and a further increase in electronic communication in the second half of the last century made a public cipher system necessary. The *National Bureau of Standards* – today known as *NIST* – invited IBM to develop such an algorithm. The outcome was the symmetric block cipher *Data Encryption Standard* (DES), which was adopted as a national standard in 1977 [38]. As recommended by the *National Security Agency* (NSA) the key size was 56 bits.

Another development in 1976 was the introduction of public key cryptosystems by Whitfield Diffie and Martin Hellman [17]. A public key cryptosystems allows the sender of a message to encrypt a message with a publicly known key. An adversary who only knows the public key has no access to the message since a mathematically related private key is necessary to decrypt the ciphertext. Since only the public key is used for encryption, the private key can be kept secret and only available to the receiver. With symmetric cryptography, trusted parties or secure channels are needed to distribute key material. With public key cryptography, the keys which are only used for encryption can be sent over an insecure network. The secret private key used for decryption never has to leave the owner. This made the key distribution massively easier.

Later, stronger symmetric block ciphers, like *Triple DES* were introduced since the security of DES became questionable due to its short key [39]. In 2001, after an open competition, NIST selected the algorithm *Rijndael* as the new *Advanced Encryption Standard* (AES) [40]. It is a symmetric block cipher in the form of a *substitution permutation network* and supports a key length of 128, 192 or 256 bits. In order to encrypt a message, the cipher works on blocks of 128 bits. In 10, 12 or 14 *rounds* (depending on the key size), the same set of linear and non-linear operations with a changing *round key* are applied to the message. It is used in all kinds of applications

and devices and its security is therefore crucial. As of today, the best known attack on AES was achieved by Bogdanov et.al. in 2011 [7]. The complexities of the attack with the different key sizes is summarized in figure 2.5. Technically, the cipher is broken.

Key length	Operations
128	$2^{126.1}$
192	$2^{189.7}$
256	$2^{254.4}$

Figure 2.5: Attack complexities AES.

Nevertheless, the time to execute the attack is still prohibitively long and therefore not practical. Thus, for all practical applications AES is still considered safe.

Other developments in the field were the European ECRYPT eStream project between 2004 and 2008 [1]. Similar to the NIST competition held in order to find AES, the eStream project was aimed at creating a portfolio of stream ciphers. During the project, several stream ciphers were broken and in the end seven different algorithms were included in the portfolio. One of these ciphers is presented in detail and analyzed in the contributions of this thesis.

Another important event is the ongoing NIST hash competition to find the new standard SHA-3 [54]. In the same fashion as in the other competitions, submissions are evaluated in several rounds. In every round candidates are discarded if they are, e.g., not well understood or their security can be shown to be insufficient. As of today, there are four finalists and the winner will be announced later in 2012.

2.4 Algebraic Cryptanalysis

One condition for a symmetric cryptosystem (2.2) to be called *secure* is that if we are given pairs

$$(m_0, E_k(m_0)), (m_1, E_k(m_1)), \ldots (m_{r-1}, E_k(m_{r-1})) \tag{2.3}$$

it should be computationally infeasible to calculate the key k faster than exhaustive search. For a cipher system with a key length of $|k|$ bits an attacker should be required to execute $\mathcal{O}(2^{|k|})$ operations on the average to find k. If an attacker can find a method to compute k in $\mathcal{O}(2^{\varepsilon})$ steps with $\varepsilon < |k|$ we say that the cipher is broken.

This case is called a *known plaintext* attack and every pair is called a *plaintext/-ciphertext* pair. There exist other, e.g., *plaintext only*, *chosen plaintext* or *related key* attacks, but the known plaintext attack is probably the most popular in algebraic cryptanalysis.

Constructing Equation Systems As (2.2) already suggests, it should be possible to formulate a cryptosystem in the form of some equation or set of equations. All unknowns in this system are then variables, and all the information which is known can be inserted as constants.

To construct the equation system one wants to create equations which connect plaintext bits, ciphertext bits, key bits and intermediate values. This can be done in different ways and there might exist several possibilities to construct such a system.

For example, if given a round based symmetric block cipher based on non-linear and linear operations, all bits in the block can be given variable indices. If a non-linear operation is applied to one or more of these bits a new variable is introduced. The input and output bits of non-linear operations can then be described as linear combinations of these variables. Known variables can be substituted with the appropriate values. One example for generating such a system for AES can be found in [13].

Symmetric stream ciphers usually work by *XOR*ing one-by-one *stream bit* and message bit. Then the decryption function is exactly the same function as the encryption function. We can then see most symmetric stream ciphers as *pseudo random number generators*. As the name suggests, their output are pseudo random bits, the *key stream*. We often assume that we know a portion of the key stream and write the equations as updates of the inner state in relation to the output. Solving such a system might give us an inner state of the cipher, information which usually has a simple relation to the secret key.

There also exists examples of equation systems for other cipher systems, see for example [5].

Solving the System Secret information of the cipher is now given in the form of variables, and the next chapter will give an outline of various different techniques used in order to solve a system of equations. If one can solve the equation system faster than the time necessary for an exhaustive key search, the cipher is broken.

In this process it is quite common to create *reduced* versions of ciphers for experimentation, e.g., the small scale versions of AES, *SR* and *SR** [8]. In most cases it is hard to solve the obtained equation system. Often it is even hard to estimate how long it will take to solve it. Therefore, smaller versions of a cipher with the same principles involved are designed. Trying to break those small-scale ciphers can give valuable clues about the complexity of solving the full-scale cipher.

Examples of successful algebraic cryptanalysis can be found in [14, 29].

Chapter 3

Computational Methods in Cryptanalysis

This chapter gives an introduction to different computational methods in cryptanalysis. A short outline of different techniques relevant to the central topic of the thesis is given. Except for *local search* only *exact* methods are presented, i.e., techniques which are guaranteed to give a result. Probabilistic methods are not considered.

3.1 Brute-Force Attack

One technique to reveal a key which works for most crypto systems is the *brute-force* attack. The attack simply tries out all possible keys to a given ciphertext and stops when intelligible plaintext is revealed. This method is simply to traverse the whole keyspace in search for the correct key. The practical problem is that the keyspace of a typical modern cipher is too big to be traversed. In order to break a cipher with a key length of n bits by brute force, and no other weakness like, e.g., an uneven distribution of keys or impossible keys which can be skipped during the search, one would have to try 2^{n-1} keys on average before the correct key is found. On any sequential computing device this would clearly take $\mathcal{O}(2^{n-1})$ steps and already be infeasible for typical values of n, like $n = 128$.

While older ciphers like the Data Encryption Standard with a key size of 56 bits can today be broken by successively trying each key on a portion of ciphertext [25], modern ciphers like the Advanced Encryption Standard, with a key length of 128 to 256 bits, are unlikely to be broken by brute-force.

In fact, the so-called *Landauer limit* can be utilized to conjecture what is possible to achieve by brute-force under the assumption of the classical paradigms of computation and physics [32].

Nevertheless, since there exist ciphers with moderate sized keys, and/or weaknesses which reduce the effective keyspace to a size that makes it a viable target for the brute-force attack, several improvements have been made. Here, notable approaches include utilizing highly specialized processors from other areas like computer graphics. Processors on graphics cards that usually do geometrical calculations are used to check a key on a given ciphertext. Circuits on graphics cards usually contain several thousand copies of a logical unit assigned a specific task in order to be highly parallel on its input data. If these units can be utilized to try keys of a cipher, they offer a much faster alternative to normal CPUs and can speed up the search massively, not only by pure parallelization but also by the relatively high speed of the logical units themselves [9, 57].

A similar approach is used by systems like COPACOBANA. Here *Field Programmable Arrays* (FPGAs) are used to create many specialized key testing instances on a micro chip [25]. FPGAs offer a cost-efficient way to realize such chips since they are designed as customizable prototyping or small series devices. Several FPGAs together then operate by the effect of parallelization and specialization of the circuits at a much higher speed than one or several all-purpose CPUs.

The most expensive approach in this direction are *application-specific integrated circuits* (ASIC). Micro chip design from scratch requires huge resources and great expertise and is unlikely to be used by for example criminal adversaries. To acheive maximum speed for testing keys, a specific chip which can be used in an attack is designed and manufactured.

Speed increases on this line of development can only be proportional to the speed increase of the technology used, and the number of copies made. This is the reason why only relatively small key sizes are possible to brute-force.

Since brute-force is the only attack which works on all encryption methods (except one-time pad) and therefore the most general attack, it is usually used as a benchmark. A cipher with a key length of n bits for which the fastest attack is brute-force, i.e., $\mathcal{O}(2^n)$, is called secure (with respect to its key length). Any other method which can find the key in an amount of steps of $\mathcal{O}(2^{\varepsilon n})$ with $\varepsilon < 1$ is said to break the cipher, since it is no longer necessary to search the whole key space for the correct key.

3.2 Algebraic Methods

In this section we describe the two major algebraic approaches for solving equation systems in cryptanalysis. In the following we consider

$$f_0(X_0) = 0, f_1(X_1) = 0, \ldots, f_{m-1}(X_{m-1}) = 0, \tag{3.1}$$

a set of equations as the input to algebraic methods.

There exist several specialized methods to extract secret information from equation systems originating from cryptographic primitives. Such methods are for example the *Cube Attack* [18] or the *Extended Linearization* [11] attack.

The techniques described here can be seen as *black-box* methods, i.e., general approaches for all kinds of non-linear equation systems and not limited to specific cryptographic primitives. These techniques have applications in all kinds of different areas in cryptography and cryptanalysis, and can be seen as the most general methods to solve a system (3.1).

We say that n is the number of variables and m is the number of equations. We further assume that we are working in $\mathbb{F}_2$, since it is the most common case in algebraic cryptanalysis or in general for applications arising from the computer sciences. There exists research which concentrates on overdefined systems [11], i.e., $m > n$ but we focus on the case $m = n$. Methods which work for $m = n$ are likely to work equally well or better for $m > n$.

3.2.1 Linearization

The most basic and probably best-known algorithm which can solve a *linear* system of equations is *Gaussian elimination*. It is the basic method from linear algebra to, e.g., find the rank of a matrix, and is named after Carl Friedrich Gauss.

By applying elementary row operations, a process called *forward elimination*, one brings the matrix of equation coefficients into what is called *triangular* or *row echelon* form. Then one can by *back substitution* find all solutions to the equation system. If the resulting *reduced* matrix has a rank lower than full rank, the system of equations has more than one solution.

The number of arithmetic operations to solve a linear system of equations by Gaussian elimination with $m = n$ is $\mathcal{O}(n^3)$ and the space requirement $\mathcal{O}(n^2)$. So the algorithm is clearly a member of the complexity class **P** and as such a feasible method for linear equation systems.

The problem arrises with the fact that all instances from algebraic cryptanalysis are non-linear and cannot be solved by Gaussian elimination alone.

Here, different strategies of *linearization* come into action. Monomials of degree greater or equal to 2 are substituted with a new variable and the resulting linear equation system is then solved by Gaussian elimination. A requirement for this method to succeed is that the number of linearly independent equations in the system is approximately the same as the number of different monomials.

Since this is not always the case, the *relinearization* technique was introduced [29]. Here, new non-linear equations are added to the linearized system and a second linearization is applied to introduce more linearly independent equations. The *XL* technique has the same goal [11, 12], but here the number of equations is increased by multiplying them with all monomials of a bounded degree.

The complexity of these methods is hard to estimate since they are very much dependent on the structure of the input systems, and at least in the case of XL further research has shown that its efficiency has been overestimated [45].

3.2.2 Gröbner Bases

Research in commutative algebra with motivation from algebraic geometry led to a greater interest in polynomials. In his doctoral thesis Bruno Buchberger developed the theory of *Gröbner bases* which he named after his supervisor, Wolfgang Gröbner. A Gröbner basis is a special finite generating subset of an ideal in a polynomial ring and it is unique if *reduced*. Furthermore, Buchberger developed a deterministic algorithm to compute such a generating set which is called *Buchberger's algorithm*. Overall, the theory of Gröbner bases can be seen as the non-linear generalization of Gaussian elimination in linear systems.

The main idea behind the approach is that the set of equations (3.1) as polynomials generates an ideal and there exists the so-called *elimination property* for the Gröbner basis.

Assume (3.1) has finitely many solutions. Then one consequence of the elimination property is that the Gröbner basis for polynomials

$$\{f_0(X_0), f_1(X_1), \ldots, f_{m-1}(X_{m-1})\} \tag{3.2}$$

contains some $g(x_{n-1})$ which depends on only one variable. Furthermore, there will be another polynomial $g'(x_{n-2}, x_{n-1})$ and so on [33]. With their help we can find the solutions to (3.1). In fact, if (3.1) has a unique solution we can form a linear equation system with the Gröbner basis for (3.2).

Computing Gröbner bases In order to compute a Gröbner basis from a set of polynomials (3.2) one has to define an *order* of monomials, also called a *term ordering*. Since we are dealing with multivariate non-linear polynomials it is necessary to define such an ordering in order to find the *leading term*. For the univariate case the usual convention is that $\alpha x^i > \beta x^j$ if $i > j$ and therefore αx^i would be leading βx^j in a polynomial. Since for the multivariate case it is not immediately clear which term is leading, e.g., $\alpha x^3 y^4$ or αz^2, such an order is established. A very common order would be for example *lexicographical*, and different term orderings have great influence on the complexity of computing the reduced Gröbner basis. Some polynomials are then minimal with respect to that ordering. Algorithms which seek to compute the Gröbner basis then iteratively try to modify the intermediate basis until, e.g., *Buchberger's criterion* is reached and the algorithm terminates. If that happens the Gröbner basis of the input is found, and it is guaranteed that such a basis exists [33].

Complexity & Variants The complexity of computing a Gröbner basis is hard to estimate. As previously mentioned it depends highly on the chosen term ordering. In the worst-case the complexity of the computation can be worse than $\mathcal{O}(2^n)$ and thus infeasible already for small n [11].

There are other methods which improve the running time of Buchberger's algorithm but are equally hard to estimate in time and space consumption. The most prominent examples here are Faugère's F4 and F5 algorithms [20, 21].

Furthermore, there exist several implementations of those algorithms and comparisons of those techniques to other methods in algebraic cryptanalysis, e.g., see [3, 4].

3.3 Boolean Constraint Propagation

In the last four decades *SAT* solvers have probably made the biggest advancements in solving huge instance of non-linear Boolean problems which were originally thought to be infeasible. Today they are used in many practical applications besides cryptanalysis.

In cryptanalysis they were for example successfully applied on reduced versions of the *Trivium* crypto system [36]. Further modified versions of a popular solver were developed which introduced the capability of dealing more efficiently with linearities [53]. There exist a big *SAT* solver community and hundreds of different implementations as well as a yearly *SAT* race (or challenge) to determine the current fastest solver implementation [2].

While there exist other solver fundaments than the *DPLL* algorithm, e.g., random walk based techniques [49], we will concentrate on the former due to its applications in cryptanalysis.

3.3.1 DPLL Family Solvers

Most of the successfully applied solvers today are derivations and extensions to the *Davis-Putnam-Logeman-Loveland algorithm* (*DPLL*) [15, 16], a complete backtracking algorithm which searches for a satisfying assignment of a Boolean formula in conjunctive normal form. It is based on the following observations on Boolean formulas in CNF.

1. **Pure literal.** If an unassigned literal l occurs only in one form, i.e., either in its pure form or as the negation of a variable, it can be assigned the value TRUE.

2. **Unit propagation.** Assume ϕ contains a clause with r literals

$$(l_0 \vee l_1 \vee \ldots \vee l_{r-1}),$$

   $l_0 = l_1 = \ldots = l_{r-2} = \text{FALSE}$ and only l_{r-1} is unassigned. We call such a clause *unit clause*. In order to satisfy ϕ we must set $l_{r-1} = \text{TRUE}$ and call this process *unit propagation*.

The most basic DPLL algorithm is then constructed by recursively applying these two rules. The algorithm takes as input a Boolean formula ϕ in CNF and returns TRUE or FALSE depending on wether there exists a satisfying assignment for ϕ.

Algorithm 1 Basic DPLL Algorithm

```
 1: procedure DPLL(φ)
 2:     if φ contains a satisfying assignment then
 3:         return TRUE
 4:     end if
 5:     if φ contains a clause C = FALSE then
 6:         return FALSE
 7:     end if
 8:     for every pure literal l do
 9:         φ ← φ_{l=1}
10:     end for
11:     for every unit clause C with unassigned literal r do
12:         φ ← φ_{r=1}
13:     end for
14:     l ← choose an unassigned literal in φ.          ▷ Guess/decision.
15:     return DPLL(φ_{l=1}) ∨ DPLL(φ_{l=0})         ▷ Splitting of the formula.
16: end procedure
```

Algorithm 1 has three states at which it can return. First, in case a satisfying assignment is found in line 3. Second, a *conflict* occurs, i.e., a clause becomes unsatisfied (it evaluates to FALSE in line 6) and third, the result of the recursive call for both assignments in line 15. Clearly, DPLL runs for a formula ϕ with n variables in at most $\mathcal{O}(2^n)$ steps. In order to terminate as fast as possible, it is important to reach one of the three points at which it can return as quickly as possible.

Furthermore, for this algorithm the satisfying assignment for ϕ (if it exists) can be easily derived by backtracking the steps of the algorithm.

Heuristics At line 14 in algorithm 1 we did not specify how the unassigned literal l is to be selected. This step is essentially a *guess*. In the *SAT* solving community it is also called a *decision step*. At this point it is of great importance to select the *optimal* literal l. With optimal we mean a literal that can lead us into one of the three possible return states as soon as possible. Essentially, for the basic DPLL algorithm we would like to reach two states in the algorithm:

I. We want to guess l correctly, i.e., ϕ has a satisfying assignment in which $l = \text{TRUE}$. (since this is the first assumption in which the algorithm branches at line 15).

II. If the latter is not possible (or very unlikely) we want to guess l so that it leads us to a conflict as soon as possible.

The reason for I. is that in line 15 we branch first into $\phi_{l=1}$. In order to reach line 3 recursively as fast as possible, we would like this guess to be correct so that we do not need to evaluate $\phi_{l=0}$. If this is not possible, we would like to produce short conflicts. In II. we try to minimize the steps or recursive calls of the algorithm until we can reach line 6.

These two objectives are obviously highly influenced by the selection of the *right* or *best* literal on line 14. This is the topic of the *decision heuristics* and there exists a great deal of research on this topic.

Early research suggested the use of *static* heuristics based on the frequency at which a literal occurs in ϕ, or additionally in relation to the clause size in which it occurs [26, 34]. Every literal is assigned a score based on this measure, and the literal with the highest score will be selected for the next decision. Those static heuristics do not take information which can be retrieved during run-time into account, e.g., how often a literal is involved in a conflict or how often it is assigned due to a unit clause.

Dynamic heuristics which incorporate such information, e.g., the *Variable State Independent Decaying Sum* (*VSIDS*), were found to be much more efficient [23, 37]. Unfortunately these complex heuristics can make the run-time behavior of a *SAT* solver hard to analyze.

Learning & Non-Chronological Backtracking The techniques of *learning* new clauses and to *jump back non-chronologically* are another important improvement for *SAT* solvers based on the DPLL algorithm [35, 37], and we will give an outline of the procedure in the following.

What a learning procedure essentially does is to analyze the reason behind a conflict and try to avoid it in future branches of the search by appending a new clause to ϕ. Introducing new guesses into ϕ usually yields several unit propagations which in turn may yield others. Assume that $d_0 = d_1 = \ldots = d_{r-1} = \text{TRUE}$ is a series of decisions that was introduced into ϕ. This caused several unit propagations by which the literals $p_0, p_1, \ldots, p_{s-1}$ got TRUE, but not necessarily in that order. Furthermore, ϕ for example contains the clause $(\overline{p_0} \vee \overline{p_1} \vee \overline{p_2})$. At this point ϕ is not satisfied, therefore at least one of the introduced guesses were wrong.

By an extension of the DPLL algorithm and by analyzing how the situation $p_0 = p_1 = p_2 = \text{TRUE}$ came to be, we can learn which subset of $d_0, d_1, \ldots, d_{r-1}$ is directly responsible for the conflict. This is usually a small subset of all the decisions made. In

order to avoid this subset of assignments in other branches of the search, a so called *conflict clause* C is introduced and the algorithm is continued on $\phi \wedge C$. Depending on which literals C contains, returning further than the last call in the recursion might be necessary. This is then so-called *non-chronological* backtracking which avoids other futile branches of the search tree.

In order to avoid running out of space during the calculation, learnt clauses have to be deleted from ϕ sporadically and there exists several different heuristics for this as well.

CNF Conversion The methods above work on CNF instances in the form (2.1). For the most applications arising from cryptanalysis it is necessary to convert equations in the form (3.1) to a set of clauses in CNF. There exist different approaches, e.g., [6], but it is not ultimately clear which one is the most efficient. Also, the conversion of instances (2.1) into another form should not affect the information the instance contains. Therefore the technique used for conversion should only result in small changes in the overall complexity of the problem.

3.4 Gluing & Agreeing

Efforts in the analysis of symmetric block ciphers, DES and AES, led to the development of the family of *Gluing and Agreeing* algorithms [42, 43]. These represent a new class of solving techniques that were primarily developed for problem instances from algebraic cryptanalysis.

For these techniques to work, equations must be *sparse*, i.e., each equation is only defined in a small number of variables.

Definition 5 (Symbol). *Let $f(x_0,x_1,\dots,x_{l-1}) = 0$ be an equation over some finite field $\mathbb{F}_q$. Every vector in variables $x_0,x_1,\dots,x_{l-1}$ which satisfies the equation is called a satisfying assignment. A symbol is then the pair (X,V), where $X = \{x_0,x_1,\dots,x_{l-1}\}$ is the ordered set of all variables and $V = \{\mathbf{v} \in \mathbb{F}_q^l \mid f(\mathbf{v}) = 0\}$ is the set of all satisfying assignments of $f(x_0,x_1,\dots,x_{l-1}) = 0$.*

From definition 5 it becomes clear that only an equation with a relatively small number of satisfying assignments can be efficiently represented as a symbol. Probably one of the most interesting applications of techniques described in this section, then, is on systems consisting of *sparse* equations. That is because the maximum number of satisfying assignments for a random equation over $\mathbb{F}_q$ is $q^l - 1$ and *large l* will make storing the equation as a symbol impractical. Specifically, when we speak of a l-sparse equation system we mean a system in which every equation contains at most l variables.

An in-depth analysis of the expected complexity of the algorithms by I. Semaev in [50] produced, among other results, the running times shown in figure 3.1, something that encouraged further research in the area.

3.4.1 Gluing

The *Gluing* procedure is the first important operation on a pair of symbols. It takes as an input two symbols $S_i = (X_i,V_i), S_j = (X_j,V_j)$ and outputs a new symbol containing

l	3	4	5	6
Gluing	1.262^n	1.355^n	1.425^n	1.479^n
Agr.-Gluing	1.113^n	1.205^n	1.276^n	1.334^n

Figure 3.1: Expected Complexities of Agreeing/Agreeing-Gluing.

all common solutions of the input symbols.

In the following, $X_{ij} = X_i \cap X_j$ and $\mathbf{v}[X]$ denotes the projection of vector $\mathbf{v}$ on to the variables X. Furthermore, $\mathbf{v} \circ \mathbf{w}$ denotes the concatenation of vectors $\mathbf{v}$ and $\mathbf{w}$ *with respect to overlap*. By that we mean the combination of two vectors into a new one, but only if common variables have equal values. Thus, for two vectors $\mathbf{v}, \mathbf{w}$ with common variables X, this operation is only possible if $\mathbf{v}[X] = \mathbf{w}[X]$. In case $\mathbf{v}[X] \neq \mathbf{w}[X]$ the result is *empty*.

To represent all common solutions the new symbol $S_r = (X_r, V_r)$ must be defined in the union of the symbol variables, i.e., $X_r = X_i \cup X_j$. The satisfying assignments to both input symbols can be then computed as follows:

$$V_r = \{\mathbf{v} \circ \mathbf{w} \mid \mathbf{v} \in V_i, \mathbf{w} \in V_j\}. \tag{3.3}$$

After this computation for every $\mathbf{u} \in V_r$ it will be true that $\mathbf{u}[X_i] \in V_i$ and $\mathbf{u}[X_j] \in V_j$. Then V_r contains all vectors which satisfy both equations simultaneously.

This Gluing of two symbols is also denoted as $S_i \circ S_j$ since it can be interpreted as a *concatenation* of the solutions to the individual symbols while eliminating solutions which do not satisfy both.

If one is given a set of symbols and one would like to find a solution which satisfies all symbols at once, i.e., find a common solution, one possibility would be to glue all symbols together. In fact, if given symbols $S_0, S_1, \ldots, S_{m-1}$ representing an equation system, and applying repeated gluing, i.e.,

$$S = S_0 \circ S_1 \circ \ldots \circ S_{m-1},$$

then S will contain all solutions that satisfy the equation system. Thus gluing all symbols together is one way to find a solution to an equation system (3.1). If the intermediate symbol S_r gets *empty* during the process of gluing all symbols together, i.e., $V_r = \emptyset$, we know that the system is inconsistent and it has no solution.

Worst-case complexity Computing the set (3.3) is the predominant contributor to the complexity of the Gluing operation. In order to calculate V_r one has to compare all pairs of vectors $\mathbf{v} \in V_i, \mathbf{w} \in V_j$ with each other and write their concatenation into V_r in case their subvectors in common variables are equal.

While projecting vectors and concatenating them can be neglected as linear factors in the estimation of the complexity, the comparison of all vector pairs yields a time complexity of $\mathcal{O}(|V_i| \cdot |V_j|)$. Likewise, the space complexity of the result is $\mathcal{O}(|V_i| \cdot |V_j|)$, since the result which has to be stored can at most be of that size.

Example 3 (Gluing). Suppose the following two symbols S_0 and S_1 in (3.4) are given. Symbol S_0 is defined in variables $X_0 = \{x_0, x_1, x_2\}$ and has the (horizontally written)

satisfying assignments $V_0 = \{a_0, a_1, a_2, a_3\}$. Likewise, symbol S_1 is defined in variables $X_1 = \{x_2, x_3, x_4\}$ and contains the satisfying assignments $V_1 = \{b_0, b_1, b_2\}$.

S_0	x_0	x_1	x_2
a_0	0	0	0
a_1	0	1	0
a_2	1	0	0
a_3	1	1	1

,

S_1	x_2	x_3	x_4
b_0	1	0	0
b_1	1	0	1
b_2	1	1	1

(3.4)

In order to calculate the gluing $S_0 \circ S_1$ we need to calculate the set of pairs of vectors which we can concatenate. In this case only a_3 of V_0 can be combined with any of the vectors in V_1. More specifically, the resulting symbol of the gluing operation in this case will be $(X_0 \cup X_1, \{a_3 \circ b_0, a_3 \circ b_1, a_3 \circ b_2\})$ as shown in (3.5).

$S_0 \circ S_1$	x_0	x_1	x_2	x_3	x_4
c_0	1	1	1	0	0
c_1	1	1	1	0	1
c_2	1	1	1	1	1

(3.5)

3.4.2 Agreeing

The *Agreeing* procedure takes as input two symbols and deletes all assignments whose projection on common variables do not appear in both symbols. If no assignments can be deleted, the symbols are said to be *pair-wise agreeing*. If this property is true for a whole set of symbols, the system is said to be pair-wise agreeing.

Algorithm 2 Agreeing Procedure

```
1:  procedure AGREE((X_i, V_i), (X_j, V_j))
2:      X_ij ← X_i ∩ X_j
3:      for v ∈ V_i do
4:          if ∀w ∈ V_j : v[X_ij] ≠ w[X_ij] then
5:              Delete v from V_i
6:          end if
7:      end for
8:      for v ∈ V_j do
9:          if ∀w ∈ V_i : v[X_ij] ≠ w[X_ij] then
10:             Delete v from V_j
11:         end if
12:     end for
13: end procedure
```

Algorithm 2 takes as input two symbols and modifies them until all pairs are pair-wise agreeing. Preprocessing the symbols and translating the relations of the subvectors of the assignments into *tuples* [50] or *pockets* [47] allows for a faster Agreeing procedure, but the output is equivalent to that of algorithm 2.

Running the `Agree` procedure on a whole equation system for all pairs *propagates knowledge* by deleting assignments throughout the equation system. This is done until no further changes to the symbols occur and all pairs are in an agreeing state.

By successively deleting assignments from symbols one arrives at a unique solution for the equation system if it holds for all symbols S_i that $|V_i| = 1$ and they are pair-wise agreeing. The gluing of all those symbols then naturally contains a single assignment which satisfies all symbols simultaneously.

Worst-case complexity The complexity of algorithm 2 is clearly dominated by the two **for**-loops in line 3-7 and 8-12. Here again the time complexity mostly depends on the number of assignments per symbol and is in the range of $\mathcal{O}(|V_i| \cdot |V_j|)$. The space complexity on the other hand is constant to the size of the input since only the existing symbols are modified.

Guessing Unlike what happens in the Gluing procedure, applying Agreeing alone does not solve an equation system. In fact, most equation systems from ciphers are pair-wise agreeing when they are constructed.

In order to solve a system by Agreeing one has to introduce *guesses*. Such guesses are deletions of specific assignments of a symbol and then running Agreeing in order to see if any new knowledge is propagated. If a symbol becomes empty during agreeing it is clear that the introduced guess was wrong and one has to backtrack. If all $|V_i| = 1$, we know that we have found a solution to the system. Different guessing heuristics are possible here, too.

Example 4 (Agreeing). Consider symbols (3.4) as input to algorithm 2. On line 2 the algorithm computes $X_{ij} = \{x_2\}$. The following **for**-loop deletes all vectors $\mathbf{v}$ from S_0 for which $\mathbf{v}[\{x_2\}] = 0$ since $\mathbf{w}[\{x_2\}] = 1$ for all vectors $\mathbf{w}$ in symbol S_1. The following **for**-loop does nothing since both symbols are already pair-wise agreeing. The result of the computation is therefore:

S_0	x_0	x_1	x_2
a_3	1	1	1

,

S_1	x_2	x_3	x_4
b_0	1	0	0
b_1	1	0	1
b_2	1	1	1

.

3.4.3 MRHS

The analysis of block ciphers, especially the analysis of DES and AES, gave rise to the technique of *Multiple Right Hand Side equation systems*, or short *MRHS* [42, 44] due to I. Semaev. A MRHS equation extends linear equation systems in such a way that it can have multiple right hand sides. Such an MRHS equation is written as

$$A\mathbf{x} = L$$

where A is a matrix with n columns and l rows. Similarly, L is a matrix with r columns and l rows. Every column vector $\mathbf{l}$ in L then forms a linear equation system

$$A\mathbf{x} = \mathbf{l}$$

which can be easily solved for variables in $\mathbf{x}$, e.g., by Gauss' algorithm. Every solution for $\mathbf{x}$ using any column $\mathbf{l}$ in L is a solution to the MRHS equation.

Equations of this type are similar to a symbol as they contain a set of assignments. Here, as opposed to the symbol representation, an assignment (or column) in L does not directly encode a solution to the equation, but provides solutions to a linear equation system. As such they give an advantage for equations and equation systems containing much inherent linearity, e.g., equation systems representing cryptographic primitives.

Similar to symbols one can by a list of MRHS equations

$$A_0\mathbf{x} = L_0, A_1\mathbf{x} = L_1, \ldots, A_{m-1}\mathbf{x} = L_{m-1}$$

express a system of equations and if the Hamming weight of each row in A_i is equal to 1 the representation is equivalent to the symbol representation. That is, every L_i is a list of satisfying assignments for linear combinations containing only one variable.

MRHS Gluing & Agreeing Similarly to the way symbols are represented one can define the Gluing and Agreeing procedure for MRHS equations. Instead of working with direct assignments of variables one has to take care of linear combinations.

In order to glue $A_0\mathbf{x} = L_0$ and $A_1\mathbf{x} = L_1$ together one has to augment the first equation with the second:

$$\begin{bmatrix} A_0 \\ A_1 \end{bmatrix} \mathbf{x} = \begin{bmatrix} L_0 \\ \mathbf{0} \end{bmatrix} \diamond \begin{bmatrix} \mathbf{0} \\ L_1 \end{bmatrix}. \tag{3.6}$$

Here $\mathbf{0}$ stands for an all zero (sub-)matrix. The operation $M \diamond N$ on two matrices denotes the addition modulo q of all possible pairs of columns, one from N and one from M, to form a new matrix. Then one has to bring $\begin{bmatrix} A_0 \\ A_1 \end{bmatrix}$ into upper triangle form. This is done while applying the same operations on both matrices on the right hand side, just as for an ordinary linear equation system, but now for all columns on the right hand side simultaneously.

If the left hand side matrix has full rank, all pairs of right hand sides can be combined together (*XORed*). However, it is possible that the left hand side has less than full rank and so contains zero rows after triangularizing. Then only columns (*assignments*) from the two right hand sides can be combined, if their sum is 0 where the left hand side has zero rows. Otherwise the subsystem resulting from the sum of the right hand side columns would be inconsistent.

The Agreeing procedure on MRHS equations seeks to delete right hand sides which cannot be combined.

Proofs the correctness, further analysis and examples of this procedures can be found in [42, 44].

Example 5 (MRHS Gluing). Suppose the MRHS equations

$$\begin{bmatrix} 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

and

$$\begin{bmatrix} 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

are given.

In order to glue them one has to to augment the first with the second one. That is

$$\left[\begin{array}{ccccc} 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ \hline 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 \end{array}\right] \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \left[\begin{array}{cccc} 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ \hline 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array}\right] \diamond \left[\begin{array}{ccc} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ \hline 0 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \end{array}\right]$$

By row operations we can find that on the left hand side row 6 is the sum of row $1,3$ and 4 and therefore the left hand side does not have full rank (note the modified right hand side):

$$\left[\begin{array}{ccccc} 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ \hline 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{array}\right] \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \left[\begin{array}{cccc} 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ \hline 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{array}\right] \diamond \left[\begin{array}{ccc} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ \hline 0 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \end{array}\right].$$

That tells us that we can only combine columns $1,2$ and 4 from the first equation with column 3 from the second. Column 3 from the first equation on the other hand can only be combined with column 1 and 2 from the second equation. Any other combination would yield a 1 in the zero row on the left hand side and therefore an inconsistent equation system.

Thus the gluing of the initial equations with the all 0-row removed is:

$$\begin{bmatrix} 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \end{bmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{bmatrix} 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 \end{bmatrix}.$$

3.5 Other Methods

Syllogism In [58] A. Zakrevskij and I. Vasilkova describe another method for dealing with large systems of Boolean polynomial equations.

Assume that (3.1) is given as a set of symbols $(X_0, V_0), (X_1, V_1), \ldots, (X_{m-1}, V_{m-1})$. We say that a symbol (X_i, V_i) has a *2-ban* if there exists $a, b \in \{0, 1\}$ such that for all

$$\mathbf{v} = (x_0, \ldots, x_r, \ldots, x_s, \ldots, x_{l-1}) \in V_i$$

it is true that

$$(x_r + a)(x_s + b) = 0. \tag{3.7}$$

In other words, a specific value of projection onto variables x_r and x_s does not exist in V_i, therefore this projection is banned. Equation (3.7) can also be written as $x_r^{(a)} \vee x_s^{(b)}$, where $x^{(0)} = \bar{x}$ and $x^{(1)} = x$. Both representations are TRUE under the same assignments for x_r, x_s, a and b.

That information can in itself be propagated to other symbols, e.g., if symbol (X_i, V_i) yields constraint $x_r^{(1)} \vee x_s^{(1)}$ we know that there exists no vector $\mathbf{v} \in V_i$ such that $\mathbf{v}[\{x_r, x_s\}] = (1,1)$. What follows from this is that no other symbol can contain any vector $(1,1)$ if projected onto x_r, x_s. Deleting those vectors is essentially what the Agreeing method does, here just limited on vectors in 2 variables.

There is another representation of such a 2-ban with implications wich makes the Zakrevskij method different. We can also write (3.7) as

$$x_r^{(a+1)} \Rightarrow x_s^{(b)} \text{ or } x_s^{(b+1)} \Rightarrow x_r^{(a)}. \tag{3.8}$$

The implications in (3.8) contain the same information in the sense that they both remove the same vector.

From Boolean algebra we know, that if implications $x \Rightarrow y$ and $y \Rightarrow z$ are TRUE then $x \Rightarrow z$ has to be TRUE, too. This method of deriving new information is also called *syllogism*. We can use this fact to create new 2-bans. In case we have two 2-bans, $(x_r + a)(x_s + b) = 0$ and $(x_s + b + 1)(x_t + c) = 0$ we can derive that for all vectors in the equation system

$$(x_r + a)(x_t + c) = 0.$$

By applying this method repeatedly one can compute the *transitive closure* of all implications. Then all information which can be derived using only 2-bans is propagated and can be done in $\mathcal{O}(n^3)$ [56].

Applying this method alone does not solve the equation system and so, e.g., guesses have to be introduced in order to find a solution to the system.

Local Search There exist different definitions of the concept of local search and as it is the only not necessarily exact method that is presented here, only the general idea is outlined. It is a popular concept in *Constraint Programming*, and frequently used for different industrial applications [55].

In order to initiate a local search, one guesses or chooses (by some heuristic) an assignment to the input instance. The input instance can be from any computational problem in which an assignment and a solution can be identified. Examples are the satisfiability of a Boolean formula or the shortest path between two points in a graph.

The chances that this assignment is the solution is low for most applications. However, it is usually possible to identify some modification in which this intermediate assignment can be *improved*. This modification is then applied and the process repeated to the new intermediate assignment. In doing so, the initial assignment is modified iteratively until a solution is found. Usually some form of randomization is introduced in order to avoid endless-loops.

Unlike the other methods, and depending on the implementation, a termination of the algorithm is not guaranteed. The method which decides the modification to the

intermediate assignment is crucial for the run-time, and there exist huge differences between different applications. Therefore it is hard to determine the run-time for cryptanalytic applications.

Our own experiments have shown that instances from cryptographic problems have too many *local minima*, i.e., intermediate assignments for which it is *unknown* how to improve them.

Chapter 4

Introduction to the Papers

Paper I: *Phase Transition in a System of Random Sparse Boolean Equations*

In this paper we study two algorithms that can be used to solve Boolean random sparse equation systems. The Agreeing algorithm and the Syllogism method are compared. We show that a phase transition occurs and give threshold values for when it can be expected to occur.

The work was done in collaboration with Pavol Zajac from the Slovak University of Technology in Bratislava. It was presented in August 2010 at the NIL-0004 project workshop in Smolenice, Slovakia.

Paper II: *Solving Equation Systems by Agreeing and Learning*

The main contribution of the paper is a learning strategy for the Agreeing algorithm. Furthermore, a technique for non-chronological backtracking is introduced. Experimental data on the extended algorithm on Boolean random sparse equation is presented.

The work on the paper was done in collaboration with Håvard Raddum. It was presented in June 2010 at the International Workshop on the Arithmetic of Finite Fields in Istanbul, Turkey.

Paper III: *Analysis of Trivium Using Compressed Right Hand Side Equations*

The paper introduces a new representation of non-linear multivariate equations. Then the new representation is applied to an equation system yielded by the cipher Trivium. It enables us to represent Trivium as one single equation in a manageable size and processing time. Furthermore, we introduce a small-scale variant of Trivium for further experimentation.

The work on the paper was done in collaboration with Håvard Raddum. It was presented in November 2011 at the 14th Annual International Conference on Information Security and Cryptology in Seoul, Korea.

Paper IV: *Solving Compressed Right Hand Side Equation Systems with Linear Absorption*

This paper builds heavily on the contribution of the predecessor. The single compressed right hand side equation from the previous paper does not immediately yield a solution to the equation system. In order to extract a solution, we have to introduce a new operation on binary decision diagrams which enables us to get to the solution. We then successfully apply it to reduced versions of Trivium.

The work on the paper was done in collaboration with Håvard Raddum. It will be presented in June 2012 at SETA 2012: SEquences and Their Applications in Waterloo, Canada.

Chapter 5

Scientific Results

Paper I

5.1 Phase Transition in a System of Random Sparse Boolean Equations

Thorsten E. Schilling, Pavol Zajac
Tatra Mountains Mathematical Publications **45**, 1–13, (2010)

Phase Transition in a System of Random Sparse Boolean Equations

Thorsten Ernst Schilling and Pavol Zajac

University of Bergen, KAIVT FEI STU Bratislava
thorsten.schilling@ii.uib.no, pavol.zajac@stuba.sk

Abstract. Many problems, including algebraic cryptanalysis, can be transformed to a problem of solving a (large) system of sparse Boolean equations. In this article we study 2 algorithms that can be used to remove some redundancy from such a system: Agreeing, and Syllogism method. Combined with appropriate guessing strategies, these methods can be used to solve the whole system of equations. We show that a phase transition occurs in the initial reduction of the randomly generated system of equations. When the number of (partial) solutions in each equation of the system is binomially distributed with probability of partial solution p, the number of partial solutions remaining after the initial reduction is very low for p's below some threshold p_t, on the other hand for $p > p_t$ the reduction only occurs with a quickly diminishing probability.

Key words: Algebraic cryptanalysis, Agreeing, Boolean equations, SAT problem.

1 Introduction

Given an equation system (1) over a finite field $\mathbb{F}_q$ it is a well known NP-complete problem to determine a common solution to all equations. Finding a solution to such an equation system can be interesting in algebraic cryptanalysis, e.g. when the solution to the equation system is a constraint to a used, unknown key.

Experiments with different solving algorithms suggest that during the solving the number of possible solutions is not decreasing *continously*. That means that during the solving process the overall number of solutions does not decrease constantly, but that at some point the number of possible solutions decreases rapidly.

In this paper we try to determine this point of *phase transition* in order to get a better measure for the hardness of a given problem.

The paper is organized as follows. In Section 2 we explain the basic representation of equations and the idea how the number of potential solutions to the equation system can be reduced. Section 3 explains the Agreeing algorithm and the reduction by Agreeing. Section 4 explains the reduction technique by syllogisms. In Section 5 we make a direct comparison of these both techniques. Section 6 shows our experimental results on a series of random sample instances and Section 7 concludes the paper.

2 Representation of the system of sparse Boolean equations and its reduction

Let $X = \{x_1, x_2, \ldots, x_n\}$ be a set of variables (unknowns), and let $X_i \subset X$ for $i = 1, ..m$, such that $|X_i| = l$. We consider X_i to be chosen uniformly at random from all possible l-subsets of X. Let $\mathcal{F}$ be a system of Boolean equations

$$f_1(X_1) = 0, ...f_m(X_m) = 0, \tag{1}$$

such that f_i depends only on variables from the set X_i. Let V_i be a set of vectors that are projections of solutions of $f_i(X) = 0$ into variables of X_i. We call (X_i, V_i) a symbol, and we say that the symbol represents the equation $f_i(X_i) = 0$. We call vectors of V_i partial solutions of the system.

To compute all solutions of the whole system we can apply the so called Gluing procedure [2]. The procedure is as follows: We merge two symbols (X_i, V_i), (X_j, V_j) together and enumerate all possible solutions V_{ij} of a new symbol $(X_i \cup X_j, V_{ij})$. Then we replace the original two symbols with a new one. Until some point the total number of solutions grows (very quickly). Gluing new symbols together removes some of the partial solutions, until only the valid solutions of the system remain. More advanced algorithms based on Gluing use different Gluing strategies, and strategies for removal excess solutions before/without Gluing, and some combinations with guessing variable values or solutions of individual equations. The fastest algorithm based on Gluing up to date is the Improved Agreeing-Gluing Algorithm introduced in [5].

In this article we want to focus on methods that do not use Gluing or any guessing. Consider the situation where V_i contains just one solution. We know immediately the values of l variables. Thus these values can be substituted into all other equations, and conflicting partial solutions get removed. Solutions from the set V_j can be removed if it shares some variables with V_i. If the remaining number of possible solutions in V_j is small, we can find new "fixed" values of variables, and spread this information, until (almost) all variables have fixed values. This technique is also called the Spreading of constants [9]. A more advanced version, the local reduction technique [9], uses fixed /resp. forbidden/ solutions for groups of variables. The similar method, although differently formulated is the Agreeing method. Agreeing uses a more efficient representation, and can be extended to more efficient variants [3, 4]. A different reduction method based on Syllogism rule (transitiveness of the implication relation) was also presented in [9], and was later adapted to the symbol representation [7].

We investigate the behavior of the reduction methods in a random sparse Boolean equation system as a function of one additional parameter: The probability of a partial solution p. We do not explicitly write down the closed form for f_i, instead we generate each symbol in a stochastic manner. We want to investigate the systems that have at least one solution, so we generate first a random solution $\mathbf{x}$. Then for each symbol we generate the set V_i in such a way that the probability of $v \in V_i$ is 1, if v is a projection of $\mathbf{x}$ to X_i, and p otherwise. The number of solutions in each symbol is then binomially distributed $|V_i| \sim Bi(2^l, p)$.

We call the variable x_j fixed, if the projection of all $v \in V_i$ to x_j in some equation (X_i, V_i), with $x_j \in X_i$ contains only one value, either 0 or 1. The system is solved by an algorithm A, if all variables are fixed after the application of the algorithm A. To investigate various algorithms we run the following experiment:

1. Given the set of parameters (m, n, l, p)[1], generate a set of N random equation systems (as defined above).
2. For each system, apply the reduction algorithm A.
3. Compute the fixation ratio $r = f/n$, where f is the number of fixed variables (after the application of A).
4. Compute the average fixation ratio $\hat{r} = 1/N \sum r$ for the whole set of experiments.

If the average fixation ratio stays near 0, then we didn't learn any significant information about the solution of the system by the application of the algorithm A. To solve the system, we must either use a different algorithm, or reduce the system by guessing some solutions. The basic guessing is exponential in nature, thus the system (in our settings) need an exponential time to solve. On the other hand, if the average fixation ratio is near 1, we have a high chance to solve the whole (randomly generated) system just by applying A. In this case, if the runtime of algorithm A is bounded in polynomial time, we can say that the average instance of the problem (m, n, l, p) is solvable in polynomial time.

3 Reduction by Agreeing

In order to find a solution to a set of symbols the Agreeing algorithm attempts to delete vectors from symbols S_i which cannot be part of a common solution. In the following, the projection of a vector v_k on variables X is denoted by $v_k[X]$ and $V[X]$ denotes the set of projections of all vectors $v_k \in V$ on variables X.

Given two symbols $S_i = (X_i, V_i)$ and $S_j = (X_j, V_j)$ with $i \neq j$ we say that S_i and S_j are in a non-agreeing state if there exists at least one vector $a_p \in V_i$ such that $a_p[X_i \cap X_j] \notin V_j[X_i \cap X_j]$. If there exists a solution to the system, each symbol will contain one vector that matches the global solution. The vector a_p cannot be combined with any of the possible assignments in symbol S_j, hence it cannot be part of a solution to the whole system and can be deleted. The deletion of all vectors $a_p \in V_i$ and $b_q \in V_j$ which are incompatible with all vectors in V_j and V_i, respectively, is called agreeing. If by agreeing the set of vectors of a symbol gets empty, there exists no solution to the equation system. The agreeing of all pairs of symbols in a set of symbols $\mathcal{S} = \{S_0, \dots, S_{m-1}\}$ until no further deletion of vectors can be done is called the Agreeing algorithm.

After running Algorithm 1 on $\mathcal{S}$ we call $\mathcal{S}$ *pair-wise agreed.* On the average all $S_i \in \mathcal{S}$ have exactly one one vector left. The solution to the system is then the gluing of the remaining vectors and the system can be regarded as *solved.* If on the other hand one or more symbols get *empty*, i.e. $V_i = \emptyset$, the system has no common solution.

[1] We use $m = n$, as this is the most important situation.

Algorithm 1 Agreeing Algorithm

```
1: procedure AGREE(S)
2:     while (X_i, V_i), (X_j, V_j) ∈ S which do not agree do
3:         Y ← X_i ∩ X_j
4:         Delete all a_p ∈ V_i for which a_p[Y] ∉ V_j[Y]
5:         Delete all a_q ∈ V_j for which a_q[Y] ∉ V_i[Y]
6:     end while
7: end procedure
```

Example 1 (Agreeing). The following pair of symbols is in a non-agreeing state:

S_0	0	1	2
a_0	0	0	0
a_1	0	0	1
a_2	0	1	0
a_3	1	1	1

S_1	0	1	3
b_0	0	0	0
b_1	1	0	1

.

The vectors a_2, a_3 differ from each b_j in their projection on common variables x_0, x_1 and can be deleted. Likewise, b_1 cannot be combined with any of the a_i and can also be deleted. After agreeing the symbols become:

S_0	0	1	2
a_0	0	0	0
a_1	0	0	1

S_1	0	1	3
b_0	0	0	0

.

Guessing and Agreeing In a usual setting, e.g. given as an input equation systems from ciphers, Agreeing does not yield a solution immediately. The algorithm has to be modified in a way that one has to introduce *guesses*.

4 Reduction by Syllogisms

Let (X, V) be an equation, $x_i, x_j \in X$. Let us have two constants $a, b \in \mathbf{F}_2$ such that for each $\mathbf{v} = (x_{i_1}, \ldots, x_i, x_j, \ldots, x_{i_l}) \in V : (x_i + a)(x_j + b) = 0$. We say that equation (X, V) is constrained by $(x_1 + a)(x_2 + b) = 0$, or that $(x_1 + a)(x_2 + b) = 0$ is a 2-constraint for the equation (X, V). A solution $\mathbf{x}$ of the whole system $\mathcal{F}$ projected to variables X_i must also be a partial solution in V_i. Thus $\mathbf{x}$ is constrained by every 2-constraint we place on each of the equations in the system. Thus we can apply 2-constraints found in (X_i, V_i) to remove those partial solutions of (X_j, V_j), that violate some of the 2-constraints. This is the basis of the syllogism reduction technique, that is similar to Agreeing. The main difference is the addition of creating new 2-constraints by the syllogism rule (see below).

We can see each 2-constraint $(x_i + a)(x_j + b) = 0$ as one clause of type $x_i^{(a)} \vee x_j^{(b)}$, , where $x^{(0)} = \overline{x}$ (negation of x), and $x^{(1)} = x$. All such clauses must be satisfied by the solution of the system. However, if some vector $\mathbf{y}$ satisfies all

such clauses, it does not automatically mean it is a solution of the system[2]. To check whether the set of 2-constraints written in a form of clauses is satisfiable is the well known 2-SAT problem. We must note, that we are not solving the 2-SAT problem, if we already know that the solution exists. However, if the system contains a large set of 2-constraints, we expect that if we remove the correct solution the system becomes unsatisfiable. Then we expect to be able to remove almost all invalid solutions from the system using just the 2-constraints.

We can also rewrite the 2-constraint in the form of two (equivalent) implications: $x_i^{(a+1)} \Rightarrow x_j^{(b)}$, and $x_i^{(b+1)} \Rightarrow x_j^{(a)}$. Implication is a transitive relation, i.e. if $x \Rightarrow y$ and $y \Rightarrow z$, it follows that $x \Rightarrow z$. This derivation is also called the syllogism rule. Thus, if we have two 2-constraints $(x_i + a)(x_j + b) = 0$, $(x_j + b + 1)(x_k + c) = 0$, we can derive a new 2-constraint $(x_i + a)(x_k + c) = 0$. The new 2-constraints then can be used to remove additional partial solutions from the system. It is also possible to derive special 2-constraints in the form $(x_k + a)(x_k + a) = 0$, which simply means that $x_k = a$, and thus x_k is fixed.

A set of 2-constraints is transitively closed, if we cannot derive any more 2-constraints using the transitiveness property of the underlying implications. A transitively closed set of 2-constraints thus contain the maximum of information we can get from the system (using just 2-constraints). We represent a set of 2-constraints in a form of the implication graph. Vertices of the graph are labelled by $\{x_1, x_2, \ldots, x_n, \overline{x}_1, \overline{x}_2, \ldots, \overline{x}_n\}$. Edge (x, y) exists if there is an implication $x \Rightarrow y$ (so a single 2-constraint is always represented by 2 edges). To find the transitively closed set of 2-constraints, we compute the transitive closure of the implication graph (by some of the known algorithms).

The Syllogism reduction method thus works as follows:

1. Examine the set of equations, and find all 2-constraints.
2. For each 2-constraint, add corresponding implications to the implication graph.
3. Compute the transitive closure of the implication graph.
4. Apply all 2-constraints back to the set of equations, i.e. remove all solutions from each V_i that violate any of the 2-constraints stored in the implication graph.
5. If some solutions were removed, repeat the algorithm, otherwise output the reduced system.

The transitive closure of an implication graph can be computed e.g. by Warshall's algorithm [6] in $O(n^3)$. After each repetition of the transitive closure algorithm, we must remove add at least one partial solution, otherwise the method stops. Thus the worst case complexity is upper bounded in $O(Mn^3)$, where $M \approx mp2^l$ is the initial number of solution. Actually the number of repetitions of the algorithm is very small in practice, especially if the system cannot be reduced (usually just one repetition). However, we need an additional $O(n^2)$

[2] It is only true, if we the set of 2-constraints is tight, i.e. for each equation (X, V) we can find such a set of 2-constraints, that no other assignment of variables is permissible except those in V.

memory storage for the implication graph. A more detailed analysis is provided in [8].

The original method presented in [9] uses immediate resolution of transitive closure after adding each new 2-constraint (also the implication graph is represented differently), but the algorithm gives the same results (although the running times differ, but these depend also on the implementation, and the platform used, respectively). Experimental results in [9] also show that a phase transition effect exists, but no theoretical explanation or expected parameters are provided.

4.1 The heuristic model for the expected behavior

The phase transition in the syllogism method can be connected to the corresponding representation of the problem in CNF clauses $x_i^{(a)} \vee x_j^{(b)}$. Each of these clauses must be satisfied simultaneously, so we get a 2-SAT problem instance in n variables with k clauses, where k is the total number of clauses (2-constraints) in the system. It was shown in [1] that if we have a random 2-SAT problem with k clauses in n variables, having $k/n = \alpha$ fixed as $n \to \infty$, then for $\alpha > 1$ almost every formula is unsatisfiable, and for $\alpha < 1$ almost all formulas can be satisfied. To use this result for the syllogism method, we must first estimate the number of constraints in the system.

Lemma 1. *Let $\mathcal{S} = (X, V)$ be a randomly chosen symbol with $l = |X| \geq 2$ active variables, and $s = |V|$ distinct solutions. Let $p_{s,l}$ denote a probability, that a randomly chosen constraint $(x_i + a)(x_j + b) = 0$, $x_i, x_j \in X$, $a, b \in \{0, 1\}$ holds for an equation defined by symbol $\mathcal{S}$. Then*

$$p_{s,l} = \prod_{i=0}^{s-1} \frac{3 \cdot 2^{l-2} - i}{2^l - i} \tag{2}$$

Proof. There are 2^l possible solutions. For $s = 1$, there are 2^{l-2} solutions for which the constraint $(x_i + a)(x_j + b) = 0$ does not hold, namely those where $x_i = a+1$ and $x_j = b+1$. For all other $3 \cdot 2^{v-2}$ solutions the constraint holds, so the probability $p_{1,l} = \frac{3 \cdot 2^{l-2}}{2^l} = 3/4$. If we have already i constrained solutions, we can choose the next constrained solution from only $3 \cdot 2^{l-2} - i$ vectors out of $2^l - i$, thus $p_{i+1,l} = p_{i,l} \frac{3 \cdot 2^{l-2} - i}{2^l - i}$. By expanding this recursion we get equation (2).

Using $p_{s,l}$ from equation (2), we can compute the probability of a constrained solution in a symbol from system generated with the binomial distribution:

$$P_{l,p} = \sum_{s=0}^{2^l} \binom{2^l}{s} p^s (1-p)^{2^l - s} p_{s,l}. \tag{3}$$

The expected number of constraints in an equation is $\alpha(l, p) = 4\binom{l}{2} P_{l,p}$. The total number of expected constraints is $k = \alpha m$. We do not take into account

the constraints found by the syllogism rule. The phase transition point should be near the value p_t for which $k/n = 1$. For our experiments $m = n$, thus we are looking for p_t for which $\alpha(l, p_t) = 1$. If $p > p_t$ we get $\alpha(p, l) < 1$, thus the corresponding 2-SAT problem is very likely satisfiable, and the syllogism method cannot eliminate much solutions. If $p < p_t$, $\alpha(p, l) > 1$, and the corresponding 2-SAT problem is very likely unsatisfiable. Then almost all excess solutions get removed by 2-constraints during the application of the syllogism method. The expected phase transition probabilities are summarized in Table 1.

l	p_t	$p_t \cdot 2^l$
5	0.3694	11.8
6	0.2258	14.5
7	0.1293	16.6
8	0.0711	18.2
9	0.0381	19.5
10	0.0201	20.6

Table 1. Probabilities p_t at which the phase transition in syllogism method is expected to occur.

5 Qualitative comparison of the methods

There exists a set of equations with all partial solutions in Agreeing state, that can be reduced by the Syllogism method. One of the examples is presented in Table 1. In the example, we get constraints between variables 1, 2 ($x_2 \Rightarrow x_1$), variables 2, 3 ($x_3 \Rightarrow x_2$), but originally no constraint between variables 1, 3. A new constraint ($x_3 \Rightarrow x_1$) can be derived using the transitive closure. This new constraint removes one partial solution ($x_1 = 0, x_3 = 1, x_6 = 1$), and furthermore allows us to find a fixed solution $x_6 = 0$. We remark that the same effect is obtained, if we glue two of the equations together, and agree them with the third equation. It is thus possible, that the syllogism method can reduce the system that the agreeing method is unable to.

S_0	1 2 4
a_0	0 0 0
a_1	0 0 1
a_2	1 0 1
a_3	1 1 1

S_1	2 3 5
b_0	0 0 1
b_1	1 0 0
b_2	1 0 1
b_3	1 1 1

S_2	1 3 6
c_0	0 0 0
c_1	0 1 1
c_2	1 0 0
c_3	1 1 0

Fig. 1. Example of the agreeing equation system (or a part of one) reducible by the method of Syllogisms.

If two equations have only one or two common variables, and if they are not agreeing, it is possible to find a 2-constraint in at least one of them, that can be used to reduce the solutions in the second one. After the reduction we get the same result as if agreeing was run. However, if we have more than two common variables, it is possible that no 2-constraints can be found that restrict the solutions, one such example is provided in the Figure 2. As l — the number of variables per equation — grows, this situation becomes more probable, and the agreeing method will be able to reduce more solutions as the syllogism method.

S_0	1 2 3		S_1	1 2 3
a_0	0 0 0		b_0	0 0 0
a_1	0 0 1		b_1	1 1 0
a_2	0 1 0	,	b_2	0 1 1
a_3	1 0 0		b_3	1 0 1
c_4	1 1 1		b_4	0 0 0

Fig. 2. Example of the disagreeing equation system without any 2-constraints.

The Syllogism method is preferable, if only weak connections (usually only one common variable) are between equations. In these cases, we can derive more information using the Syllogism rule than just by Agreeing (which only checks projection to this single common variable). In a system of random equations, this situation is more probable, when the system is very sparse. The Agreeing method provides more information when there are 3 or more common variables, and a low probability of 2-constraints. The practical experiments show (see Section 6) that the two methods have almost the same behaviour when $l = 7$. The method of syllogisms is preferable for $l < 7$, and vice-versa.

6 Experimental Results

In this section we present the results of the experiments used to locate the point of phase transition for equation systems with $m = n = 80$ variables and varying sparsity. We used each of the methods on the same set of $N = 1000$ random equation system, and $p = 0, 0.005, \ldots, 0.35$ and sparsities $l = 6, 7, 8$. Figure 3 shows the phase transition for different methods, and sparsities, respectively.

Table 2 summarizes the upper and lower bound for the transition in systems with $m = n = 100$. Precision for p is 0.02. The lower bound is the highest p, for which all 1000 equations were solved, and the upper bound is the lowest p, for which no equation was solved, respectively.

7 Conclusions

The experimental results confirm the phase transition effect. The transition is not sharp for smaller systems and sparsities. There is a region of probabilities p,

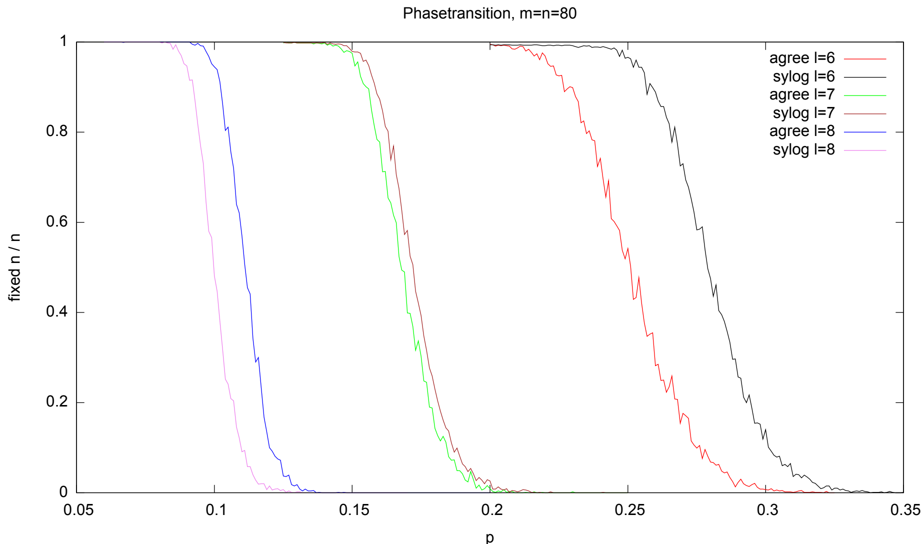


Fig. 3. Plot of the average fixation ratio showing the phase transition effect.

l	Agreeing low	Agreeing up	Syllogisms low	Syllogisms up
5	0.26	0.42	0.34	0.46
6	0.18	0.32	0.22	0.34
7	0.12	0.20	0.14	0.22
8	0.08	0.14	0.06	0.14
9	0.04	0.10	0.04	0.08

Table 2. Experimental bounds on phase transition for $m = n = 100$.

where it is possible to generate both solvable and unsolvable systems. However, as the number of variables and equation grows, the phase transition becomes sharper, and it is less probable to reduce the system above the phase transition point.

A typical situation for the random equation system is $p = 1/2$, which is above the phase transition point in every case examined. However, the consequence of the phase transition effect for smaller p's is that we can reduce the required number of guesses required before we can solve the whole equation system even if it is originally above the phase transition point.

Let us suppose we have a system of m (random) equations with $n = m$ variables, l-sparse. Each of 2^l $\{0,1\}$-vectors can be a solution of an equation in the system with probability p (usually $1/2$), i.e. the expected number of solutions in each equation is $p2^l$. The expected total number of partial solutions (listed in symbols) is then $mp2^l$. Let us guess the value of one variable, without the loss of generality x_1. We expect x_1 to be an active variable on average in l

equations. Thus we expect that we remove on average a half of $lp2^l$ partial solutions. The expected new number of solutions is thus $mp2^l - p2^{l-1}$, which is the same number, as if expect from a system generated with a lower solution probability $p' = p(1 - \frac{l}{2m})$.

After x (independent) guesses we expect the same number of partial solutions as in a system generated with $p_x = p(1 - \frac{l}{2m})^x$. To reach the zone below the phase transition point, we need to find $p_x \leq p_t$. The expected number of required guesses to reach this point is then

$$x = \frac{\log p_t - \log p}{\log(1 - \frac{l}{2m})}$$

It means, that we have to check only 2^x instead of the full 2^n possible vectors to eliminate incorrect/find the correct solution. If we can write $x = cn$ for some constant c, we get the complexity estimate $O(2^{cn})$ to determine the whole solution of the system by the guessing algorithm (in combination with A, e.g. Agreeing or Syllogism method). Estimates based on lower bounds from experimental results (see Table 2) are summarized in Table 3. We must stress, that this is only an estimate based on experiments. It is necessary to provide proper mathematical models to find the exact asymptotic behaviour of the methods. However, a full mathematical model for the reduction that takes into account all parameters m, n, p, l for both the Agreeing and Syllogism methods is still an open question.

Table 3. Estimated complexities $O(C^n)$ of the guessing algorithm for different l's. p_A is the experimental lower bound for phase transition of Agreeing, and p_S is the experimental lower bound for phase transition of Syllogism method. Columns Worst and IAG are provided for comparison with [5].

l	p_A	C	p_S	C	Worst	IAG
5	0.26	1.199	0.34	1.113	1.569	1.182
6	0.18	1.266	0.22	1.209	1.637	1.239
7	0.12	1.327	0.14	1.287		
8	0.08	1.373	0.06	1.444		

Another consequence is for the guessing order. If we want to guess a new value, we should choose the variable in such a way, so that we affect the highest number of partial solutions by the guess (resp. by guessing 0 as well as guessing 1). In this way, after removing the partial solutions that have an incorrect value for the guessed variable, we get nearer to the phase transition point. This should be the best generic guessing strategy possible. If we want to evaluate more advanced guessing strategy, e.g. applications of learning [4], it can be considered effective, if it gives a solution to the system in lower number of guesses (on average) than the guessing strategy using the phase transition.

The phase transition point is also useful for evaluating the different reduction algorithms. If two polynomial time reduction algorithms A_1, A_2 both have a

phase transition effect at solution probabilities $p_1 < p_2$, then a theoretically a more effective one is A_2. However in practice the advantage of A_2 can only be realized in large systems, which cannot be solved in practice with the present computational resources.

References

1. Goerdt, A.: *A threshold for unsatisfiability* , J Compute System Sci **53** (1996), 469–486.
2. Raddum, H., Semaev, I.: *New Technique for Solving Sparse Equation Systems*, Cryptology ePrint Archive: Report 2006/475. Available at `http://eprint.iacr.org/2006/475`.
3. Raddum, H., Semaev, I.: *Solving Multiple Right Hand Sides linear equations,* Designs, Codes and Cryptography **49** (2008), 147–160.
4. Schilling, T.E., Raddum, H.: *Solving Equation Systems by Agreeing and Learning,* preprint (2010).
5. Semaev, I.: *Improved Agreeing-Gluing Algorithm*, Cryptology ePrint Archive: Report 2010/140. Available at `http://eprint.iacr.org/2010/140`.
6. Warshall, S.: *A theorem on Boolean matrices,* Journal of the ACM **9**, No. 1 (1962), 11–12.
7. Zajac, P.: *Solving SPN-based system of equations with syllogisms*, in: 1st Plenary Conference of the NIL-I-004, Bergen, August, 24–27, 2009. (A. Kholosha — K. Nemoga — M. Sýs eds.), STU v Bratislave, 2009, pp. 21–30.
8. Zajac, P.: *Implementation of the method of syllogisms,* preprint (2010).
9. Zakrevskij, A., Vasilkova, I.: *Reducing Large Systems of Boolean Equations*, in: 4th International Workshop on Boolean Problems, Freiberg University, September, 21–22, 2000.

Paper II

5.2 Solving Equation Systems by Agreeing and Learning

Thorsten E. Schilling, Håvard Raddum
Springer Lecture Notes in Computer Science **6087**, 151–165, (2010)

Solving Equation Systems by Agreeing and Learning

Thorsten Ernst Schilling and Håvard Raddum

Selmer Center, University of Bergen
{thorsten.schilling,havard.raddum}@ii.uib.no

Abstract. We study sparse non-linear equation systems defined over a finite field. Representing the equations as symbols and using the Agreeing algorithm we show how to learn and store new knowledge about the system when a guess-and-verify technique is used for solving. Experiments are then presented, showing that our solving algorithm compares favorably to MiniSAT in many instances.

Key words: agreeing, multivariate equation system, SAT-solving, dynamic learning

1 Introduction

In this paper we present a dynamic learning strategy to solve systems of equations defined over some finite field where the number of variables occuring in each equation is bounded by some constant l. The algorithm is based on the group of Gluing-Agreeing algorithms by Håvard Raddum and Igor Semaev[1, 2]. Solving non-linear systems of equations is a well known NP-complete problem already when all equations are of degree 2; this is known as the MQ-problem [3]. Finding a method to solve such systems efficiently is crucial to algebraic cryptanalysis and could break certain ciphers that can be expressed by a set of algebraic equations, such as AES [4], HFE [5], etc.

Several approaches have been proposed to solve such systems, among them SAT-solving [6], Gröbner-basis algorithms [7] and linearization [4]. Since our algorithm falls into the category of the *guess and verify methods*, we compared our solving technique to a state-of-the-art SAT-solving implementation, namely MiniSAT [8].

We adapt the two past major improvements to the DPLL [9] algorithms, which are *watching* and *dynamic learning* [10]. During the search for a solution the method obtains new information from wrong guesses and requires for many instances much less or almost no guessing to obtain a solution to the equation system. The method we present learns new constraints on vectors over some finite field $\mathbb{F}_q$ and can therefore be seen as a generalization of the most common learning method SAT-solvers use, which operates on single variables over $\mathbb{F}_2$. Like in SAT-solving the learning routine of our algorithm runs in polynomial time. Furthermore we show by experimental results that our approach outperforms

MiniSAT for a certain class of equation systems, while there still is space for improvement of the method.

The paper is organized as follows. In Section 2 we explain the symbol representation of equations and the basic idea for agreeing. Section 3 introduces the concept of pockets, and how pockets efficiently integrate with guessing and agreeing. Section 4 shows how the solving technique can gather new (valuable) information from wrong guesses, and Section 5 compares our proposed method to MiniSAT. Section 6 conculdes the paper.

2 Preliminaries

Let

$$f_0(X_0) = 0, f_1(X_1) = 0, \dots, f_{m-1}(X_{m-1}) = 0 \qquad (1)$$

be an equation system in m equations and $n = |X| = |X_0 \cup X_1 \cup \dots X_{m-1}|$ variables over some finite field $\mathbb{F}_q$. Equations f_i are often given in their ANF-form using the variables in X_i, but here we will use symbol representation.

Definition 1 (Symbol). *Let $f_i(X_i) = 0$ be an equation over some finite field $\mathbb{F}_q$. We say that $S_i = (X_i, V_i)$ is its corresponding symbol where X_i is the set of variables in which the equation f_i is defined and V_i is the set of vectors over $\mathbb{F}_q$ in variables X_i for which $f_i(X_i) = 0$ is satisfied.*

Following this definition the system (1) can be expressed by a set of symbols $\{S_0, S_1, \dots, S_{m-1}\}$. The cost of transforming (1) to a set of symbols is clearly dominated by the number of equations and the variables involved per equation. Let $l = max\{|X_i| \mid 0 \leq i < m\}$. Transforming the system (1) to a set of symbols can be done in time $O(mq^l)$ and we say that (1) is l-sparse. The examples in this paper will only consider $q = 2$, which is the case for most equation systems arising in practice.

Example 1 (Symbol). Let the equation

$$f_0(X_0 = \{x_0, x_1, x_2\}) = x_0 \oplus x_1 x_2 = 0$$

be given over $\mathbb{F}_2$. In order to construct $S_0 = (X_0, V_0)$ we need to know V_0. Every vector $v_i \in V_0$ represents by definition a solution to $f_0(X_0) = 0$ and by searching over all 2^3 vectors in 3 variables and evaluating them we can compute V_0. Therefore the corresponding symbol is

$$S_0 = (X_0 = \{x_0, x_1, x_2\}, V_0 = \{(0,0,0), (0,0,1), (0,1,0), (1,1,1)\}).$$

Throughout the paper a symbol S_0 is represented in table-form for better readability. For this example S_0 it is

S_0	0 1 2
a_0	0 0 0
a_1	0 0 1
a_2	0 1 0
a_3	1 1 1

where the integers $0, 1, 2$ in the first row indicate the variables x_0, x_1, x_2 and $a_0, \ldots, a_3$ are identifiers of the vectors in V_0.

2.1 Agreeing

In order to find a solution to (1) the Agreeing algorithm attempts to delete vectors from symbols S_i which cannot be part of a common solution. In the following, the projection of a vector v_k on variables A is denoted by $v_k[A]$ and $V[A]$ denotes the set of projections of all vectors $v_k \in V$ on variables A.

Given two symbols $S_i = (X_i, V_i)$ and $S_j = (X_j, V_j)$ with $i \neq j$ we say that S_i and S_j are in a non-agreeing state if there exists at least one vector $a_p \in V_i$ such that $a_p[X_i \cap X_j] \notin V_j[X_i \cap X_j]$. If there exists a solution to the system, each symbol will contain one vector that matches the global solution. The vector a_p cannot be combined with any of the possible assignments in symbol S_j, hence it cannot be part of a solution to the whole system and can be deleted. The deletion of all vectors $a_p \in V_i$ and $b_q \in V_j$ which are incompatible with all vectors in V_j and V_i, respectively, is called agreeing. If by agreeing the set of vectors of a symbol gets empty, there exists no solution to the equation system. The agreeing of all pairs of symbols in a set of symbols $\{S_0, \ldots, S_{m-1}\}$ until no further deletion of vectors can be done is called the Agreeing algorithm.

Example 2 (Agreeing). The following pair of symbols is in a non-agreeing state:

S_0	0	1	2
a_0	0	0	0
a_1	0	0	1
a_2	0	1	0
a_3	1	1	1

S_1	0	1	3
b_0	0	0	0
b_1	1	0	1

.

The vectors a_2, a_3 differ from each b_j in their projection on common variables x_0, x_1 and can be deleted. Likewise, b_1 cannot be combined with any of the a_i and can also be deleted. After agreeing the symbols become:

S_0	0	1	2
a_0	0	0	0
a_1	0	0	1

S_1	0	1	3
b_0	0	0	0

.

2.2 Guessing

In the example above a further simplification of the equation system by agreeing is not possible. One has to introduce a *guess* to the system. With Example 2, that can be the deletion of vector a_0. The system is in an agreeing state and there exists only a single vector in V_0 and V_1 which gives us a local solution to the equation system, namely the combination of a_1 and b_0, that is $x_0 = 0, x_1 = 0, x_2 = 1, x_3 = 0$.

Since practical examples of equation systems are fully or almost fully pair-wise agreeing, a single run of the Agreeing-algorithm obtains no or little extra

information about the solution to the system. Thus guessing a vector $g \in V_i$ and deleting all other $v \in V_i, v \neq g$ of a symbol and verifying the partial solution by agreeing is a way to find a solution. If the guess was wrong the changes to the equation system are undone and another guess is introduced.

3 Pocket-Agreeing

We introduce an improvement of the Agreeing algorithm based on the tuple propagation by I. Semaev [11]. The Pocket Agreeing is closer to a potential software implementation and offers some speed advantages and a simple learning process.

The goal is to implement a software method to verify a guess fast. Another aspect is fast backtracking. That means that when a guess is confirmed as incorrect, the guess should be undone fast to avoid unnecessary overhead during the computation.

Definition 2 (Pocket). *Let $S_i = (X_i, V_i)$ and $S_j = (X_j, V_j)$ be two pair-wise agreeing symbols with $X_i \cap X_j = X_{i,j}$ and $|X_{i,j}| > 0$. For every projection $\rho \in V_i[X_{i,j}]$ one creates a pair of pockets*

$$p_\alpha = (\{a \mid a \in V_i \text{ and } a[X_{i,j}] = \rho\}, \beta), p_\beta = (\{b \mid b \in V_j \text{ and } b[X_{i,j}] = \rho\}, \alpha))$$

with α and β as unique identifiers or $\emptyset$. For the pocket $p_\alpha = (A, \beta)$, we use the notation $V(p_\alpha) = A$ and $I(p_\alpha) = \beta$.

The purpose of pockets is to have a system that easily identifies vectors that cannot be part of a global solution. Assume that all the vectors in a pocket p are identified as incompatible with a global solution for the system in its current state, and get deleted. Then we can immediately delete all vectors in pocket $I(p)$ since these have the same assignment of variables also found in p, and so must be inconsistent with a global solution too. Also note that one particular vector from a symbol will in general appear in several different pockets. When a vector is deleted from one pocket it is also simultaneously deleted from all the other pockets where it appears.

Example 3 (Pocket). Given the symbols S_0, S_1 from Example 2 after they are pairwise agreeing, $X_0 \cap X_1 = X_{0,1} = \{0, 1\}$. There exists only one projection $V_0[X_{0,1}] = \{(0, 0)\}$, thus there is only one pair of pockets to create, namely

$$p_0 = (\{a_0, a_1\}, 1)$$
$$p_1 = (\{b_0\}, 0).$$

3.1 Propagation

Given a set of pockets generated from symbols $S_0, \ldots, S_{m-1}$ one can run agreeing through pockets. In order to do so efficiently one assigns a *flag* to each vector

in the problem instance instead of actually deleting them. The flag of a vector a_i can have three values: *undefined, marked*, and *selected*, where the flags of all vectors are initially undefined. If a vector a_i is marked, denoted by $\overline{a_i}$, it is not suitable for extending the current partial solution, i.e. it is considered to be deleted. If an a_i is selected, denoted by a_i^+, it is considered to be part of the current partial solution, and cannot be deleted. In other words a_i^+ is *guessed.*

The main rule of propagating information in Pocket-Agreeing for the set of pockets is: "While there is a pocket $p_q = (A, b)$ where all $a_i \in A$ are marked, mark all vectors in the pocket p_b, if $b \neq \emptyset$."

This method is analogous to agreeing, where vectors whose projection is not found in another symbol are deleted. In Pocket-Agreeing equal projections are calculated beforehand, stored as pockets, and instead of being deleted as soon as they are not suitable for extending a partial solution, the vectors are flagged as marked.

3.2 Watching

One technical improvement of the Pocket-Agreeing is the possibility to introduce watches as done in SAT-solving. If one wants to implement the Pocket-Agreeing one has to check constantly if all $a_i \in A$ are marked in a pocket (A, b). Experiments show that this consumes a lot of time during the propagation.

In order to avoid this, one assigns in every pocket p a *watch* $w \in V(p)$. Only if the w gets marked it is checked if all the other $a_i \in V(p)$ are marked too. If w gets marked there are two possible cases to distinguish:

1. All $a_i \in V(p)$ are marked, and by the propagation rule all vectors in the pocket $I(p)$ have to be marked, too.
2. There exists at least one $a_i \in V(p)$ which is not marked. This is then the new watch.

This technique reduces the time used in the propagation phase. Also backtracking, i.e., *undoing* a guess in case it was wrong, is sped up. If at some point in the program the conclusion is reached that the guess was wrong, one wants to undo the changes - namely markings - caused by the last guess, in order to try another guess.

To do so one just undoes the marking of vectors from the last guess, since pockets were not changed. The watches can stay the same, since they were by construction the last vectors which got marked in the pocket, or they are not marked at all.

3.3 Guessing

The process of guessing starts with selecting one symbol S_i where all but one vector from V_i are marked. The remaining vector v^+ gets flagged as selected in order to remember that it is guessed to be part of a correct solution. Then Pocket-Agreeing based on the latest markings is started.

Two possible outcomes of the agreeing are possible:

1. Only non-selected vectors get marked. The system is in an agreeing state.
2. At some point the algorithm marks some g^+, which is by the description above the last vector remaining for some symbol. This is called a *conflict*.

If the system ends in an agreeing state, we pick another symbol, select one of its vectors, mark the others and continue the propagation. In the case of a conflict the extension of the partial solution with the previous guess(es) was not possible, and we must backtrack.

4 Learning

During the computation of a solution to the input equation system, it is natural that wrong guesses occur. It is now interesting *why* these occur, since a wrong guess implies that a wrong branch of the search tree was visited. Usually, the implications that show a guess must be wrong only involve a subset of the introduced markings. The purpose of this section is to identify exactly which markings yield a proof of inconsistency for the system. By storing this information the solver *learns* new facts about the system, and the overall number of guesses needed to find the solution is reduced.

Definition 3 (Implication Graph). *An implication graph G is a directed graph. Its vertices are vectors which are marked.*

For a marked vector a_i the pocket $P(a_i)$ is the pocket where all vectors became marked, and by propagation caused the marking of a_i. If the marking of a_i is due to an an introduced guess then $P(a_i) = \emptyset$. The set of directed edges E consists of all markings due to propagation, i.e.:

$$(a_i, a_j) \in E \text{ if } a_i \in V(P(a_j)).$$

Edges (a_i, a_j) are labeled by $P(a_j)$.

Example 4 (Implication Graph). Let the following pockets be given.

$$\begin{aligned} p_0 &= (\{a_0\}, 1) \\ p_1 &= (\{c_0, c_1\}, 0) \\ p_2 &= (\{b_0, b_1\}, \emptyset) \\ p_3 &= (\{c_0\}, 2) \end{aligned}$$

Introducing the marking $\overline{a_0}$ would yield the following implication graph.

The implication graph is not unique and depends on the order in which empty pockets are processed.

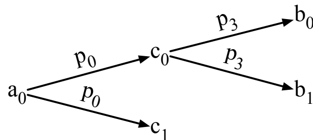


Fig. 1. Example Implication Graph.

4.1 Conflict Analysis

Let g^+ be the vector which yielded the conflict, that is it was flagged as selected and by agreeing became marked. The immediate source of the conflict is the marking of all $h_j \in V(P(g^+))$. But for further analysis we are more interested in vectors which caused the conflict by introducing a guess. These are h_j's connected to g^+ in the implication graph, where $P(h_j) = \emptyset$. By analyzing the graph we can find the h_j's recursively:

$$R(g) = \{h_j | h_j \in V(P(g)) \text{ and } P(h_j) = \emptyset\} \cup \bigcup_{\substack{h_j \in V(P(g)) \\ P(h_j) \neq \emptyset}} R(h_j). \qquad (2)$$

$R(g)$ will then be the set of marked vectors due to *guesses*, that caused g to be marked. In other words, $R(g)$ tells us exactly *which* of the introduced guesses that are incompatible with g being part of the solution. This information can be stored as a new pocket, as shown in the following.

4.2 Conflict Construction & Reduction

Assume the marking of g^+ yields a conflict and we have found that $R(g) = \{h_0, h_1, \ldots, h_r\}$ are the marked vectors that imply the marking of g^+. We can now create a new pair of pockets with the implication

$$R(g) \Rightarrow g,$$

i.e., if all vectors in $R(g)$ are marked, then g must be marked. The pockets expressing this are

$$\begin{aligned} p_{s^*} &= (\{h_1, \ldots, h_r\}, t) \\ p_t &= (g, \emptyset). \end{aligned} \qquad (3)$$

However, storing (3) for further computation does not give us any new information, since it is a direct consequence of agreeing. We are more interested in a reduced condition under which we can mark g and exclude it from a common solution during the search process. The following lemma shows how to find a reduced condition for when g can be marked.

Lemma 1. *Let the pockets $p = (\{h_1, \ldots, h_r\}, q)$ and $p_q = (\{g_1, \ldots, g_s\}, \emptyset)$ be given. For any h_j and g_i, let $X_{g_i h_j}$ be the set of variables that are common to both*

h_j and g_i. Let H be the set of vectors $h_j \in V(p)$ such that $h_j[X_{g_i h_j}] \neq g_i[X_{g_i h_j}]$ for all i. Then marking all vectors in $V(p) \setminus H$ implies marking all vectors in $V(p_q)$.

Proof: Mark all vectors in $V(p) \setminus H$ and assume that some g_i is part of the solution to the system and should not be marked. Since any vector $h_j \in H$ is different in its projection on $X_{g_i h_j}$ from $g_i[X_{g_i h_j}]$, no vectors in H can be combined with g_i in a global solution, so all vectors in H must be marked. Then the pocket p yields that g_i must also be marked. This conflict shows that g_i cannot be part of the solution to the system after all, so all vectors in $V(p_q)$ should be marked once the vectors in $V(p) \setminus H$ are marked. □

Using this lemma, we delete from the vectors in $R(g)$ all h_j for which is true that

$$h_j[X_{gh_j}] \neq g[X_{gh_j}],$$

and save the implication in a pair of pockets:

$$\begin{aligned} p_s &= (\{h_j | h_j \in R(g) \text{ and } h_j[X_{gh_j}] = g[X_{gh_j}]\}, t) \\ p_t &= (g, \emptyset). \end{aligned}$$

These two pockets are then added to the list of pockets the system already knows.

From the conflict described above we can also derive further new knowledge. Up until now we have our reduced implication $p_s \Rightarrow p_t$, i.e. if all vectors in p_s are marked, mark the vector $g \in V(p_t)$. Also, it holds for any vector g that

$$g^+ \equiv \overline{g_1}, \overline{g_2}, \ldots, \overline{g_r} \text{ with } g_i \neq g \text{ and } g, g_1, \ldots, g_r \text{ are all vectors in a symbol} \quad (4)$$

Thus g can become an *implicit guess* by marking all other g_i's in the same symbol. From the pair of pockets p_s, p_t we can now further derive that if g is guessed, at least one of the vectors in p_s has to be selected. Otherwise all h_j in p_s would be marked, and the pockets p_s, p_t would yield a conflict. We express this with the following lemma.

Lemma 2. *Let the pockets $p_s = (h_1, \ldots, h_r, t)$ and $p_t = (g, \emptyset)$ be given. For any symbol $S_\gamma = (X_\gamma, V_\gamma)$ such that $V_\gamma \cap V(p_s) \neq \emptyset$ the implication of the following pockets must hold:*

$$\begin{aligned} p_{s_\gamma} &= (\{g_1, \ldots, g_r | g_i \neq g\} \cup (V(p_s) \setminus V_\gamma), t_\gamma) \\ p_{t_\gamma} &= (V_\gamma \setminus V(p_s), \emptyset) \end{aligned}$$

Proof: Let $p_s = (\{a_u, \ldots, a_v, b_x, \ldots, b_y\}, t)$ where $\{b_x, \ldots, b_y\} = V(p_s) \cap V_\gamma$. Then the condition $g^+, \overline{a_u}, \ldots, \overline{a_v}$ implies that one of $b_x, \ldots, b_y$ has to be selected (guessed). Otherwise, if none of $b_x, \ldots, b_y$ are selected all vectors in $V(p_s)$ are marked, and g has to be marked too (by $p_s \Rightarrow p_t$). This would be a conflict since g^+ is implicitly selected. Guessing one of $b_x, \ldots, b_y$ implies the marking of all vectors in $V_\gamma \setminus \{b_x, \ldots, b_y\}$, which is exactly the set of vectors in p_{t_γ}. □

By using Lemma 1, we should also reduce the condition for when the vectors in $V(p_{t_\gamma})$ can be deleted by excluding vectors in $V(p_{s_\gamma})$ that differ in projection on common variables to all vectors in $V(p_{t_\gamma})$.

Remark 1 (Cycle-rule). Lemma 1 is an extension to the cycle-rule by Igor Semaev [12]. The cycle-rule states that through (4) it is possible to delete from an implication $\overline{a_0}, \ldots, \overline{a_r} \Rightarrow \overline{h_0}, \ldots, \overline{h_s}$ those a_i which belong to the same symbol as $h_0, \ldots, h_s$. However, the cycle-rule is extended by removing vectors from $a_0, \ldots, a_r$ which do not belong to the same symbol, but only differ in their projection from the vectors $h_0, \ldots, h_s$. Note that if two vectors belong to the same symbol, they always differ in their projection on common variables.

4.3 Non-chronological Backtracking

After the learning is completed the last guess should be undone and based on the extended pocket database Agreeing should run again. If the system is now in a non-agreeing state it can only be due to newly learnt pockets p_s. Thus any change to the system that does not involve vectors in $V(p_s)$ will necessarily result in a conflict again. Therefore we can jump back to the tree-level at which the last change in an p_s occurred, depending on which pocket yielded the conflict. This way we cut futile branches of the search tree and economize the search in the number of guesses.

Example 5. Let the following equation system be given:

$$\begin{array}{c|c} S_0 & 1\,2\,3 \\ \hline a_0 & 0\,0\,0 \\ a_1 & 0\,1\,1 \\ a_2 & 1\,1\,0 \\ a_3 & 1\,1\,1 \end{array}, \quad \begin{array}{c|c} S_1 & 2\,4\,5\,6\,12 \\ \hline b_0 & 0\,1\,0\,0\,\;0 \\ b_1 & 0\,1\,0\,1\,\;0 \\ b_2 & 0\,1\,1\,0\,\;1 \\ b_3 & 1\,0\,1\,1\,\;1 \end{array}, \quad \begin{array}{c|c} S_2 & 4\,7\,8 \\ \hline c_0 & 1\,0\,0 \\ c_1 & 1\,0\,1 \\ c_2 & 0\,1\,0 \\ c_3 & 0\,1\,1 \end{array}, \quad \begin{array}{c|c} S_3 & 1\,9\,10 \\ \hline d_0 & 0\,0\,\;1 \\ d_1 & 0\,1\,\;0 \\ d_2 & 1\,0\,\;0 \\ d_3 & 1\,1\,\;1 \end{array}, \quad \begin{array}{c|c} S_4 & 10\,11\,12 \\ \hline e_0 & 0\;\;0\;\;1 \\ e_1 & 0\;\;1\;\;0 \\ e_2 & 1\;\;0\;\;0 \\ e_3 & 1\;\;1\;\;1 \end{array}, \quad \begin{array}{c|c} S_5 & 9\,11\,12 \\ \hline f_0 & 0\;\;0\;\;1 \\ f_1 & 0\;\;1\;\;0 \\ f_2 & 1\;\;0\;\;0 \\ f_3 & 1\;\;1\;\;1 \end{array}.$$

The intersection graph in Figure 2 indicates pairs of symbols from which pockets are generated. The labled edges between symbols show intersections in the sets of variables. No pockets are generated from the pair S_1, S_5 since changes of variable x_{12} will propagate through the path S_1, S_4, S_5 while agreeing.

Assume that by some heuristic the order of symbols to be guessed is $S_0, S_1, S_2, S_3, S_4, S_5$. The partial solutions a_0^+, b_0^+, c_0^+ are selected in that order. This results in the following equation system after agreeing:

$$\begin{array}{c|c} S_0 & 1\,2\,3 \\ \hline a_0 & 0\,0\,0 \end{array}, \quad \begin{array}{c|c} S_1 & 2\,4\,5\,6\,12 \\ \hline b_0 & 0\,1\,0\,0\,\;0 \end{array}, \quad \begin{array}{c|c} S_2 & 4\,7\,8 \\ \hline c_0 & 1\,0\,0 \end{array}, \quad \begin{array}{c|c} S_3 & 1\,9\,10 \\ \hline d_0 & 0\,0\,\;1 \\ d_1 & 0\,1\,\;0 \end{array}, \quad \begin{array}{c|c} S_4 & 10\,11\,12 \\ \hline e_1 & 0\;\;1\;\;0 \\ e_2 & 1\;\;0\;\;0 \end{array}, \quad \begin{array}{c|c} S_5 & 9\,11\,12 \\ \hline f_1 & 0\;\;1\;\;0 \\ f_2 & 1\;\;0\;\;0 \end{array}.$$

For a further extension of the partial guess one tries to extend the partial solution by d_0. The resulting implication graph after marking d_1 is shown below. Marking d_1 causes e_1 to be marked by pocket p_{18}, which again causes f_1 and d_0 to be marked by pockets p_{26} and p_{21}. This is clearly a conflict, since d_0 was previously selected but should be marked now. Now we analyze the source of the conflict in order to learn from it.

$$R(d_0) = \{a_1, a_2, a_3, b_2, d_1\}$$

$p_0 = (\{a_0\}, 1)$ $p_1 = (\{b_0, b_1, b_2\}, 0)$
$p_2 = (\{a_1, a_2, a_3\}, 3)$ $p_3 = (\{b_3\}, 2)$
$p_4 = (\{a_0, a_1\}, 5)$ $p_5 = (\{d_0, d_1\}, 4)$
$p_6 = (\{a_2, a_3\}, 7)$ $p_7 = (\{d_2, d_3\}, 6)$
$p_8 = (\{b_0, b_1, b_2\}, 9)$ $p_9 = (\{c_0, c_1\}, 8)$
$p_{10} = (\{b_3\}, 11)$ $p_{11} = (\{c_2, c_3\}, 10)$
$p_{12} = (\{b_0, b_1\}, 13)$ $p_{13} = (\{e_1, e_2\}, 12)$
$p_{14} = (\{b_2, b_3\}, 15)$ $p_{15} = (\{e_0, e_3\}, 14)$
$p_{16} = (\{d_0, d_3\}, 17)$ $p_{17} = (\{e_2, e_3\}, 16)$
$p_{18} = (\{d_1, d_2\}, 19)$ $p_{19} = (\{e_0, e_1\}, 18)$
$p_{20} = (\{d_0, d_2\}, 21)$ $p_{21} = (\{f_0, f_1\}, 20)$
$p_{22} = (\{d_1, d_3\}, 23)$ $p_{23} = (\{f_2, f_3\}, 22)$
$p_{24} = (\{e_0\}, 25)$ $p_{25} = (\{f_0\}, 24)$
$p_{26} = (\{e_1\}, 27)$ $p_{27} = (\{f_1\}, 26)$
$p_{28} = (\{e_2\}, 29)$ $p_{29} = (\{f_2\}, 28)$
$p_{30} = (\{e_3\}, 31)$ $p_{31} = (\{f_3\}, 30)$

Fig. 2. The intersection graph and the resulting pockets. Dotted edges in the intersection graph are ignored.

Fig. 3. Implication Graph of guess a_0, b_0, c_0, d_0.

To create the reduced p_s we compare projections of a_1, a_2, a_3, b_2, d_1 in common variables to projections of d_0. We see that a_2 and a_3 have a different projection than d_0 on their common variable x_1, so these vectors can be excluded from p_s by Lemma 1. d_1 can obviously also be excluded since it belongs to the same symbol as d_0. After this reduction we get:

$$p_{32} = (\{a_1, b_2\}, 33)$$
$$p_{33} = (\{d_0\}, \emptyset).$$

Using Lemma 2 we also derive:

$$p_{34} = (\{d_1, d_2, d_3, a_1\}, 35)$$
$$p_{35} = (\{b_0, b_1, b_3\}, \emptyset)$$
$$p_{36} = (\{d_1, d_2, d_3, b_2\}, 37)$$
$$p_{37} = (\{a_0, a_2, a_3\}, \emptyset)$$

After this learning process we agree the system again, with our newly obtained knowledge. The pockets p_{32} and p_{33} cause d_0 to be marked. This implicitly selects d_1^+, which immediately yields another conflict, without introducing any new guess. Thus the guesses a_0^+, b_0^+, c_0^+ cannot all be right. We can immediately read from p_{32} where to backtrack. We see from p_{32} that the guessing of c_0^+ was not a cause for the conflict, otherwise there would be some c_i-vectors in p_{32}. This tells us that if we now backtrack and select, say c_1^+, we will end up in the very same conflict again. Hence we can go back to the point where b_0 got guessed (and b_2 marked) and try selecting another b_j-vector. Bypassing the guesses on all c_i-vectors that would be due in a naive search algorithm saves a lot of time.

Figure 4 shows the decision tree until the first solution is found. Branches not incorporating vectors from all symbols indicate conflicts. Connected to the dotted lines are the newly learned pockets. In comparison the naive search tree, without learning, is depicted in Figure 5.

4.4 Variable-based guessing

In the algorithm we have explained, we guess on which of the possible assignments in a symbol that is the correct one. It may look more natural to guess on the value of single variables as is done in SAT-solving. Given an instance $S_0, S_1, \ldots, S_m$ in variables X there exists a simple way to realize variable-based guessing. Instead of establishing a separate mechanism of introducing the guess on a single variable one inserts new symbols of the form $S_{x_i} = (\{x_i\}, \{v_0 = 0, v_1 = 1\})$ for every $x_i \in X$ before the pocket generation. These symbols contain no information but can easily be integrated into the system. Assume one wants to guess that $x_i = 0$. From the newly inserted symbol one just marks v_1 and propagates the guess by agreeing instead of keeping a separate table of all vectors in which x_i occurs as 1 and marking them. Another advantage is that this way of introducing variable guessing integrates with the learning without problems.

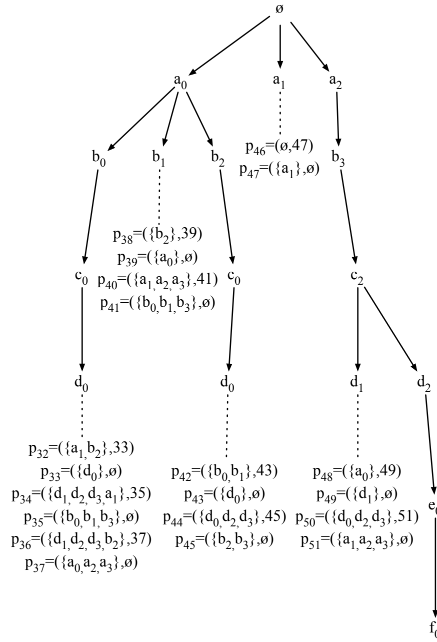


Fig. 4. Search tree with learning.

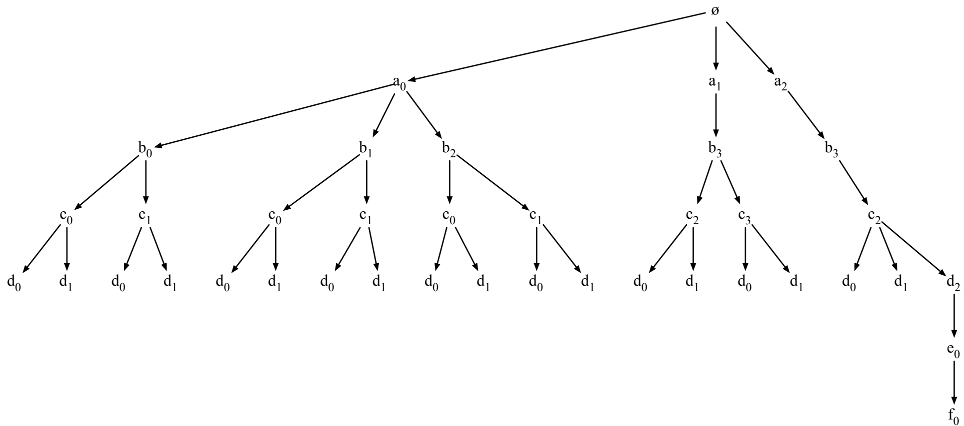


Fig. 5. Naive search tree.

Of course this approach works for other fields than $\mathbb{F}_2$, too. Assume an equation system over $\mathbb{F}_q$ then one inserts for every $x_i \in X$ a symbol $S_{x_i} = (\{x_i\}, V_{x_i} = \{v_j | v_j \in \mathbb{F}_q\})$.

5 Experiments

5.1 Results

In order to evaluate the strength of the proposed solving algorithm, several experiments were made with random equation systems over $\mathbb{F}_2$. A software, called *Gluten*, that implements the algorithm was developed. To get a comparison with another solving technique we took a SAT-solver, namely MiniSAT since the guess/verify technique to obtain a solution is similar. Furthermore SAT-solving is a well researched field and MiniSAT among the fastest programs in this field.

Rather than comparing pure solving time we compare the number of variable guesses needed until a solution to the system is obtained. During all the experiments it holds $m = n$, i.e. the number of equations is equal to the number of variables. We make sure the systems have at least one solution. The sparsity l is also fixed to $l = 5$. The ANF degree for the equations we generate will be randomly distributed, but will of course be upper bounded by the sparsity. Furthermore every $m = n$ was tested with 100 randomly generated instances and the arithmetic mean calculated afterwards.

Figures which display both very large and very small values are log-scaled for better readability.

5.2 Random Instances

In this experiment the expected number of roots for every equation is $E(|V_i|) = 2^4$ and binomially distributed, as would be the case when the symbols are obtained from random ANF's.

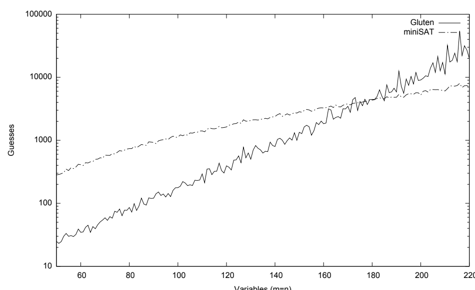


Fig. 6. Gluten vs. MiniSAT (log-scale)

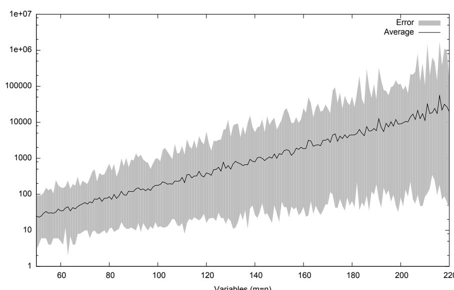


Fig. 7. Gluten average and error (log-scale)

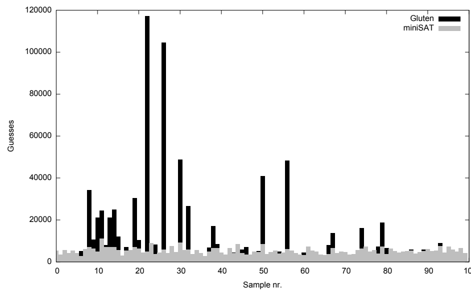


Fig. 8. $n = m = 200$

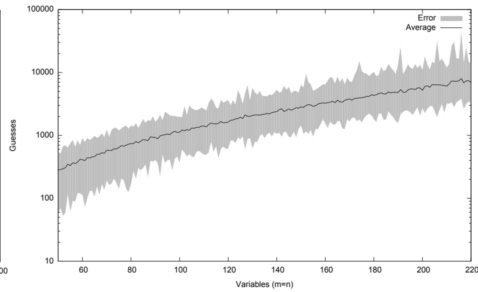


Fig. 9. MiniSAT average and error (log-scale)

In Figure 6 one can see that Gluten performs clearly better up til around $m = n = 170$. Afterwards the average values for MiniSAT stay low while the number of guesses for Gluten rise fast. Figure 7 shows that the error margin is very high to the average in comparison to the error margin of MiniSAT, shown in Figure 9. In other words, Gluten runs into a few cases where it makes an extremely high number of guesses whereas MiniSAT is able to keep its number of guesses not too far from the average.

To get a better comparison of both methods in Figure 8, the case of $n = m = 200$ along with the sample number is given. For every of the 100 samples the black bar indicates the number of variable guesses Gluten took to obtain a solution and the grey bar shows the number of guesses MiniSAT took to find a solution. In approximately 1/3 of all samples Gluten performs worse, in the rest approximately equally or better.

5.3 Uniformly distributed number of roots

The case when the number of roots in the equations are distributed uniformly at random was also investigated. That means that the size of V_i is taken uniformly at random from $[1, 2^l - 1]$ for each symbol.

In this scenario Gluten performs much better on the whole spectrum of the experimental data. As Figure 10 and 11 shows the number of guesses for Gluten rise linearly while the curve giving the number of MiniSAT's guessings seems to be quadratic (the polynomial $0.0232n^2 + 1.6464n - 15.4$ fits the dashed curve very well). The Gluten values are less than 50; note the different scalings in Figure 10 and 11. It is also interesting to notice that Gluten only needs to make very few guesses, even for systems with over 250 variables.

6 Conclusion & Further Work

We have shown how new knowledge about the equation system can be obtained in polynomial time when guessing partial solutions and running the Agreeing

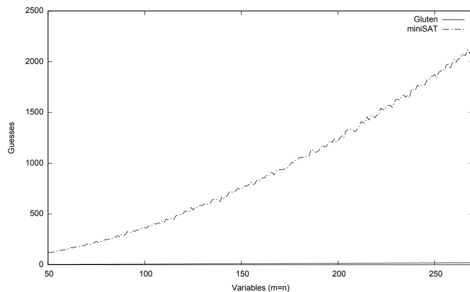


Fig. 10. Gluten vs. MiniSAT

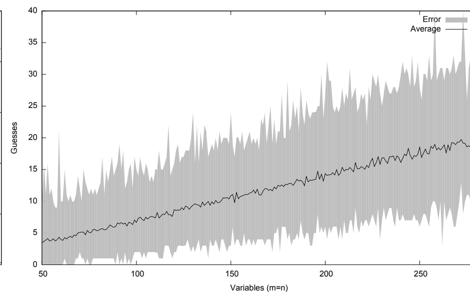


Fig. 11. Gluten average and error

algorithm. New constraints on vectors defining partial solutions can be added and using this, futile search-regions can be pruned. Our experiments show our proposed algorithm performs better than SAT-solving in a large number of instances. In particular, the experimental data shows that it is only necessary to make a small number of guesses to solve systems where the number of roots are uniformly distributed.

Several mechanisms are not yet introduced to our algorithm. Among them are random restarts during the search process or random guesses. It is obvious that a good guessing heuristic is crucial for the success of a solver of this kind. While SAT-solving is well studied and a lot of different search-heuristics are available, this is still an open field and topic for future research for the algorithm proposed in this paper.

References

1. Raddum, H.: MRHS Equation Systems. Lecture Notes in Computer Science **4876** (2007) 232–245
2. Raddum, H., Semaev, I.: Solving Multiple Right Hand Sides linear equations. Designs, Codes and Cryptography **49**(1) (2008) 147–160
3. Courtois, N., Patarin, J.: About the XL Algorithm over GF(2). Lecture Notes in Computer Science **2612** (2003) 141–157
4. Courtois, N., Pieprzyk, J.: Cryptanalysis of Block Ciphers with Overdefined Systems of Equations. Lecture Notes in Computer Science **2501** (2002) 267–287
5. Kipnis, A., Shamir, A.: Cryptanalysis of the HFE Public Key Cryptosystem by Relinearization. In: CRYPTO '99: Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology, London, UK, Springer-Verlag (1999) 19–30
6. Massacci, F., Marraro, L.: Logical cryptanalysis as a SAT problem. Journal of Automated Reasoning **24**(1) (2000) 165–203
7. Faugère, J.: A new efficient algorithm for computing Gröbner bases (F4). Journal of Pure and Applied Algebra **139**(1-3) (1999) 61–88
8. Een, N., Sörensson, N.: Minisat v2.0 (beta). Solver description, SAT Race `http://fmv.jku.at/sat-race-2006/` (2006)

9. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Commun. ACM **5**(7) (1962) 394–397
10. Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proceedings of the 38th conference on Design automation, ACM New York, NY, USA (2001) 530–535
11. Semaev, I.: Sparse algebraic equations over finite fields. SIAM Journal on Computing **39**(2) (2009) 388–409
12. Semaev, I., Schilling, T.: Personal correspondence (2009)

Paper III

5.3 Analysis of Trivium Using Compressed Right Hand Side Equations

Thorsten E. Schilling, Håvard Raddum
Springer Lecture Notes in Computer Science **7259**, 18–32, (2012)

Analysis of Trivium Using Compressed Right Hand Side Equations

Thorsten Ernst Schilling, Håvard Raddum
thorsten.schilling@ii.uib.no,havard.raddum@ii.uib.no

Selmer Center, University of Bergen

Abstract. We study a new representation of non-linear multivariate equations for algebraic cryptanalysis. Using a combination of multiple right hand side equations and binary decision diagrams, our new representation allows a very efficient conjunction of a large number of separate equations. We apply our new technique to the stream cipher Trivium and variants of Trivium reduced in size. By merging all equations into one single constraint, manageable in size and processing time, we get a representation of the Trivium cipher as one single equation.

Key words: multivariate equation system, BDD, algebraic cryptanalysis, Trivium

1 Introduction

In this paper we present a new way of representing multivariate equations over $GF(2)$ and their application in algebraic cryptanalysis of the stream cipher Trivium.

In algebraic cryptanalysis one creates an equation system of the cipher being analyzed and tries to solve it. The solution will reveal the key or some other secret information. Solving the system representing a cipher in time faster than exhaustive search will be a valid attack on the cipher.

There exist several ways to represent such a system, e.g., ANF, CNF [1] or MRHS [2]. Along these representations different families of algorithms to solve equation systems have been proposed, e.g., Gröbner Basis like algorithms [3], XL [4] SAT-solving [1] and Gluing/Agreeing algorithms [5, 2, 6].

For the stream cipher Trivium, which has an especially simple structure, one can easily construct an equation system describing its inner state constraints using some known keystream bits. Attempts at solving this system have nevertheless been unsuccessful. While reduced versions of Trivium could be broken [1], there is no attack better than brute-force known for the full version.

Previous methods describe the Trivium-equation system as a set of non-linear constraints, which have to be true in conjunction. One can simplify those equation systems by joining several constraints into a single new one. Unfortunately the conjunction operation usually leads to exponentially big objects, which quickly become too big for today's computers.

In this paper we present a new way of representing the constraints given by a non-linear equation system. This representation allows all equations in the TRIVIUM-equation system to be merged into one single equation. The process of merging equations has asymptotically exponential complexity, but using our new technique we are nevertheless still able to complete it in practice, with an actual complexity far lower than the $O(2^{80})$-bound for TRIVIUM.

The paper is organized as follows. In Section 2 we explain the Multiple Right Hand Side equation representation and Binary Decision Diagrams as well as some operations on both constructions. The cipher TRIVIUM is also briefly described. Section 3 introduces Compressed Right Hand Side equations and shows how a solution to such equations can be found. In Section 4 we present our experimental results and explain how to reduce the TRIVIUM equation system to a single Compressed Right Hand Side equation. Section 5 concludes the paper. The appendix contains examples for several of the used constructions and algorithms.

2 Preliminaries

2.1 Multiple Right Hand Side Equation Systems

The Multiple Right Hand Side (MRHS) representation [2, 5] is an efficient way to represent equations containing much inherent linearity. Equation systems coming from cryptographic primitives are well suited for MRHS representation, since cryptographic algorithms are usually built using both linear and non-linear components.

A MRHS equation is a linear system with, as the name suggests, multiple right hand sides. We write one MRHS equation as $Ax = B$, where A and B are matrices with the same number of rows, and x is a vector of variables. Any assignment of x such that Ax equals some column in B satisfies the equation.

We construct a system of MRHS equations from a cryptographic primitive as follows. First we assign variable names to the bits of cipher states at several places in the encryption process. The assignment of variables should be done such that the bits of the input and output of any non-linear component can be written as linear combinations of variables. Then we construct one MRHS equation $Ax = B$ for each non-linear component f. The rows of A are the input and output linear combinations of f. Finally, we list all possible inputs to f, with their corresponding outputs. Each input/output pair becomes a column in B. An example of this can be found in the appendix.

Following this procedure we can construct a system of MRHS equations

$$A_1x = B_1, \dots, A_mx = B_m$$

for any cryptographic primitive that uses relatively small non-linear components.

For a given solution to the system, there is exactly one column in each B_i corresponding to this solution. We say such a column is *correct*. If the system has a unique solution, there is only one correct right hand side in each B_i. Solving

MRHS equation systems means identifying columns in the B_i that cannot be correct, and delete them.

Several techniques for solving MRHS systems exist. One of them is called *gluing* and is used in this paper. Gluing means to merge two equations into one, making sure that only solutions that satisfy both original equations are carried over into the new (glued) equation.

Gluing two equations reduces the number of equations by one. The process of gluing can be repeated, packing all initial equations into one MRHS equation. The resulting equation is nothing more than a system of linear equations, and can easily be solved. The solution we find will necessarily satisfy all the original initial MRHS equations, so this strategy will solve the system in question.

The problem we face when applying the technique of gluing in practice, is that the number of right hand sides in glued equations tends to increase exponentially. Only when there are just a few equations remaining, with large A-matrices, will the restrictions on potential solutions be so limiting that the number of possible right hand sides rapidly decreases. As we shall see, however, the problem of exponential growth in the number of right hand sides may be circumvented using *binary decision diagrams.*

2.2 Binary Decision Diagrams

In this section we will introduce binary decision diagrams (BDDs). A BDD is a directed acyclic graph used to represent a set of binary vectors or a Boolean formula. They are mostly used in design and verification systems and were introduced by S.B. Akers [7]. Later implementations and refinements led to a broad interest in the computer science community as BDDs allow the manipulation of large propositional formulae [8, 9] in compressed form. Sometimes they are used as an alternative to *guess-and-verify* solvers of propositional problems since they enable one to keep track of all satisfying assignments at once and offer polynomial time algorithms to count the number of solutions of a propositional problem given in the form of a BDD.

The use of BBDs in cryptanalysis for LFSRs was proposed by Krause [10] and successfully applied to Grain with NLFSRs by Stegemann [11].

Definition 1 (Binary Decision Diagram). *A binary decision diagram is a pair $\mathcal{D} = (G, L)$ where $G = (V, E)$ is a directed acyclic graph, and $L = (l_0, l_1, \ldots, l_{r-1}, \epsilon)$ is an ordered set of variables.*

The vertices of G are $V = \{v_0, v_1, \ldots, v_{s-1}\} \cup \{\top, \bot\}$ where all v_i denote inner vertices and contain exactly one root vertice with no incoming edges. Every inner vertex v has exactly two outgoing edges, which we call the 1-edge and the 0-edge. We call $\top$ and $\bot$ terminal vertices, they have no outgoing edges. Every vertex v is associated with a variable, denoted $L(v)$, and for all edges (u, v) we have $L(u)$ appearing before $L(v)$ in L. We always have $L(\top) = L(\bot) = \epsilon$.

We denote with $G(v)$ the subgraph of G rooted at v, i.e., the graph consisting of vertices and edges along all directed paths originating at v. For any pair of vertices u, w it holds that if $G(u) = G(w)$ then $u = w$.

There exist other definitions of BDDs which do or do not include the order L or the reducedness property of unequal subgraphs. The definition above is also known as a reduced ordered BDD and is canonical [9]. We denote the number of vertices in a binary decision diagram $\mathcal{D}$ by $\mathcal{B}(\mathcal{D}) = |G|$. The size of a BDD depends heavily on the order L. Finding the optimal ordering to minimize $\mathcal{B}(\mathcal{D})$ is an NP-hard problem [9].

In Definition 1 L induces a partial order of the vertices. We visualize a BDD by drawing it from top to bottom, with vertices of the same order on the same line, and we say that these vertices are at the same *level*. There is only one root vertex and it must necessarily associated with the first variable in L. This node associated with l_0 is drawn on top, and the nodes $\top$ and $\bot$ are drawn on the bottom. An example of a BDD can be found in the appendix.

Definition 2 (Accepted Inputs of a BDD). *In a BDD $\mathcal{D}$ every path from the root vertex to the terminal vertex $\top$ is called an accepted input of $\mathcal{D}$.*

Since every inner node is associated with a variable, we can regard every edge as a *variable assignment*. To find a variable assignment (or vector) which is accepted by the BDD, we start with an empty vector of length $|L|$. Following a path from the root vertex to $\top$ we visit at most one node at each level.

Whenever we go from v through a 1-edge, we say that $L(v)$ is assigned to 1, and $L(v) = 0$ whenever we go via a 0-edge. A path that ends up in $\top$ gives us one accepted input in terms of variable assignments. Likewise, a path from the root vertex to $\bot$ gives us a rejected input to a specific BDD. By traversing all paths to $\top$ we can build the set of all vectors which are accepted by the BDD.

If a path from the root to $\top$ *jumps* a level, i.e. the assignment to a variable l_k is undefined since the path does not contain a vertex v with $L(v) = l_k$, both assignments to this variable are accepted and we get two different variable assignments. If an accepted input jumps r levels in total we get 2^r different satisfying assignments from this path. An example of accepted inputs of a BDD can be found in the appendix.

AND-Operation on BDDs. As shown above, we can use BDDs to represent the set of vectors that satisfy a Boolean equation. By the nature of our equation systems, we need a way to merge solution sets from different equations. Below is a simple recursive algorithm which does this. A more general version of the algorithm can be found in [12].

Let $\mathcal{D}$ and $\mathcal{D}'$ be two BDDs with v_0 as the root of $\mathcal{D}$ and u_0 the root of $\mathcal{D}'$. The conjunction of $\mathcal{D}$ and $\mathcal{D}'$ into a new BDD $\mathcal{E}$ is done as follows.

First we need to define an ordering on the union of variables from $\mathcal{D}$ and $\mathcal{D}'$. Next, we set the root node of $\mathcal{E}$ at the top level, and label it (v_0u_0). Then we perform Algorithm 1, which will fill in nodes and edges in $\mathcal{E}$, from top to bottom.

The paths in the BDD that results after merging $\mathcal{D}$ and $\mathcal{D}'$ using Algorithm 1 will correspond to vectors that satisfy both Boolean equations related to $\mathcal{D}$ and $\mathcal{D}'$. One feature of the conjunction of two BDDs is that all nodes in the new BDD can be labelled with (vu) where v and u come from the two orginal BDDs.

Algorithm 1 Merging BDDs $\mathcal{D}$ and $\mathcal{D}'$ into $\mathcal{E}$

```
while ∃ a node (vu) in E without outgoing edges do
    Let v^e be child of v in D through e-edge
    Let u^e be child of u in D' through e-edge
    if L(v) = L(u) then                               ▷ v and u are at the same level
        Insert (v^0 u^0) at level min{L(v^0), L(u^0)} with 0-edge from (vu).
        Insert (v^1 u^1) at level min{L(v^1), L(u^1)} with 1-edge from (vu).
    end if
    if L(v) < L(u) then                               ▷ v is higher up than u
        Insert (v^0 u) at level min{L(v^0), L(u)} with 0-edge from (vu).
        Insert (v^1 u) at level min{L(v^1), L(u)} with 1-edge from (vu).
    end if
    if L(v) > L(u) then                               ▷ u is higher up than v
        Insert (v u^0) at level min{L(v), L(u^0)} with 0-edge from (vu).
        Insert (v u^1) at level min{L(v), L(u^1)} with 1-edge from (vu).
    end if
end while
```

It is then not hard to see that the following upper bound holds

$$\mathcal{B}(\mathcal{E}) \leq \mathcal{B}(\mathcal{D})\mathcal{B}(\mathcal{D}'). \tag{1}$$

We will use this fact later in the paper. For a more detailed description and analysis of operations on BDDs one might consult [12, 9, 8]. An example of the AND-operation on BDDs can be found in the appendix.

2.3 Trivium

Trivium [13] is a synchronous stream cipher and part of the ECRYPT Stream Cipher Project portfolio for hardware stream ciphers. It consists of three connected non-linear feedback shift registers (NLFSR) of lengths 93, 84 and 111. These are all clocked once for each key stream bit produced.

Trivium has an inner state of 288 bits, which are initialized with 80 key bits, 80 bits of IV, and 128 constant bits. The cipher is clocked 1152 times before actual keystream generation starts. The generation of keystream bits and updating the registers is very simple. The pseudo-code in [13] is a good and compact description of the whole process of generating keystream as shown in Algorithm 2.

Here z_i is the key stream bit, and the registers are filled with the bits $s_1, \ldots, s_{288}$ before clocking.

For algebraic cryptanalysis purposes one can create four equations for every clock; three defining the inner state change of the registers and one relating the inner state to the key stream bit. Solving this equation system in time less than trying all 2^{80} keys is considered a valid attack on the cipher.

Small Scale Trivium. For our experiments we considered small scale versions of Trivium. While reduced versions of a cipher sometimes dismiss some structural

Algorithm 2 Trivium Pseudo-Code

```
for i = 1 to N do
    t_1 ← s_66 + s_93
    t_2 ← s_162 + s_177
    t_3 ← s_243 + s_288

    z_i ← t_1 + t_2 + t_3                              ▷ Keystream bit

    t_1 ← t_1 + s_91 · s_92 + s_171
    t_2 ← t_2 + s_175 · s_176 + s_264
    t_3 ← t_3 + s_286 · s_287 + s_69

    (s_1, s_2, ..., s_93) ← (t_3, s_1, ..., s_93)
    (s_94, s_95, ..., s_177) ← (t_1, s_94, ..., s_176)
    (s_178, s_179, ..., s_288) ← (t_2, s_178, ..., s_287)
end for
```

component of the full scale cipher, e.g. Bivium [1], we try to keep our reduced versions as close to Trivium as possible.

We scale with respect to the number of bits in the state. When we speak about Trivium-N, we are speaking about a cipher with N bits of internal state, that is, scaled down by a factor $\alpha = N/288$. The lengths of the two first registers will be 93α and 84α, rounded to the nearest integers. The length of the last register will be what remains to get N as the total number of state bits (either $\lfloor 111\alpha \rfloor$ or $\lceil 111\alpha \rceil$).

In the full Trivium, the three top positons in each register are all used as tap positions. This property is also carried over to all the scaled versions. For the tap positions appearing elsewhere in the registers, we simply scale their indices with α. For example, as 66 is used as a tap position in the full Trivium, for Trivium-N the corresponding tap position will be 66α, rounded to the nearest integer, with the following exception: Tap positions that are close to each other in the full Trivium may get the same indices in some Trivium-N if α is small enough. When this happens, we reduce the tap position of the smaller index by one, thus ensuring that all tap positions in Trivium-N are distinct. The equation systems representing Trivium-N and Trivium will then have similar structures.

3 Compressed Right Hand Side Equation Systems

With MRHS equations a clear separation between the linear and the non-linear part of an equation was introduced. Overall it yielded a much smaller representation for equations typical in algebraic cryptanalysis. Nevertheless, solving MRHS equations has been limited to relatively small-scale examples because of the problem with a big number of right hand sides.

It was shown in [7] that representing Boolean equations as BDDs is canonical with respect to the ordering of variables. This way of recording sets of assign-

ments gives us the advantage that we may have a moderate number of nodes in a BDD, but very many paths from the root leading to $\top$. Rather than writing out all satisfying assignments, or a truth table for a Boolean equation, only a BDD is retained in memory. However, when experimenting with equations from certain ciphers, BDDs may also become too big to keep in computer memory [11].

By combining the MRHS and BDD approaches, we get a new way to handle large equation systems in algebraic cryptanalysis. We call this representation of equations *Compressed Right Hand Sides* (CRHS) equations.

Definition 3 (CRHS). *A compressed right hand side equation is written as $Ax = \mathcal{D}$, where A is a $k \times n$-matrix with rows $l_0, \ldots, l_{k-1}$ and $\mathcal{D}$ is a BDD with variable ordering (from top to bottom) $l_0, \ldots, l_{k-1}$. Any assignment to x such that Ax is a vector corresponding to an accepted input in $\mathcal{D}$, is a satisfying assignment.*

An easy example of a CRHS equation can be found in the appendix.

CRHS Gluing. If we are given two Boolean equations $f_1(X_1) = 0, f_2(X_2) = 0$ and we want to find vectors in variables $X_1 \cup X_2$ which satisfy both equations simultanously we can do this by investigating their individual satisfying vectors at common variables. If two vectors have the same values at common variable indices we have found a vector which satisfies both equations. This operation is part of the Gluing operation described in Section 2.1.

If we are given two CRHS equations $[C_1]x = \mathcal{D}_1, [C_2]x = \mathcal{D}_2$ and we want to compute their common solutions we use a similar technique called *CRHS Gluing.* The result of gluing both equations above is

$$\begin{bmatrix} C_1 \\ C_2 \end{bmatrix} x = \mathcal{D}_1 \wedge \mathcal{D}_2.$$

Any assignment of x such that $\begin{bmatrix} C_1 \\ C_2 \end{bmatrix} x$ is an accepted input in the conjunction $\mathcal{D}_1 \wedge \mathcal{D}_2$ gives a solution to both initial equations simultanously. Like the Gluing operation on MRHS equations the right hand side BDD contains all possible combinations of vectors from the original equations. The difference is that satisfying vectors are no longer explicit in the computer memory, but are recorded in a compressed format, namely as paths in the BDD.

It is easy to output all possible vectors from the paths in a BDD. There also exists an easy polynomial-time (in the number of nodes) algorithm to count the number of accepted inputs to a BDD. An example of CRHS-gluing can be found in the appendix.

3.1 Dependencies among linear combinations

The left hand side in a CRHS equation is equal to the left hand side in a MRHS equation, namely a set of linear combinations $\{l_0, \ldots, l_{k-1}\}$ in the variables of

the system. If we glue several CRHS equations together, it might happen that the resulting left hand side matrix in the glued equation does not have full rank, that is, the set of linear combinations in the left hand side contains linear dependencies.

The BDD on the right hand side treats the l_i as variables, and is oblivious to the constraint that some of them should sum to zero or one. Therefore, an accepted input in the BDD may or may not satisfy the linear dependencies known to the left hand side. These paths should be taken out of the BDD in order to not produce false solutions.

The straight-forward way to remove paths that do not satisfy some linear dependency is to use the AND-operation. The number of nodes in the BDD representing a linear equation $g(l_0, \ldots, l_{k-1})$ is two times the number terms in g. It is then easy to construct the BDD for any g, and combine it with the BDD in the equation using the AND-operation. This will remove all false solutions.

4 Experimental Results

While exploring the possibilities of CRHS equations we used a software library called *Cudd*[14]. The Cudd software library implements various types of BDDs and algorithms/operations which can be performed on BDDs. The code base is optimized and usable on a personal computer even for very big BDDs.

We used Cudd together with C++ code and developed a program capable of reading different equation systems representing scaled Triviums and then gluing the equations together.

It was crucial in the experiments to find out the size of the resulting CRHS equation when gluing many of them together. This number is important to determine in order to evaluate the feasability of our method. Theoretically the size of the final CRHS equation C is upper bounded by

$$\mathcal{B}(C) \leq \mathcal{B}(c_0) \cdot \mathcal{B}(c_1) \cdot \ldots \cdot \mathcal{B}(c_{r-1})$$

when gluing CRHS equations $c_0, c_1, \ldots, c_{r-1}$ into C. This value is exponential in the number of nodes and might lead to infeasible sizes of BDDs, even for quite small versions of Trivium. However, our experiments showed that the size of the BDD for the glued CRHS equations was far smaller than the upper bound, and stayed manageable. Thus we are indeed, in contrast to MRHS equation systems, able to glue *all* equations in large CRHS equation systems together. For MRHS equation systems, gluing all equations together will reveal the solutions to the system. As we explain below, it is more complicated for CRHS equation systems, due to false solutions in the right hand side BDD.

In the experiments reported below, we created CRHS equation systems representing Trivium-N for various values of N. Then we glued all equations into one single big CRHS equation. We examined different aspects of the equation systems, which can tell us something about their solvability with our method. For several small scale versions we measured the following properties:

Value	Description
n	# of variables = # of initial CRHS equations
k	# of different linear combinations of variables
$\mathcal{B}$	# vertices in BDD in final equation
lc	# of linear constraints for solution
Sol.	# paths in final BDD
Mem.	Memory consumption in MB

N	n	k	$\mathcal{B}$	lc	Sol.	Mem.
35	85	173	$2^{18.86}$	88	$2^{85.67}$	87
40	94	191	$2^{20.57}$	97	$2^{93.77}$	182
45	106	215	$2^{21.68}$	109	$2^{106.60}$	358
50	115	233	$2^{21.15}$	118	$2^{115.60}$	258
55	127	257	$2^{21.55}$	130	$2^{127.60}$	329
60	138	282	$2^{22.34}$	144	$2^{140.35}$	560
65	148	299	$2^{22.66}$	151	$2^{148.60}$	687
70	160	323	$2^{22.42}$	163	$2^{160.49}$	588
75	171	349	$2^{22.78}$	178	$2^{173.83}$	742

Table 1. Experimental results

Initial equations have 4 nodes in the BDD, so we see from Table 1 that the size of the BDD after gluing all equations together is far from the theoretical upper bound. However, the growth of $\mathcal{B}$ is exponential just with a very small constant. It is worth to notice that $\mathcal{B}$ is not strictly increasing with N. We also see that the expected number of paths that satisfy all constraints given by lc is between 2^{-4} and 2^{-2}.

A point worth mentioning is that the exponential upper bound for gluing CRHS equations together is tight, in general. There *are* equations that will achieve the bound when glued together. Equation systems coming from ciphers tend to be very sparse, in the sense that each initial equation contain few variables, and each variable only appears in a few equations. This is also the case for Trivium. Two equations that do not share any variables have a linear size when glued together. As shown in (5), the gluing in this case is basically putting one BDD on top of the other. This may explain why it is particularly easy to glue together CRHS equations coming from scaled versions of Trivium.

Full Trivium. So what about $N = 288$? For full Trivium our computer ran out of memory before finishing gluing all equations together. On the other hand, we were able to glue 404 of the 666 initial equations together, producing a CRHS equation C_1 of size $2^{22.9}$. Then we glued the remaining initial equations into C_2, of size $2^{24.8}$. By using the upper bound (1) for merging two BDDs, we have then demonstrated that the single CRHS equation representing the full Trivium has a size smaller than $2^{47.7}$. The true size of the BDD for the full Trivium is probably

a lot smaller than $2^{47.7}$, given that the upper bound we use has proved to be very loose for the systems we study. In any case, we know that the size of the CRHS equation representing the full Trivium is quite far from the 2^{80}-bound for a valid attack.

4.1 Solving Attempts

If a single CRHS equation gave a solution as readily as a MRHS equation, we would be done, and have an algebraic attack on Trivium with complexity much smaller than the $O(2^{80})$-bound for exhaustive search. As noted above, we can not deduce a solution straight from the CRHS equation, since we have eventually to find a path in the BDD that satisfies a number of linear constraints. For scaled Triviums, we have of course tried the straight-forward approach mentioned in Section 3.1. Gluing BDDs representing linear constraints onto the BDD of the cipher CRHS equation unfortunately makes the size grow too large very rapidly.

Another solving method we have tried works as follows. Let the set of linear constraints to be satisfied be contained in a matrix LC. We set LC at the (single) top node in the BDD, and will propagate the matrix through the whole BDD according to Algorithm 3.

Algorithm 3 Propagating linear constraints through BDD with k levels.

```
for i = 0 to k do
    for every node a at level i do
        if a contains matrix then
            Build matrix M of linear constraints present in all matrices in a
            if l_i = 0 is consistent with M then
                Send M|_{l_i=0} through 0-edge
            end if
            if l_i = 1 is consistent with M then
                Send M|_{l_i=1} through 1-edge
            end if
        end if
    end for
end for
```

What we are bascally doing is to fix the value of l_i in LC to 0 or 1 when passing LC through a 0- or 1-edge out of a node at level i. If the linear constraints of LC would become inconsistent by sending it across an edge, the matrix is not propagated in that direction. Nodes receiving more than one LC-matrix will only keep linear constraints present in all matrices.

A node containing a matrix could be interpreted as saying *“Any path below me must satisfy the linear constraints in my matrix.”* We hope that the matrix ending up in the $\top$-node will contain some other linear constraints than the ones we started with. If this is the case, we can repeat Algorithm 3 with increasingly large LC.

In small examples (that can be checked by hand) the method of propagating the linear constraints through the BDD works, but for Trivium-35 it did not, as there were no new linear constraints in the matrix arriving at the bottom. What we did see for Trivium-35 however, was that there is a significant amount of nodes at levels $113 - 138$ in the BDD that did not receive any matrices (due to inconsistencies). At some levels almost half of the nodes were empty. We learn from this that there is no path satisfying the linear constraints in LC that can pass through these nodes, and so they can be deleted. Hence we can use Algorithm 3 to prune the BDD, and reduce its size.

5 Conclusion & Further Work

In this paper we have introduced a new way of representing algebraic equations, and shown its advantages compared to previously known representations. With the CRHS representation it is possible to merge many more equations together, than what is possible by other approaches. Building the CRHS equation system for Trivium, we have shown that Trivium may be described by a single CRHS equation with a BDD of size $2^{47.7}$ nodes, at most.

We have not yet been able to solve big CRHS equation systems, due to the many false solutions appearing in the right hand side BDD. The problem that needs to be solved is: **How do we efficiently find a path in a BDD that satisfies a set of linear constraints?** The method of matrix propagation helps in reducing the size of the BDD, and may be an approach worth pursuing. This is a topic for further research.

Finally, we should keep in mind that the operation of merging equations in a system is a process with exponential complexity. This is also true for CRHS equations, but for systems representing versions of Trivium we can do full merging anyway, because of the structure of the system. Solving non-linear equation systems is NP-hard in general, so we cannot hope to have a solving algorithm without any exponential step in it. Gluing all equations together is an exponential step, and full gluing normally solves the system. We can then speculate that after gluing all initial equations into one, we have overcome the exponential step and that the remaining problem for finding a solution can be solved efficiently. It is not clear that the problem of finding a path in a BDD subject to a set of linear constraints must have exponential complexity in the number of nodes. Further investigation into this question is needed.

References

1. McDonald, C., Charnes, C., Pieprzyk, J.: Attacking Bivium with MiniSat. eSTREAM, ECRYPT Stream Cipher Project, Report 2007/040 (2007) `http://www.ecrypt.eu.org/stream`.
2. Raddum, H., Semaev, I.: Solving Multiple Right Hand Sides linear equations. Designs, Codes and Cryptography **49**(1) (2008) 147–160
3. Faugère, J.: A new efficient algorithm for computing Gröbner bases (F4). Journal of Pure and Applied Algebra **139**(1-3) (1999) 61–88

4. Courtois, N., Alexander, K., Patarin, J., Shamir, A.: Efficient algorithms for solving overdefined systems of multivariate polynomial equations. Lecture Notes in Computer Science **1807** (2000) 392–407
5. Raddum, H.: MRHS Equation Systems. Lecture Notes in Computer Science **4876** (2007) 232–245
6. Semaev, I.: Sparse algebraic equations over finite fields. SIAM Journal on Computing **39**(2) (2009) 388–409
7. Akers, S.: Binary decision diagrams. IEEE Transactions on Computers **27**(6) (1978) 509–516
8. Somenzi, F.: Binary decision diagrams. In: Calculational System Design, volume 173 of NATO Science Series F: Computer and Systems Sciences, IOS Press (1999) 303–366
9. Knuth, D.: The Art of Computer Programming. Number Vol 4, Fascicles 0-4 in The Art of Computer Programming. ADDISON WESLEY (PEAR) (2009)
10. Krause, M.: BDD-based cryptanalysis of keystream generators. Lecture Notes in Computer Science **1462** (2002) 222–237
11. Stegemann, D.: Extended BDD-Based Cryptanalysis of Keystream Generators. Lecture Notes in Computer Science **4876** (2007) 17–35
12. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers **35** (1986) 677–691
13. Cannière, C.D., Preneel, B.: Trivium specifications. ECRYPT Stream Cipher Project (2005)
14. Somenzi, F.: CUDD: CU Decision Diagram Package. `http://vlsi.colorado.edu/~fabio/CUDD/` (2009)

Appendix

Example 1 (MRHS). The basic non-linear component in Trivium is the bitwise multiplication found in the function updating the registers. The new bit (x_6) coming into a register at some point is related to the old ones ($x_1, \ldots, x_5$) by

$$x_1 \cdot x_2 + x_3 + x_4 + x_5 = x_6.$$

The multiplication is the non-linear component, with inputs x_1 and x_2, and a single linear combination as output, namely $x_3 + x_4 + x_5 + x_6$. There are four different inputs to this function, hence there will be four columns in the B-matrix. The corresponding MRHS equation is

$$\begin{bmatrix} 1\,0\,0\,0\,0\,0 \\ 0\,1\,0\,0\,0\,0 \\ 0\,0\,1\,1\,1\,1 \end{bmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{pmatrix} = \begin{bmatrix} 0\,1\,0\,1 \\ 0\,0\,1\,1 \\ 0\,0\,0\,1 \end{bmatrix}. \tag{2}$$

Example 2 (BDD). Figure 1 shows an example BDD. The vertex v_0 is the root. Solid lines indicate 1-edges and dashed lines indicate 0-edges. In this example the order is (l_0, l_1, l_2) as indicated to the left.

Fig. 1. Example BDD

Example 3 (Accepted Inputs). The accepted inputs for the BDD in Figure 1 are the vectors (l_0, l_1, l_2):

$$\begin{gathered}(0,0,0)\\(0,0,1)\\(0,1,1)\\(1,0,1)\ .\end{gathered}$$

One can see that $(0,0,0)$ and $(0,0,1)$ are on the same path from v_0 to $\top$. On that path no node associated with l_2 is visited, so l_2 can be assigned both values.

A Boolean equation may be characterized by its set of satisfying assignments. Building a BDD whose accepted inputs match the set of satisfying assignments, gives us another representation of the same equation. For example, the Boolean equation corresponding to the BDD in Figure 1 is $l_0l_1 + l_0l_2 + l_1l_2 + l_0 + l_1 = 0$.

Example 4 (AND operation). The top half of Fig. 2 shows the BDDs of two Boolean functions. The left BDD shows $l_0 + l_1 + l_2 = 0$, the right BDD represents $l_0l_1 + l_2 = 0$. Both BDDs share the same order of variables, and the resulting BDD of their conjunction after reduction is shown below the two original BDDs.

Example 5 (CRHS). We write equation (2) from Example 1 as a CRHS equation by converting the right hand side into a BDD.

Instead of writing out the left hand matrix of equation (2), we write down the corresponding linear combinations, and give them the names l_0, l_1, l_2.

$$\begin{bmatrix} x_1 & = l_0 \\ x_2 & = l_1 \\ x_3 + x_4 + x_5 + x_6 & = l_2 \end{bmatrix} = \tag{3}$$

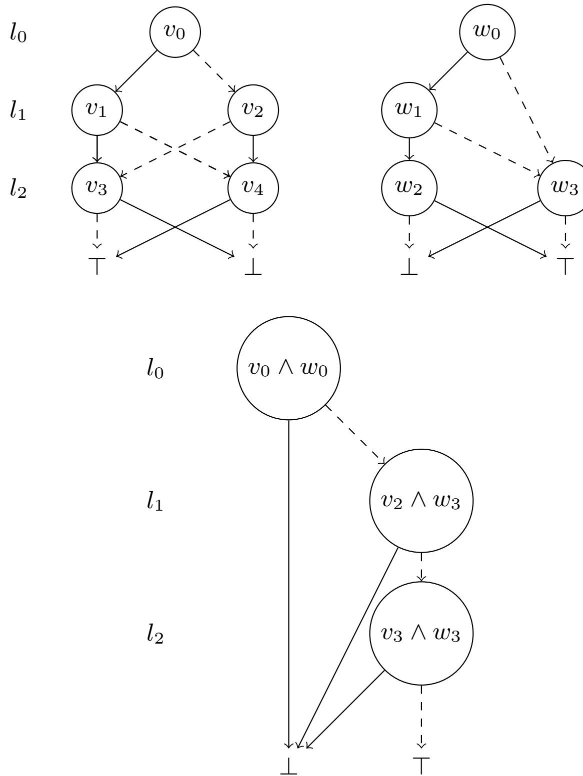


Fig. 2. AND-operation example

The right hand side of the CRHS equation is a *compressed* version of the right hand side in a MRHS equation. Every accepted input in the graph of the CRHS equation stands for one right hand side of the corresponding MRHS equation. The example above contains the edge (v_0, v_3). This edge is *jumping* over a level, i.e. every path through this edge does not contain any vertex at level l_1. That means that for a path containing the edge (v_0, v_3), the variable l_1 can take any value. The path $\langle v_0, v_3, \top \rangle$ thus contains two vectors for (l_0, l_1, l_2), namely $(0, 0, 0)$ and $(0, 1, 0)$.

Example 6 (CRHS Gluing). The following two equations are similar to equations in a Trivium equation system. In fact, the right hand sides of the following are taken from a full scale Trivium equation system. The left hand matrices have

been shortened.

$$\begin{bmatrix} x_1 & = l_0 \\ x_2 & = l_1 \\ x_3 + x_4 & = l_2 \end{bmatrix} = \begin{cases} l_0 & u_0 \\ l_1 & u_1 \\ l_2 & u_2 \quad u_3 \\ & \top \quad \bot \end{cases}, \begin{bmatrix} x_4 & = l_3 \\ x_5 & = l_4 \\ x_6 + x_7 & = l_5 \end{bmatrix} = \begin{cases} l_3 & v_0 \\ l_4 & v_1 \\ l_5 & v_2 \quad v_3 \\ & \top \quad \bot \end{cases} \tag{4}$$

The gluing of the equations above is

$$\begin{bmatrix} x_1 & = l_0 \\ x_2 & = l_1 \\ x_3 + x_4 & = l_2 \\ x_4 & = l_3 \\ x_5 & = l_4 \\ x_6 + x_7 & = l_5 \end{bmatrix} = \begin{cases} l_0 & w_0 \\ l_1 & w_1 \\ l_2 & w_2 \quad w_3 \\ l_3 & w_4 \\ l_4 & w_5 \\ l_5 & w_6 \quad w_7 \\ & \top \end{cases}, \tag{5}$$

where $\bot$-paths in this last graph are omitted for better readability. Note that omitting these paths does not decrease the overall number of vertices. The resulting equation has 8 nodes where the corredsponding MRHS equation would have 16 right hand sides.

Paper IV

5.4 Solving Compressed Right Hand Side Equation Systems with Linear Absorption

Thorsten E. Schilling, Håvard Raddum
Sequences and their Applications, Waterloo, to appear in Springer LNCS, (2012)

Solving Compressed Right Hand Side Equation Systems with Linear Absorption

Thorsten Ernst Schilling, Håvard Raddum
thorsten.schilling@ii.uib.no,havard.raddum@ii.uib.no

Selmer Center, University of Bergen

Abstract. In this paper we describe an approach for solving complex multivariate equation systems related to algebraic cryptanalysis. The work uses the newly introduced Compressed Right Hand Sides (CRHS) representation, where equations are represented using Binary Decision Diagrams (BDD). The paper introduces a new technique for manipulating a BDD, similar to swapping variables in the well-known sifting-method. Using this technique we develop a new solving method for CRHS equation systems. The new algorithm is successfully tested on systems representing reduced variants of Trivium.

Key words: multivariate equation system, BDD, algebraic cryptanalysis, Trivium

1 Introduction

Keystream generators produce pseudo-random sequences to be used in stream ciphers. A strong keystream generator must produce the sequence from a secret internal state such that it is very difficult to recover this initial state from the keystream. The security of a stream cipher corresponds to the complexity of finding the internal state that corresponds to some known keystream.

The relation between the keystream sequence and the internal state of the generator can be described as a system of algebraic equations. The variables in the system are the unknown bits of the internal state (at some time), and possibly some auxilliary variables. Solving the equation system will reveal the internal state of the generator, and hence break the associated stream cipher. Solving equation systems representing cryptographic primitives is known as algebraic cryptanalysis, and is an active research field.

This paper explores one approach for efficiently solving big equation systems, and is based on the work in [1], where the concept of Compressed Right Hand Side (CRHS) equations was introduced. A CRHS equation is a Binary Decision Diagram (BDD) together with a matrix with linear combinations of the variables in the system as rows. The problem of solving CRHS equation systems comes mainly from linear dependencies in the matrices associated with the BDD's. In this paper we introduce a new method for handling linear dependencies in CRHS equations, which we call *linear absorption*. The basis for linear absorption are

two methods for manipulating BDD's. One of them is the technique of swapping variables in the well-known sifting method [2]. The other is similar, but, to the best of our knowledge, not described in literature earlier. We call it *variable XOR.*

We have tested the method of linear absorption on systems representing scaled versions of Trivium [3]. We are able to break small versions of Trivium using linear absorption, proving that the method works. From these tests we derive an early estimate for the complexity of breaking the full Trivium using linear absorption. Our results indicate that the complexity of solving systems representing scaled Triviums increases with a factor $2^{0.4}$ each time the size of the solution space doubles.

2 Preliminaries

2.1 Binary Decision Diagrams

A Binary Decision Diagram (BDD) [4, 5] is a directed acyclic graph. BDDs were initially mostly used in design and verification systems. Later implementations and refinement led to a broader interest in BDDs and they were successfully applied in the cryptanalysis of LFSRs [6] and the cipher Grain [7]. For our purposes, we think of a BDD in the following way, more thoroughly described in [1].

A BDD is drawn from top to bottom, with all edges going downwards. There is exactly one node on top, with no incoming edges. There are exactly two nodes at the bottom, labelled $\top$ and $\bot$, with no outgoing edges. Except for $\top$ and $\bot$ each node has exactly two outgoing edges, called the 0-edge and the 1-edge. Each node (except for $\top$ and $\bot$) is associated to a variable. There are no edges between nodes associated to the same variable, which are said to be at the same *level.* An order is imposed on the variables. The node associated to the first variable is drawn on top, and the nodes associated to the last variable are drawn right above $\top$ and $\bot$. Several examples of BDDs are found in the following pages.

A path from the top node to either $\top$ or $\bot$ defines a vector on the variables. If node F is part of the path and is associated to variable x, then x is assigned 0 if the 0-edge is chosen out from F, and x is assigned 1 if the 1-edge is part of the path. A path ending in $\top$ is called an *accepted* input to the BDD.

There is a polynomial-time algorithm for reducing the number of nodes in a BDD, without changing the underlying function. It has been proven that a reduced BDD representing some function is unique up to variable ordering. In literature this is often referred to as a *reduced, ordered* BDD, but in this work we always assume BDDs are reduced, and that a call to the reduction algorithm is done whenever necessary.

2.2 Compressed Right Hand Side Equations

In [1] the concept of the Compressed Right Hand Side Equations was introduced. CRHS equations give a method for representing large non-linear constraints

along with algorithms for manipulating their solution spaces. In comparison to previous methods from the same family of algorithms [8–10] they offer an efficient way of joining equations with a very large number of solutions.

CRHS equations are a combination of the two different approaches *Multiple Right Hand Side Equations* [9] (MRHS equations) and BDDs. While MRHS equations were initially developed for cryptanalysis, BDDs were developed for other purposes. Combining the two provides us with a powerful tool for algebraic cryptanalysis. For instance, using CRHS equations it is possible to create a single large BDD representing the equation system given by the stream cipher Trivium.

Definition 1 (CRHS Equation [1]). *A compressed right hand side equation is written as $Ax = \mathcal{D}$, where A is a binary $k \times n$-matrix with rows $l_0, \ldots, l_{k-1}$ and $\mathcal{D}$ is a BDD with variable ordering (from top to bottom) $l_0, \ldots, l_{k-1}$. Any assignment to x such that Ax is a vector corresponding to an accepted input in $\mathcal{D}$, is a satisfying assignment. If C is a CRHS equation then the number of vertices in the BDD of C, excluding terminal vertices, is denoted $\mathcal{B}(C)$.*

Example 1 (CRHS Equation). In order to write:

$$f(x_1, \ldots, x_6) = x_1x_2 + x_3 + x_4 + x_5 + x_6 = 0$$

as a CRHS equation one chooses a name for every linear component in $f(x_1, \ldots, x_6) = 0$. Here we decide to name the linear components $l_0 = x_1, l_1 = x_2, l_2 = x_3 + x_4 + x_5 + x_6$. Furthermore one needs to define an ordering on these linear components. For this example we we select the order l_0, l_1, l_2, from top to bottom.

The matrix A formed by the linear components is then our left hand side of the CRHS equation. The BDD formed by the possible values of l_0, l_1, l_2 in $f(x_1, \ldots, x_6) = 0$ together with the before defined order forms the right hand side of the CRHS equation.

The resulting CRHS equation is then:

$$\begin{bmatrix} x_1 & = l_0 \\ x_2 & = l_1 \\ x_3 + x_4 + x_5 + x_6 & = l_2 \end{bmatrix} = \left\{ \begin{array}{lll} l_0 & v_0 & \\ l_1 & v_1 & \\ l_2 & v_2 & v_3 \\ & \top & \bot \end{array} \right. . \qquad (1)$$

The right hand side of the CRHS equation represents the possible values of l_0, l_1, l_2 in $f(x_1, \ldots, x_6) = 0$ in *compressed* form. The set of solutions of (1) is the union of all solutions of $Ax = L$, where L is a vector contained in the right hand side as an accepted input to the BDD. Naming equation (1) as E_0, we have $\mathcal{B}(E_0) = 4$.

2.3 Joining CRHS equations

Given two CRHS equations A and B it is natural to ask: *What are the common solutions to A and B?*

In [1] an algorithm, called *CRHS Gluing* is introduced. The algorithm takes as input two CRHS equations and has as output a new CRHS equation which contains the solutions of the conjunction of the input. This algorithm is exponential in space and time consumption. Nevertheless, the constant of this exponential has been shown to be small enough for practical applications.

Here, we use a simpler and cheaper method of joining two CRHS equations. Given two BDDs $\mathcal{D}_1$ and $\mathcal{D}_2$, the notation $(\mathcal{D}_1 \rightarrow \mathcal{D}_2)$ is defined to simply mean that $\top$ in $\mathcal{D}_1$ is replaced with the top node in $\mathcal{D}_2$. The two $\bot$-nodes from $\mathcal{D}_1$ and $\mathcal{D}_2$ are merged into one $\bot$, and the resulting structure is a valid BDD.

Given the two CRHS equations $[L_1]x = \mathcal{D}_1$ and $[L_2]x = \mathcal{D}_2$ the result of joining them is

$$\begin{bmatrix} L_1 \\ L_2 \end{bmatrix} x = (\mathcal{D}_1 \rightarrow \mathcal{D}_2)$$

Any accepted path in $(\mathcal{D}_1 \rightarrow \mathcal{D}_2)$ gives accepted paths in both $\mathcal{D}_1$ and $\mathcal{D}_2$. In other words, any x such that $\begin{bmatrix} L_1 \\ L_2 \end{bmatrix} x$ yields an accepted path in $(\mathcal{D}_1 \rightarrow \mathcal{D}_2)$ gives solutions to the two initial CRHS equations.

When there are linear dependencies among the rows in $\begin{bmatrix} L_1 \\ L_2 \end{bmatrix}$ we get paths in $(\mathcal{D}_1 \rightarrow \mathcal{D}_2)$ that lead to false solutions. The problem of false solutions is the only problem preventing us from having an efficient solver for CRHS equation systems. This problem is addressed in Section 3.3.

Example 2 (Joining CRHS equations). The following two equations are similar to equations in a Trivium equation system. In fact, the right hand sides of the following are taken from a full scale Trivium equation system. The left hand matrices have been shortened.

$$\begin{bmatrix} x_1 & = l_0 \\ x_2 & = l_1 \\ x_3 + x_4 & = l_2 \end{bmatrix} = \left\{ \begin{array}{lccc} l_0 & u_0 & & \\ l_1 & u_1 & & \\ l_2 & u_2 & u_3 & \\ & \top & \bot & \end{array} \right. , \quad \begin{bmatrix} x_4 & = l_3 \\ x_5 & = l_4 \\ x_6 + x_7 & = l_5 \end{bmatrix} = \left\{ \begin{array}{lccc} l_3 & v_0 & & \\ l_4 & v_1 & & \\ l_5 & v_2 & v_3 & \\ & \top & \bot & \end{array} \right. \qquad (2)$$

The joining of the equations above is

$$\begin{bmatrix} x_1 & = l_0 \\ x_2 & = l_1 \\ x_3 + x_4 & = l_2 \\ x_4 & = l_3 \\ x_5 & = l_4 \\ x_6 + x_7 & = l_5 \end{bmatrix} = \left\{ \begin{array}{lll} l_0 & w_0 & \\ l_1 & w_1 & \\ l_2 & w_2 & w_3 \\ l_3 & w_4 & \\ l_4 & w_5 & \\ l_5 & w_6 & w_7 \\ & \top & \end{array} \right. , \qquad (3)$$

where $\perp$-paths in this last graph are omitted for better readability. The resulting equation has 8 nodes, where the corredsponding MRHS equation would have 16 right hand sides.

Joining two CRHS equations E_0 and E_1 is really nothing more than putting one on top of the other and connect them. If E_0 and E_1 are joined to form E, it is easy to see that $\mathcal{B}(E) = \mathcal{B}(E_0) + \mathcal{B}(E_1)$. The complexity of joining CRHS equations is linear, and we can easily build a single CRHS equation representing, for instance, the full Trivium. The CRHS equation representing the full Trivium will have less than 3000 nodes, but 2^{1336} paths in the long BDD, of which maybe only one is not a false solution.

3 Solving Large CRHS Equation Systems

After joining several CRHS equations together the left hand side of the resulting equation may contain linear dependencies which are not reflected in the right hand side BDD. The matrix of the CRHS equation contains rows which sum to 0. The BDD on the other hand is oblivious to this fact and contains paths which sum to 1 on the affected variables.

Since the set of solutions of the CRHS equation is the union of solutions to the individual linear systems formed by each vector of the right hand side, we need to filter out those vectors which yield an inconsistent linear system. Let for example the left hand side of a CRHS equation contain the linear combinations l_i, l_j and l_k and assume we found that $l_i + l_j + l_k = 0$. The BDD might nevertheless

contain a path which assigns l_i, l_k and l_k to values that make their sum equal to 1. Since we know that this path in the BDD does not correspond to a solution we would like to eliminate it from the BDD.

In practical examples from the cryptanalysis of Trivium we end up with the situation that almost all paths on the right hand side are of this kind, i.e., not corresponding to the left hand side. The major problem is that we cannot easily *delete a path* by some simple operation, e.g., deleting a node. This is because there are many paths passing through a single node.

In order to delete all invalid solutions from a CRHS equation, we introduce the techniques *Variable XOR* and *Linear Absorption* in the following. They are new methods for the manipulation of BDDs and can be used to take care of removing paths which correspond to false solutions.

3.1 Variable Swap

A usual operation on a BDD is to swap variable levels [2] while preserving the function the BDD represents. This means to change the permutation of variables in a BDD by exchanging adjacent positions of two variables. This is done for example to change the size of a specific BDD. We will use this technique in the following and give a short introduction to it.

The origins of the BDD data structure lie within the *Shannon Expansion* [11]. In the following let be $F = f(x_0, \ldots, x_{n-1})$, $F_{x_r} = f(x_0, \ldots, x_{r-1}, 1, x_{r+1}, \ldots, x_{n-1})$ and $F_{\overline{x_r}} = f(x_0, \ldots, x_{r-1}, 0, x_{r+1}, \ldots, x_{n-1})$. Then by the Shannon expansion every Boolean function can be represented in the form

$$F = x \cdot F_x + \overline{x} \cdot F_{\overline{x}}. \tag{4}$$

We write the function as a BDD with the root node denoted $F = (x, F_x, F_{\overline{x}})$. Here x is the variable defining the level of the node, F_x is the node connected through the 1-edge and $F_{\overline{x}}$ is the node connected to the 0-edge. F_x and $F_{\overline{x}}$ are called the co-factors of the node F.

Let the variable coming after x in the variable order be y. To expand (4) by the variable y, we have to expand the subfunctions F_x and $F_{\overline{x}}$ accordingly:

$$F = x \cdot (y \cdot F_{xy} + \overline{y} \cdot F_{x\overline{y}}) + \overline{x} \cdot (y \cdot F_{\overline{x}y} + \overline{y} \cdot F_{\overline{xy}}). \tag{5}$$

Again, as a root node of a BDD we have $F = (x, (y, F_{xy}, F_{x\overline{y}}), (y, F_{\overline{x}y}, F_{\overline{xy}}))$ but this time with explicitly written co-factors. Assume we would like to swap the order of x and y. Then we can equivalently write (5) as

$$F' = y \cdot (x \cdot F_{xy} + \overline{x} \cdot F_{\overline{x}y}) + \overline{y} \cdot (x \cdot F_{x\overline{y}} + \overline{x} \cdot F_{\overline{xy}}) \tag{6}$$

which leads us to the new node representation of $F' = (y, (x, F_{xy}, F_{\overline{x}y}), (x, F_{x\overline{y}}, F_{\overline{xy}}))$. Now the order of the variables x and y is swapped. Since (5) and (6) are equivalent so are our BDD nodes before and after the swap.

Moreover, it becomes clear that swapping two variables is a local operation, in the sense that only nodes at levels x and y are affected. If one would like to

swap the levels x and y (where as above x is before y in the BDD permutation) one has to apply the operation above to every node at level x and change it accordingly.

Example 3 (Variable Swap).

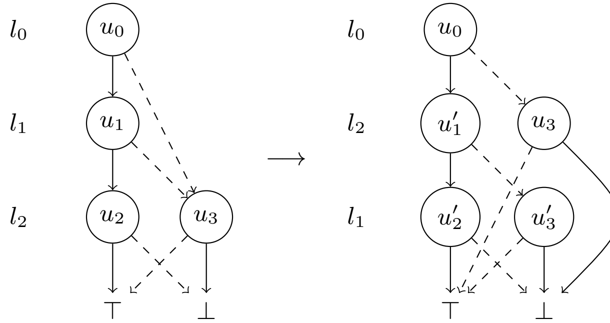


Fig. 1. Swapping l_1 and l_2.

On the left side in Fig. 1 a BDD along with its permutation (l_0, l_1, l_2) is depicted. In order to swap levels l_1 and l_2, i.e., change the permutation to (l_0, l_2, l_1), one has to apply the swapping routine described above to all nodes at level l_1. In this case $u_1 = (l_1, u_2, u_3)$ is the only node affected. With explicitly written co-factors we get $u_1 = (l_1, (l_2, \top, \bot), (l_2, \bot, \top))$. From the swapping procedure above we know that the resulting new node is $u_1' = (l_2, (l_1, \top, \bot), (l_1, \bot, \top)) = (l_2, u_2', u_3')$. Node u_3 stays unchanged.

3.2 Variable XOR

In this section we introduce a new method for manipulating BDDs, the *variable XOR* operation. As the name suggests, we change a variable by XORing a different variable onto it. To preserve the original function we have to change the BDD accordingly. Below we explain how this is done. In fact, the procedure is quite similar to Variable Swap, and is only a local operation.

Let x and y be two consecutive BDD variables (x before y) and $\sigma = x + y$. We want to transform (5) into:

$$F' = x \cdot (\sigma \cdot F_{x\sigma} + \overline{\sigma} \cdot F_{x\overline{\sigma}}) + \overline{x} \cdot (\sigma \cdot F_{\overline{x}\sigma} + \overline{\sigma} \cdot F_{\overline{x}\overline{\sigma}}). \tag{7}$$

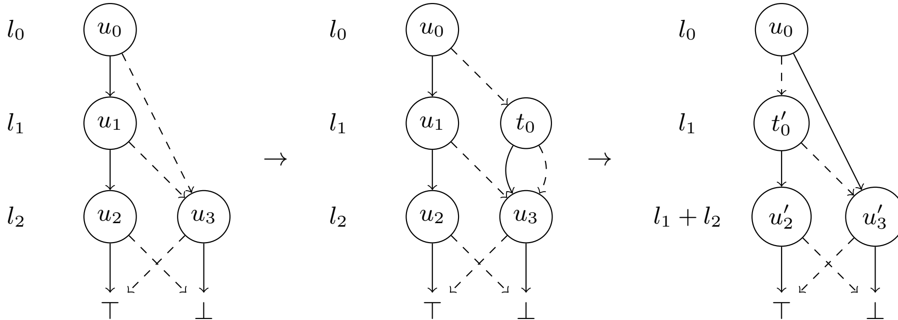
We can see that if $x = 1$ then $F_{x\sigma} = F_{x\overline{y}}$ and $F_{x\overline{\sigma}} = F_{xy}$. Similarly if $x = 0$ then $F_{\overline{x}\sigma} = F_{\overline{x}y}$ and $F_{\overline{x}\overline{\sigma}} = F_{\overline{x}\overline{y}}$. With that in mind (7) can be written as

$$F' = x \cdot (\sigma \cdot F_{x\overline{y}} + \overline{\sigma} \cdot F_{xy}) + \overline{x} \cdot (\sigma \cdot F_{\overline{x}y} + \overline{\sigma} \cdot F_{\overline{x}\overline{y}}) \tag{8}$$

which leads immediately to the new node representation
$F' = (x, (\sigma, F_{x\overline{y}}, F_{xy}), (\sigma, F_{\overline{x}y}, F_{\overline{x}\overline{y}}))$. With this manipulation extra care has to

be taken of edges incoming to nodes at the y-level that *jumps* over the x-level. Here temporary nodes have to be introduced since y goes over into σ and cannot longer be *addressed* directly.

Example 4 (Variable XOR).



The first diagram shows the initial BDD in which the variable levels l_1 and l_2 are to be XORed. The second diagram represents how the auxilliary node t_0 needs to be introduced since the edge (u_0, u_3) ignores the l_1 level. Then the variable XOR procedure is applied to both u_1 and t_0, and the resulting BDD is reduced. After the application of the modification of equation (5) to (7) the result of the variable XOR method to variables l_1 and l_2 of the initial diagram is depicted.

3.3 Linear Absorption

We are now ready to explain the method of linear absorption.

Assume we have a BDD with $(l_0, \ldots, l_{k-1})$ as the ordered set of linear combinations associated with the levels. We can easily find all linear dependencies among the l_i's. Assume that we have found the dependency $l_{i_1} + l_{i_2} + \ldots + l_{i_r} = 0$, where $i_1 < i_2 < \ldots < i_r$.

By using variable swap repeatedly, we can move the linear combination l_{i_1} down to the level just above l_{i_2}. Then we use variable XOR to replace l_{i_2} with $l_{i_1} + l_{i_2}$. Next, we use varlable swap again to move $l_{i_1} + l_{i_2}$ down to the level just above l_{i_3}, and variable XOR to replace l_{i_3} with $l_{i_1} + l_{i_2} + l_{i_3}$. We continue in this way, picking up each l_{i_j} that is part of the linear dependency, until we replace l_{i_r} with $l_{i_1} + l_{i_2} + \ldots + l_{i_r}$. Let us call the level of nodes associated with $l_{i_1} + l_{i_2} + \ldots + l_{i_r}$ for the zero-level.

We know now that the zero-level has the 0-vector associated with it. This implies that any path in the BDD consistent with the linear constraint we started with has to select a 0-edge out of a node on the zero-level. In other words, all

1-edges going out from this level lead to paths that are inconsistent with the linear constraint $l_{i_1} + l_{i_2} + \ldots + l_{i_r} = 0$, and can be deleted.

After deleting all outgoing 1-edges, there is no longer any choice to be made for any path going out from a node at the zero-level. If F is a node at the zero-level, any incoming edge to F can go directly to F_0, jumping the zero-level altogether. After all incoming edges have been diverted to jump the zero-level, all nodes there can be deleted, and the number of levels in the BDD decreases by one. We are now certain that any path in the remaining BDD will never be in conflict with the constraint $l_{i_1} + l_{i_2} + \ldots + l_{i_r} = 0$; we say that the linear constraint has been absorbed.

We can repeat the whole process, and absorb one linear constraint at the time, until all remaining l_i are linearly independent. At that point, any remaining path in the BDD will yield a valid solution to the initial equation system.

4 Experimental Results

We have tested Linear Absorption on equation systems representing scaled versions of Trivium.

4.1 Trivium & Trivium-N

Trivium is a synchronous stream cipher and part of the ECRYPT Stream Cipher Project portfolio for hardware stream ciphers. It consists of three connected nonlinear feedback shift registers (NLFSR) of lengths 93, 84 and 111. These are all clocked once for each keystream bit produced.

Trivium has an inner state of 288 bits, which are initialized with 80 key bits, 80 bits of IV, and 128 constant bits. The cipher is clocked 1152 times before actual keystream generation starts. The generation of keystream bits and updating the registers is very simple. For algebraic cryptanalysis purposes one can create four equations for every clock; three defining the inner state change of the registers and one relating the inner state to the keystream bit. Solving this equation system in time less than trying all 2^{80} keys is considered a valid attack on the cipher.

Small Scale Trivium. In [1] a reduced version of Trivium, called Trivium-N was introduced. N is an integer value which defines the size of the inner state of that particular version of Trivium. Trivium-288 is by our construction equivalent to the originally proposed Trivium.

All versions of Trivium-N with $N < 288$ try to preserve the structure of the original Trivium as well as possible. This yields equation systems which are comparable to the full cipher. Other small scale version of Trivium e.g., Bivium [12], in which an entire NLFSR was removed, seems to be too easy to solve.

4.2 Results

We have constructed CRHS equation systems representing Trivium-N for several values of N, and run the algorithm for absorbing linear constraints described in Section 3.3. For $N \leq 41$ we were able to absorb all linear constraints, which means that any remaining path in the BDD is a valid solution to the system (we have also verified this).

The number of nodes in the BDD grows very slowly when absorbing the first linear constraints, but increases more rapidly when the linear constraints of length two have been absorbed. We know, however, that the number of paths will be very small once all linear constraints have been absorbed since we expect a unique, or very few, solution(s). Thus the number of nodes must also decrease quickly after the number of absorbed constraints is past some tipping point. For each instance we have recorded the maximum number of nodes the BDD contained during execution, and used this number as our measure of complexity. The memory consumtion is dominated by the number of nodes, and in our implementation each node took 60 bytes. The memory requirement in bytes can then be found approxiamtely by multiplying the number of nodes with 60.

The results for testing the algorithm on Trivium-N for $30 \leq N \leq 41$ is written below.

N	max. # of nodes
30	$2^{19.92}$
31	$2^{21.02}$
32	$2^{21.15}$
33	$2^{20.84}$
34	$2^{21.41}$
35	$2^{22.32}$
36	$2^{21.61}$
37	$2^{23.27}$
38	$2^{23.49}$
39	$2^{23.79}$
40	$2^{23.69}$
41	$2^{24.91}$

The number of solutions (paths) in each instance was found to be between 1 and 3. The number of levels in the final BDD was 73 for $N = 30$, and 97 for $N = 41$.

The numbers above have been produced using only a single test for each N. We can expect some variation in the maximum number of nodes when re-doing tests using different initial states for some particular Trivium-N. The numbers are plotted in Fig. 2 to show the general trend in the increase of complexity.

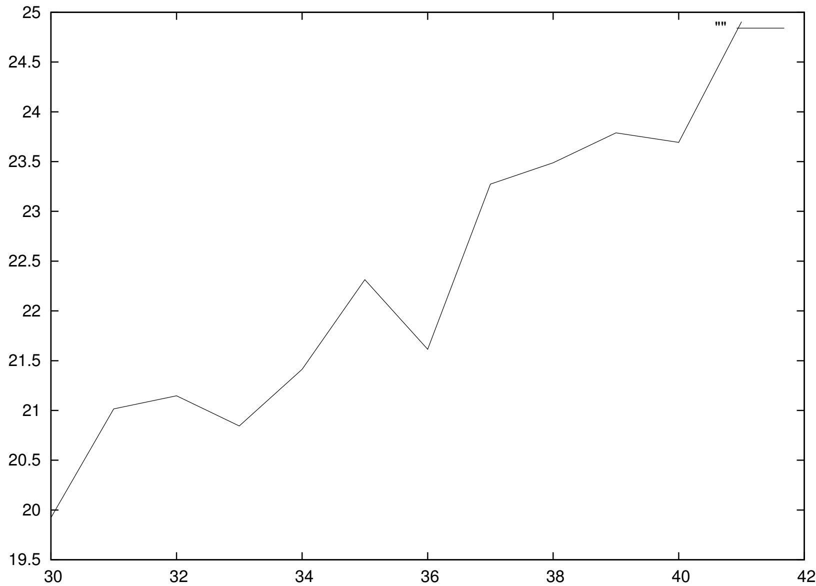


Fig. 2. Trend of complexities for Trivium-N

4.3 Extrapolating

We can use the least-square method to fit a linear function to the data points we have. Letting 2^M be the maximum number of nodes needed, the linear function that best approximates our data is $M = 0.4N + 7.95$.

When N increases by 1, the size of the solution space for the variables in the initial state doubles. However, the total number of variables in the system increases by three when N increases by 1. This is because we need to clock the cipher one step further to have enough known keystream for a unique solution, and each clock introduces three new variables. Hence we can say that the size of the problem instance increases by a factor 2^3 for each increase in N. The complexity of our solving method only increases with a factor of approximately $2^{0.4}$ on the tested instances, which we think is quite promising.

Admittedly, we have too little data to draw any clear conclusions, but it is still interesting to see what value of M we get for $N = 288$. Based on the data we have, we find that currently we need to be able to handle around 2^{123} nodes in a BDD for successfully attacking the full Trivium.

5 Conclusions and Future Work

We have introduced how to alter a BDD to preserve the underlying function when two variables are XORed. Together with variable swap, we have introduced a new solving method in algebraic cryptanalysis, which we call linear absorption. The solving technique works on equations represented in CRHS form.

The work in this paper gives more insight into how to solve some of the open questions in [1], and provides a complete solving method. We have shown how the method works on systems representing scaled versions of Trivium. The structure of the equations is exactly the same in the down-scaled and the full versions of Trivium, it is only the number of equations and variables that differ. Our tests thus gives some information on the complexity of a successful algebraic attack on the full Trivium.

Unfortunately, we have not had the time to test linear absorption on other ciphers, or test more extensively on Trivium-N. This is obviously a topic for further research. We also hope to further investigate the problem of how to find a path in a BDD that satisfies a set of linear constraints. There may be tweaks to the algorithm of linear absorption, or there may be a completely different and better method. In any case, we hope to see more results on solving methods for CRHS equation systems.

References

1. Schilling, T.E., Raddum, H.: Analysis of trivium using compressed right hand side equations. 14th International Conference on Information Security and Cryptology, Seoul, Korea, November 30 - December 2, 2011, to appear in Lecture Notes in Computer Science (2011)
2. Rudell, R.: Dynamic variable ordering for ordered binary decision diagrams. Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design **12** (1993) 42–47
3. Cannière, C.D., Preneel, B.: Trivium specifications. ECRYPT Stream Cipher Project (2005)
4. Akers, S.: Binary decision diagrams. IEEE Transactions on Computers **27**(6) (1978) 509–516
5. Somenzi, F.: Binary decision diagrams. In: Calculational System Design, volume 173 of NATO Science Series F: Computer and Systems Sciences, IOS Press (1999) 303–366
6. Krause, M.: Bdd-based cryptanalysis of keystream generators. CRYPTO '98, Lecture Notes in Computer Science **1462** (1998) 222–237
7. Stegemann, D.: Extended BDD-Based Cryptanalysis of Keystream Generators. Selected Areas in Cryptography 2007, Lecture Notes in Computer Science **4876** (2007) 17–35
8. Raddum, H.: MRHS Equation Systems. Selected Areas in Cryptography 2007, Lecture Notes in Computer Science **4876** (2007) 232–245
9. Raddum, H., Semaev, I.: Solving multiple right hand sides linear equations. Designs, Codes and Cryptography **49**(1) (2008) 147–160
10. Schilling, T.E., Raddum, H.: Solving equation systems by agreeing and learning. Proceedings of Arithmetic of Finite Fields, WAIFI 2010, Lecture Notes in Computer Science **6087** (2010) 151–165
11. Shannon, C.E.: The synthesis of two-terminal switching circuits. Bell Systems Technical Journal **28** (1949) 59–98
12. McDonald, C., Charnes, C., Pieprzyk, J.: Attacking Bivium with MiniSat. eSTREAM, ECRYPT Stream Cipher Project, Report 2007/040 (2007) `http://www.ecrypt.eu.org/stream`.

Chapter 6

Other Results

This chapter contains two other results which did not fit in any other publication. They represent new approaches either on how to solve the main problem studied in this thesis or to reduce the technical complexity of the task. As opposed to the other scientific results in this thesis the ones presented in this chapter are *works in progress*.

The first result might make the main approach used in this thesis easier to understand. Primarily for those working more in the fields of algorithms and graph theory, the presented method might transfer the theme of cryptanalysis into a more familiar realm. Due to the relatively intuitive structure of the presented reduction of the problem, one can say that it bridges the two areas.

The second result represents more of a technical improvement. One algorithm is incorporated into another by preprocessing the input of the former. By doing this, additional information can be obtained which would otherwise require a whole separate processing from a second algorithm. This reduces the implementation complexity of a potential solving algorithm.

6.1 Independent Set Reduction

One technique to show the NP-hardness of a problem A is to give a *deterministic polynomial time reduction* of another problem B, known to be NP-hard, to A. By such a reduction we mean a function f, which takes as input a problem instance $b \in B$ and has as output a problem instance of A. That is a function $f : B \to A$ which can be computed in polynomial time and $f(b) \in A \Leftrightarrow b \in B$.

Such a reduction demonstrates that problem A is at least as hard to solve as B since all instances of B can be solved if we can solve A. Reductions which satisfy these requirements are also called Karp- or Cook-reductions [10, 28, 41] and are in the following denoted by $B \leq_p A$.

Since those reductions can be done in deterministic polynomial time, the asymptotical running time of B cannot be larger than that of A, and therefore the complexity of A is dominant.

The problem of solving a set of equations is almost intuitively NP-complete since there exists an obvious relation to SAT solving. Other decision problems – such as the INDEPENDENT SET-problem – belong to one of the first problems shown to be NP-complete [46], but might need a proper formal reduction for clarification.

In the following we will show an alternative reduction of the decision version of the Gluing/Agreeing algorithm to the INDEPENDENT SET-problem (IS-problem). Not for the purpose of demonstrating (again) the *NP*-completeness of the IS-problem, but rather to provide an alternative formulation of the Gluing/Agreeing in the context of graph theory. Along with it, we will present some remarks on the structure of the resulting graphs. A similar reduction of the SAT-problem $\leq_p$ IS-problem along with more details on polynomial time reductions and complexity theory in general can be found in [46].

Then we will present an algorithm by Fomin et. al. [22] to solve the IS-problem on instances yielded by random equation systems and/or by a cipher. Finally, we will present some experimental results and conclusions.

To begin, we define the problem of finding a solution to a set of equations as a decision problem of the Gluing/Agreeing algorithm.

Definition 6 (GA-problem)**.** *Let*

$$\mathcal{S} = \{S_0, S_1, \ldots, S_{m-1}\} \tag{6.1}$$

be an input to the Gluing/Agreeing algorithm where $S_i = (X_i, V_i)$ *is a symbol consisting of a set of variables* X_i *and a set of satisfying vectors* $V_i = \{v_0, v_1, \ldots, v_{r-1}\}$ *defined in* X_i.

The decision problem is then to ask if there exists a set of vectors $\mathcal{V} \subseteq V_0 \cup V_1 \cup \ldots \cup V_{m-1}$ *such that*

1. *for each* V_i *it holds that* $|V_i \cap \mathcal{V}| = 1$, *and*
2. *for each* $a, b \in \mathcal{V}$ *with* $a \in V_i$ *and* $b \in V_j$ *it is true for* $X_{i,j} = X_i \cap X_j$ *that* $a[X_{i,j}] = b[X_{i,j}]$.

It is easy to see that every such set $\mathcal{V}$ satisfying both conditions in definition 6 constitutes a solution to the input of the Gluing/Agreeing algorithm.

We want to reduce the GA-problem to the well known k-IS-problem. Again, we will give a definition of the problem as the decision version.

Definition 7 (k-IS-problem)**.** *Given a graph* $\mathcal{G} = (V, E)$ *with vertices* V, *edges* E *and a non-negative integer* k, *does there exist a subset* $\mathcal{I} \subseteq V$ *of size* k *such that* $\mathcal{I}$ *is an independent set, i.e.,* $\mathcal{I}$ *induces an edgeless subgraph on* $\mathcal{G}$?

6.1.1 Reduction

Now we will introduce our new reduction GA-problem $\leq_p$ IS-problem. That is a method to transform an input to the GA-problem to an input of the k-IS-problem. Then we will show that this reduction is correct, i.e., the instance of the GA-problem with m symbols has a solution if and only if the resulting k-IS-problem has a solution with $k = m$ and that the reduction can be done in polynomial time.

The reduction consists of two steps:

Step 1 First we create an empty undirected graph $\mathcal{G} = (V, E)$. For every V_i in (6.1) we construct a clique consisting of vertices labeled by vectors in V_i and insert them into $\mathcal{G}$. That is for every vector $v \in V_i$ we insert a vertex labeled v into V and connect the vertex labeled by v to all other vertices with labels in V_i.

Step 2 After *Step 1* there exists for every vector v in $V_0 \cup V_1 \cup \ldots \cup V_{m-1}$ a corresponding vertex in $\mathcal{G}$ labeled v. For all pairs of symbols S_i, S_j with $X_{i,j} = X_i \cap X_j \neq \emptyset$ we compare every pair of vectors $u \in V_i, w \in V_j$. If $u[X_{i,j}] \neq w[X_{i,j}]$ we insert the edge uw into $\mathcal{G}$.

Lemma 1. *$\mathcal{G}$ has an independent set of size m, if and only if $\mathcal{S}$ has a solution. Furthermore, the reduction above can be done in polynomial time.*

Proof. We will show that the vertices of $\mathcal{G}$ which form an independent set of size m are, as their corresponding vectors, a solution set for $\mathcal{S}$.

($\Rightarrow$) Assume $\mathcal{G}$ has an independent set $\mathcal{I}$ of size m. By *Step 1* $\mathcal{G}$ consists of m cliques, so $\mathcal{I}$ can contain at most 1 vertex of every clique. Since all vertices labeled by vectors in V_i form a clique, the first requirement of the GA-problem is met, namely that at most one vector per V_i is selected.

Furthermore, since by *Step 2* all vertices whose corresponding vectors are unequal in their projection on common variables are connected, $\mathcal{I}$ can only contain those vertices whose corresponding vectors are equal in their projection. That satisfies the second requirement of the GA-problem.

($\Leftarrow$) Assume $\mathcal{S}$ has a solution $\mathcal{V}$. Then we can form an independent set $\mathcal{I}$ in $\mathcal{G}$ by selecting all vertices with labels in $\mathcal{V}$. This is an independent set of size m in $\mathcal{G}$ since only one vertex per clique is selected (by construction in *Step 1*). Furthermore none of these vertices share an edge, since by *Step 2* only those vertices are connected whose corresponding vectors are unequal in their projection of common variables. Thus, if $\mathcal{I}$ would not be an independent set of size m, $\mathcal{V}$ would not be a solution to $\mathcal{S}$, which contradicts the assumption.

Step 1 of the reduction can be done in $\mathcal{O}(m \cdot |V|_{max}{}^2)$ where $|V|_{max}$ is the maximum number of vectors in any V_i. *Step 2* is comparing each pair of vectors for each pair of symbols and therefore runs in $\mathcal{O}(m^2 \cdot |V|_{max}{}^2)$. The whole reduction can therefore be done in $\mathcal{O}(m^2 \cdot |V|_{max}{}^2)$. □

Example 6 (Reduction example). Let

S_0	x_0	x_1	x_2	x_3
a_0	0	0	0	1
a_1	0	1	0	0
a_2	1	0	1	0

,

S_1	x_2	x_3	x_4	x_5
b_0	0	0	0	1
b_1	0	1	0	0
b_2	1	0	1	0

,

S_2	x_0	x_1	x_4	x_5
c_0	0	0	0	1
c_1	0	1	0	0
c_2	1	0	1	0

(6.2)

be a given GA-problem instance. We will apply the two-step reduction explained above in order to transform S_0, S_1, S_2 into an IS-problem instance. In *Step 1* we create a clique for every symbol.

The only possible independent set of size $k = 3$ in Step 2 of Figure 6.1 is indicated in orange. The only solution to equations (6.2) is therefore the combination of vectors a_2, c_2, b_2 which is written $(1,0,1,0,1,0)$ in the variables $(x_0, x_1, x_2, x_3, x_4, x_5)$.

6.1.2 Graph Structure

Graphs resulting from this reduction are different from random graphs, even when the input was a random equation system. Let us assume that we know the following about the input instance:

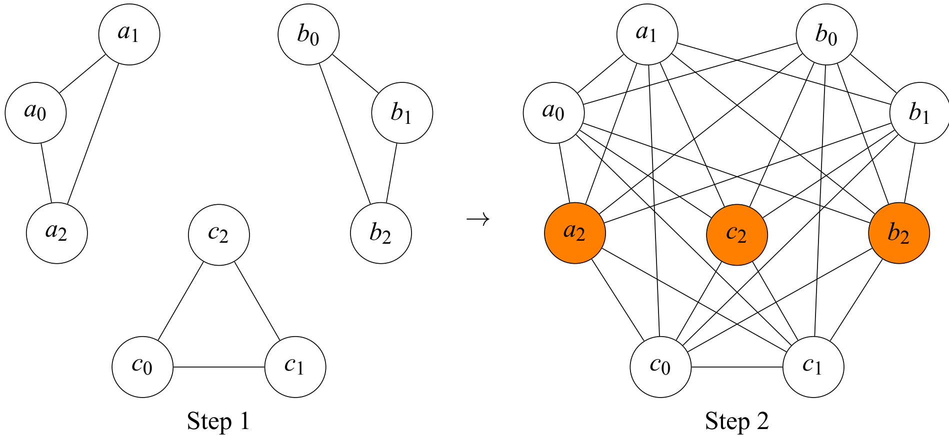


Figure 6.1: 2-Step Reduction to IS-problem

1. The equation system is over $\mathbb{F}_q$,
2. in m equations,
3. with a sparsity of $\leq l$,
4. all equations are pair-wise agreeing,
5. and the system contains no trivial partial solutions, i.e., fixed variables.

Then we can make the following observations about the resulting graph:

Lemma 2. *The graph will contain m complete subgraphs. I.e., for each* (X_i, V_i) *with* $r = |V_i|$ *there exists the complete subgraph* K_r *in* $\mathcal{G}$.

Proof. By construction. □

Lemma 3. *For any pair of complete subgraphs* K_r^i, K_s^j *containing r and s nodes, respectively in* $\mathcal{G} = (V, E)$ *let* η_{ij} *be the size of the cut-set between* K_r^i *and* K_s^j*, i.e.,* $\eta_{ij} = |\{uv \in E \mid u \in K_r^i \text{ and } v \in K_s^j\}|$*. Then either* $\eta_{ij} = 0$ *or* $r \leq \eta_{ij} \leq r(s-1)$*,* $r \leq s$*.*

Proof. If two pair-wise agreeing equations have no common variables, their vectors cannot disagree – hence their cliques have no common edges, i.e., $\eta_{ij} = 0$. If they on the other hand have common variables, we know that they must have vectors that differ in these variables since by assumption there are no trivial partial solutions. Each vector in K_r^i therefore has to be unequal to at least one vector in K_s^j, i.e., $r \leq \eta_{ij}$. Since the equations are agreeing pair-wise we also know that no vector can be unequal to all vectors of K_s^j and therefore $\eta_{ij} \leq r(s-1)$. □

This casts some constraints on the structure of the resulting graph after the transformation, even if the input equation system was purely random (and then pair-wise agreed). Unfortunately, it is unclear at this point if this observations can give any advantage when implementing a solving algorithm and the question remains an open problem.

6.1.3 IS-Solving Algorithm

We have now established a method to reduce the problem of solving a set of equations to the problem of finding a k-independent set in a graph. The next step is obviously to try to solve the transformed instances, and to do so we will present the algorithm `mis` introduced by Fomin et. al. [22]. The algorithm will return the size of the biggest independent set in the input instance. If we know that our input instance of equations before the transformation has a solution, we can actually answer that question immediately, i.e., the size is m. On the other hand, we cannot say which vertices are contained in the set. By tracing `mis`' calculation we can reconstruct the independent set and ultimately provide the solution to our input equation system.

The reason for `mis`' significance in the context of equation solving is that one of its operations, namely *folding*, transforms the *equations* (which correspond to the complete graphs K_r^i) in a non-trivial way. That means that the modification to the input equations is new and to our knowledge not duplicated by any other algorithm.

In the following we call $N(v) = \{u \in V | uv \in E\}$ the *open neighborhood* of v. $N[v] = N(v) \cup \{v\}$ is the *closed neighborhood* of v. Further, we say that $\alpha(\mathcal{G})$ denotes the size of a maximum independent set in a graph $\mathcal{G}$.

We can then find that for a graph $\mathcal{G}$ the following two observations are true:

- For every connected component C of $\mathcal{G}$,

$$\alpha(\mathcal{G}) = \alpha(C) + \alpha(\mathcal{G} \setminus C).$$

- For any two vertices v, w s.t. $N[w] \subseteq N[v]$ it is true that

$$\alpha(\mathcal{G}) = \alpha(\mathcal{G} \setminus \{v\}).$$

The following two Lemmas along with their proofs can be found in [22]. They will play a central role in the algorithm `mis`.

Folding A vertex v is *foldable* if $N(v) = \{u_1, u_2, \ldots, u_{d(v)}\}$ contains no anti-triangle. *Folding* a vertex v is then applying the following steps to $\mathcal{G}$ in order to obtain the modified graph $\widetilde{\mathcal{G}}(v)$:

1. Add vertex u_{ij} for each anti-edge $u_i u_j$ in $N(v)$
2. Add edges between each u_{ij} and vertices in $N(u_i) \cup N(u_j)$
3. Add one edge between each pair of new vertices
4. Remove $N[v]$

Lemma 4. *Consider a graph $\mathcal{G}$, and let $\widetilde{\mathcal{G}}(v)$ be the graph obtained by folding a foldable vertex v. Then*

$$\alpha(\mathcal{G}) = 1 + \alpha(\widetilde{\mathcal{G}}(v)).$$

Mirroring A *mirror* of a vertex v is any $u \in \bigcup_{w \in N(v)} N(w)$ s.t. $N(v) \setminus N(u)$ forms a clique or $N(v) \setminus N(u) = \emptyset$. $M(v)$ denotes the set of all mirrors of v.

Lemma 5. *For any graph $\mathcal{G}$ and for any vertex v of $\mathcal{G}$,*

$$\alpha(\mathcal{G}) = max\{\alpha(\mathcal{G} \setminus (\{v\} \cup M(v))), 1 + \alpha(\mathcal{G} \setminus N[v])\}.$$

With the observations and lemmas above we are ready to state the algorithm `mis`. It takes as an argument a graph $\mathcal{G}$ and returns the size of the biggest independent set in $\mathcal{G}$. As stated before, the set of vertices forming the maximum independent set in $\mathcal{G}$ can be derived by backtracking `mis`' steps.

Algorithm 3 mis

1: **procedure** MIS($\mathcal{G}$)
2: **if**($|V(\mathcal{G})| \leq 1$) **return** $|V(\mathcal{G})|$
3: **if**($\exists$ connected component $C \subset \mathcal{G}$) **return** `mis`(C) + `mis`($\mathcal{G} \setminus C$)
4: **if**($\exists$ vertices v, w such that $N[w] \subseteq N[v]$) **return** `mis`($\mathcal{G} \setminus \{v\}$)
5: **if**($\exists$ vertex v with $deg(v) = 2$) **return** 1+`mis`($\widetilde{\mathcal{G}}(v)$)
6: select a vertex v of maximum degree which minimizes $|E(N(v))|$
7: **return** max$\{\alpha(\mathcal{G} \setminus (\{v\} \cup M(v))), 1 + \alpha(\mathcal{G} \setminus N[v])\}$
8: **end procedure**

Algorithm 3 represents just one method utilizing the previously presented lemmas on the properties of independent sets to solve the problem. Other algorithms, e.g., more specialized in solving instances resulting from the reduction, can be imagined and might improve the running time profoundly. For further details on the method used to find a maximum independent set, along with examples of the techniques, one can consult [22].

6.1.4 Experimental Results

We applied this algorithm to a small set of instances coming from random equation systems and examples from algebraic cryptanalysis. Unfortunately our simple implementation of the presented algorithm did not yield any improvement over the other techniques presented. Furthermore, the case of a foldable vertex occurred rather rarely.

Another problem during the short phase of experimentation, was the insufficiently optimized approach in handling the graphs generated from input equation systems.

6.1.5 Conclusion & Further Work

We have presented an alternative approach for solving a system of sparse equations over some finite field. A reduction to the well-known IS-problem was shown, along with a very short analysis of the resulting graph structure. Furthermore, we presented an algorithm by Fomin et. al. to solve the general IS-problem.

The presented algorithm by Fomin et. al. is directed toward the general IS-problem, i.e., not specialized on input from equation system. It is possible that additional heuristics in the algorithm can speed up the solving process.

It is furthermore unclear how much the graph structure, which is rather specific even for random equation systems, influences the hardness of the IS-problem. A further analysis of the graph structure of equation systems might lead to a greater insight and to more efficient algorithms.

Another problem during the investigation of the possibilities of this approach were the rather intricate implementational details. It might be possible that an efficient implementation of the presented approach can already compete against other existing techniques and we therefore think that a further investigation of the IS-problem in relation to the solving of equation systems might be worthwhile.

6.2 Algorithm Unification

When experimenting with versions of the Gluing/Agreeing algorithm and with the Syllogism method of solving equation systems [58], it becomes clear that running either one of them on one and the same instance can lead to different new information. That is, one algorithm can produce a different output from another. This fact is not very surprising in itself since they are two different algorithms. Nevertheless, the aim of both algorithms is the same and their guess and verify strategy is similar.

We denote by $\mathfrak{A}(r)$ the output of the Agreeing algorithm, by $\mathfrak{S}(r)$ the output of the Syllogism algorithm on an instance r. We can say that in general $\mathfrak{A}(r) \neq \mathfrak{S}(r)$ (there might exists r where the output is equal). We can illustrate this by the following example.

Example 7 (Agreeing vs. Syllogism). The equation system $r = \{S_0, S_1, S_2\}$ is pairwise agreeing

$$\begin{array}{c|ccc} S_0 & x_0 & x_1 & x_3 \\ \hline a_0 & 0 & 0 & 0 \\ a_1 & 0 & 0 & 1 \\ a_2 & 1 & 0 & 1 \\ a_3 & 1 & 1 & 1 \end{array}\ ,\quad \begin{array}{c|ccc} S_1 & x_1 & x_2 & x_4 \\ \hline b_0 & 0 & 0 & 1 \\ b_1 & 1 & 0 & 0 \\ b_2 & 1 & 0 & 1 \\ b_3 & 1 & 1 & 1 \end{array}\ ,\quad \begin{array}{c|ccc} S_2 & x_0 & x_2 & x_5 \\ \hline c_0 & 0 & 0 & 0 \\ c_1 & 0 & 1 & 1 \\ c_2 & 1 & 0 & 0 \\ c_3 & 1 & 1 & 0 \end{array} \tag{6.3}$$

and $\mathfrak{A}(r) = r$ would not yield any modification. On the other hand $\mathfrak{S}(r)$ would result in a modification to S_2, i.e., the vector c_1 would not be contained in $\mathfrak{S}(r)$.

The case for the Syllogism algorithm is similar. Let us assume that our instance r is now

$$\begin{array}{c|ccc} S_0 & x_0 & x_1 & x_2 \\ \hline a_0 & 0 & 0 & 0 \\ a_1 & 0 & 0 & 1 \\ a_2 & 0 & 1 & 0 \\ a_3 & 1 & 0 & 0 \\ a_4 & 1 & 1 & 1 \end{array}\ ,\quad \begin{array}{c|ccc} S_1 & x_0 & x_1 & x_2 \\ \hline b_0 & 0 & 0 & 0 \\ b_1 & 0 & 1 & 1 \\ b_2 & 1 & 0 & 1 \\ b_3 & 1 & 1 & 0 \\ b_4 & 1 & 1 & 1 \end{array}\ .$$

Then $\mathfrak{S}(r) = r$ despite the fact that we can *learn* from $\mathfrak{A}(r)$ that a_1, a_2, a_3, b_1, b_2 and b_3 cannot be part of any common solution to S_0 and S_2.

The straightforward way to make use of both algorithms simultaneously would be to run them consecutively on the same instance. While this is a a very easy solution which guarantees that the output contains no information which either algorithm would not be able to filter out, it would make the code of an implementation more complicated,

therefore more error-prone and probably less efficient. Less efficient since it is not guaranteed that Syllogism might yield additional information over Agreeing, and vice versa.

Another problem is that the analysis of the run-time of this compound algorithm might be more complicated, and it seems to be the least favorable solution. A *unified* algorithm, which would behave exactly as the composition $\mathfrak{A} \circ \mathfrak{S}$, could circumvent this problem. The following section describes how such an algorithm can be constructed by adding extra information to an Agreeing instance during a separate stage of preprocessing, and how this information is used during Agreeing to yield the same result as running first Agreeing, and then the Syllogism algorithm on the same instance.

This preprocessing step can then be applied to any Agreeing instance and changes complexity estimates of the algorithm only relative to the size of the input instance.

6.2.1 Syllog Preprocessing

First we have to recall that the Agreeing algorithm can work on *tuples* [44] or *pockets* [47]. Both approaches exploit the fact that it is not necessary to repeatedly calculate projections of vectors in common variables to other symbols. Instead, tuples or pockets are used to keep track of pairs of sets of vectors which are equal in their projection across different symbols. While the tuple-approach is limited to equivalences between sets[1], we make use of pockets which can express one-way implications and give us a higher degree of freedom to form the necessary constraints.

We will explain the preprocessing technique with the help of the following three symbols:

$$\begin{array}{c|cc} S_a & x_i & x_j \\ \hline a_0 & 0 & 0 \\ a_1 & 0 & 1 \\ a_2 & 1 & 0 \\ a_3 & 1 & 1 \end{array}, \quad \begin{array}{c|cc} S_b & x_j & x_k \\ \hline b_0 & 0 & 0 \\ b_1 & 0 & 1 \\ b_2 & 1 & 0 \\ b_3 & 1 & 1 \end{array}, \quad \begin{array}{c|cc} S_c & x_i & x_k \\ \hline c_0 & 0 & 0 \\ c_1 & 0 & 1 \\ c_2 & 1 & 0 \\ c_3 & 1 & 1 \end{array}. \tag{6.4}$$

These symbols are obviously pair-wise agreeing and do not contain any information. They contain each possible binary vector in two variables for all pairs (x_i,x_j), (x_j,x_k) and (x_i,x_k). In order that the Agreeing algorithm becomes active (as in *propagating knowledge*) one has to delete at least two vectors from either symbol. However, we know that if we delete the two correct vectors from two different symbols, the Syllogism algorithm can propagate some knowledge.

For example the deletion of a_0 (denoted as $\overline{a_0}$) yields the implication $\overline{x_i} \Rightarrow x_j$. The deletion of either b_2 or b_3 can give us new information which can be used to reduce S_c, i.e.:

- $\overline{b_2}$ would yield $x_j \Rightarrow x_k$ and therefore $\overline{x_i} \Rightarrow x_k$ which deletes c_0, or
- $\overline{b_3}$ which would yield $x_j \Rightarrow \overline{x_k}$ and therefore $\overline{x_i} \Rightarrow \overline{x_k}$ which deletes c_1.

[1]The tuple {A,B} represents the fact that: If the set of vectors A is excluded from a common solution then B must be excluded, and vice-versa.

We can express this fact with the following implications in the terms of marking vectors:

$$\begin{aligned}\{a_0, b_2\} &\Rightarrow \{c_0\} \\ \{a_0, b_3\} &\Rightarrow \{c_1\},\end{aligned}$$

where $A \Rightarrow B$ is to be understood as that the marking (*deletion*) of all vectors in A entails the marking of all vectors in B, but not the other way around.

Proceeding in the same way for all possible implications

$$\begin{aligned}&x_i^{(\alpha)} \Rightarrow x_j^{(\beta)} \Rightarrow x_k^{(\gamma)},\ x_i^{(\alpha)} \Rightarrow x_k^{(\beta)} \Rightarrow x_j^{(\gamma)} \\ &x_j^{(\alpha)} \Rightarrow x_i^{(\beta)} \Rightarrow x_k^{(\gamma)},\ x_j^{(\alpha)} \Rightarrow x_k^{(\beta)} \Rightarrow x_i^{(\gamma)} \\ &x_k^{(\alpha)} \Rightarrow x_i^{(\beta)} \Rightarrow x_j^{(\gamma)},\ x_k^{(\alpha)} \Rightarrow x_j^{(\beta)} \Rightarrow x_i^{(\gamma)}\end{aligned}$$

where $\alpha, \beta, \gamma \in \{0, 1\}$, $x_r^{(0)} = \overline{x_r}$ and $x_r^{(1)} = x_r$ can give us all pockets we need to express the behavior of the Syllogism algorithm.

All pockets which can be derived from all possible implications $x_i^{(\alpha)} \Rightarrow x_j^{(\beta)} \Rightarrow x_k^{(\gamma)}$ on (6.4) is shown in Figure 6.2.

$$\begin{aligned}p_0 &= (\{a_0, b_2\}, 8) & p_6 &= (\{a_3, b_0\}, 10) \\ p_1 &= (\{a_0, b_3\}, 9) & p_7 &= (\{a_3, b_1\}, 11) \\ p_2 &= (\{a_1, b_0\}, 8) & p_8 &= (\{c_0\}, \emptyset) \\ p_3 &= (\{a_1, b_1\}, 9) & p_9 &= (\{c_1\}, \emptyset) \\ p_4 &= (\{a_2, b_2\}, 10) & p_{10} &= (\{c_2\}, \emptyset) \\ p_5 &= (\{a_2, b_3\}, 11) & p_{11} &= (\{c_3\}, \emptyset)\end{aligned}$$

Figure 6.2: Syllogism pockets for $x_i^{(\alpha)} \Rightarrow x_j^{(\beta)} \Rightarrow x_k^{(\gamma)}$.

Again, just like in the Syllogism algorithm, the transitive closure of the implications is maintained, but this time with the help of pockets and the Agreeing algorithm. Instead of finding new implications we have expressed the technique of Syllogism as the deletion of vectors in an equation system.

The following algorithm `spre` implements such a preprocessing. It takes as input an equation system $\mathcal{S} = \{S_0, S_1, \ldots, S_{m-1}\}$ and returns all newly derived pockets according to the rules above from $\mathcal{S}$.

The crucial operations begin at line 5 in the algorithm, i.e., the generation of the pockets. Assume without loss of generality that $(\alpha, \beta, \gamma) = (0, 0, 0)$. Then set A will after the execution of line 5 contain all vectors from symbol S_a such that $v[x_i, x_j] = (0, 0)$. The deletion of all these vectors suggests that if $x_i = 0$ then $x_j = 1$ which is the implication $\overline{x_i} \Rightarrow x_j$. In line 6 all vectors of symbol S_b are collected in B for which it is true that $v[x_j, x_k] = (1, 0)$. A deletion of all these vectors would immediately yield that if $x_j = 1$ then $x_k = 1$, i.e., $x_j \Rightarrow x_k$. If both implications become simultaneously true due to the deletion (marking) of all vectors in A and B, we can derive by $\overline{x_i} \Rightarrow x_j \Rightarrow x_k$ the fact that $\overline{x_i} \Rightarrow x_k$. We therefore know that in this case from symbol S_c all vectors with $v[x_i, x_k] = (0, 0)$ need to be deleted, exactly the vectors which are collected in line 7.

Algorithm 4 Syllogism Preprocessing

1: **procedure** SPRE($\mathcal{S}$)
2: $N \leftarrow \emptyset$
3: **for** all triples (x_i, x_j, x_k), s.t., $x_i, x_j \in X_a,\ x_j, x_k \in X_b$ and $x_i, x_k \in X_c$ **do**
4: **for** all binary vectors (α, β, γ) **do**
5: $A \leftarrow \{v \in V_a \mid v[x_i, x_j] = (\alpha, \beta)\}$
6: $B \leftarrow \{v \in V_b \mid v[x_j, x_k] = (\overline{\beta}, \gamma)\}$
7: $C \leftarrow \{v \in V_c \mid v[x_i, x_k] = (\alpha, \gamma)\}$
8: $p_r \leftarrow (A \cup B, s)$
9: $p_s \leftarrow (C, \emptyset)$
10: Insert p_r and p_s into N
11: **end for**
12: **end for**
13: **return** N
14: **end procedure**

Example 8 (spre algorithm). Assume we apply spre on equation system (6.3) and we say that $x_i = x_0,\ x_j = x_1$ and $x_k = x_2$. At some point the algorithm would be in the state that $(\alpha, \beta, \gamma) = (0, 1, 1)$. At that point A would be empty since there is no vector $v[x_0, x_1] = (0, 1)$ in V_0. Likewise there does not exist any vector $v[x_1, x_2] = (0, 1)$ in V_1 and therefore $B = \emptyset$. This situation is equivalently expressed by the two implications

$$\overline{x_0} \Rightarrow \overline{x_1}$$
$$\overline{x_1} \Rightarrow \overline{x_2}$$

which yield $\overline{x_0} \Rightarrow \overline{x_2}$. The set C contains all vectors $v[x_0, x_2] = (0, 1)$. Since $A \cup B$ is empty we get an empty p_r pocket and all vectors in p_s, namely c_1 have to be deleted immediately.

6.2.2 Conclusion & Further Work

In this section we presented a method to *emulate* the Syllogism algorithm in the Agreeing algorithm without any modification of the Agreeing. With the preprocessing routine spre, we make full use of the transitive closure on implications throughout the equation system by running the Agreeing algorithm on the augmented set of pockets.

Open questions are for example if and how this augmentation of the pocket database influences the learning described in [47]. The overall complexity of Agreeing is only influenced by the number of the pockets, but a new estimate on the complexity in terms of the *number of equations*, *number of variables* and *sparsity* for the augmented algorithm would be interesting.

Bibliography

[1] 2012. http://www.ecrypt.eu.org/stream/. 2.3

[2] 2012. http://www.satcompetition.org/. 3.3

[3] ALBRECHT, M., CID, C., FAUGÈRE, J. C., AND PERRET, L. On the relation between the MXL family of algorithms and Gröbner basis algorithms. Cryptology ePrint Archive, Report 2011/164, 2011. http://eprint.iacr.org/. 3.2.2

[4] ARS, G., FAUGÈRE, J. C., IMAI, H., KAWAZOE, M., AND SUGITA, M. Comparison between XL and Gröbner Basis Algorithms. In *Advances in Cryptology — ASIACRYPT 2004, Springer-Verlag* (2004), pp. 338–353. 3.2.2

[5] BARD, G. V. *Algebraic Cryptanalysis*. Springer-Verlag, 2009. 2.4

[6] BARD, G. V., COURTOIS, N. T., AND JEFFERSON., C. Efficient Methods for Conversion and Solution of Sparse Systems of Low-Degree Multivariate Polynomials over GF(2) via SAT-Solvers. Cryptology ePrint Archive, Report 2007/024, 2007. http://eprint.iacr.org/. 3.3.1

[7] BOGDANOV, A., KHOVRATOVICH, D., AND RECHBERGER, C. Biclique Cryptanalysis of the Full AES. *Cryptology ePrint Archive, Report 2011/449* (2011). http://eprint.iacr.org/. 2.3

[8] CID, C., MURPHY, S., AND ROBSHAW, M. J. B. Small Scale Variants of the AES. In *Proceedings of the 12th international conference on Fast Software Encryption* (Berlin, Heidelberg, 2005), FSE'05, Springer-Verlag, pp. 145–162. 2.4

[9] COOK, D., AND KEROMYTIS, A. *CryptoGraphics: Exploiting Graphics Cards For Security (Advances in Information Security)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. 3.1

[10] COOK, S. A. An overview of computational complexity. *Communications of the ACM 26* (1983), 401–408. 6.1

[11] COURTOIS, N., ALEXANDER, K., PATARIN, J., AND SHAMIR, A. Efficient Algorithms for Solving Overdefined Systems of Multivariate Polynomial Equations. *Lecture Notes in Computer Science 1807* (2000), 392–407. 3.2, 3.2.1, 3.2.2

[12] COURTOIS, N., AND PATARIN, J. About the XL Algorithm over GF(2). *Lecture Notes in Computer Science 2612* (2003), 141–157. 3.2.1

[13] COURTOIS, N., AND PIEPRZYK, J. Cryptanalysis of Block Ciphers with Overdefined Systems of Equations. *Lecture Notes in Computer Science 2501* (2002), 267–287. 2.4

[14] COURTOIS, N. T., BARD, G. V., AND WAGNER, D. Algebraic and Slide Attacks on KeeLoq. In *Fast Software Encryption*, K. Nyberg, Ed. Springer-Verlag, Berlin, Heidelberg, 2008, pp. 97–115. 2.4

[15] DAVIS, M., LOGEMANN, G., AND LOVELAND, D. A Machine Program for Theorem-Proving. *Commun. ACM 5*, 7 (1962), 394–397. 3.3.1

[16] DAVIS, M., AND PUTNAM, H. A Computing Procedure for Quantification Theory. *J. ACM 7*, 3 (1960), 201–215, doi: 10.1145/321033.321034. 3.3.1

[17] DIFFIE, W. The First Ten Years of Public-Key Cryptography. In *Innovations in Internetworking*. Artech House, Inc., 1988, pp. 510–527. 2.3

[18] DINUR, I., AND SHAMIR, A. Cube Attacks on Tweakable Black Box Polynomials. Cryptology ePrint Archive, Report 2008/385, 2008. `http://eprint.iacr.org/`. 3.2

[19] EULER, L. Solutio problematis ad geometriam situs pertinentis. *Commentarii academiae scientiarum Petropolitanae 8* (1741), 128–140. 2.2

[20] FAUGÈRE, J. A new efficient algorithm for computing Gröbner bases (F4). *Journal of Pure and Applied Algebra 139*, 1-3 (1999), 61–88. 3.2.2

[21] FAUGÈRE, J. A new efficient algorithm for computing Gröbner bases without reduction to zero (F5). *Proceedings of the 2002 international symposium on Symbolic and algebraic computation* (2002), 75–83, doi: 10.1145/780506.780516. 3.2.2

[22] FOMIN, F. V., GRANDONI, F., AND KRATSCH, D. A Measure & Conquer Approach for the Analysis of Exact Algorithms. *J. ACM 56* (2009), 25:1–25:32. 6.1, 6.1.3, 6.1.3

[23] GOLDBERG, E., AND NOVIKOV, Y. BerkMin: A fast and robust SAT-solver. *Discrete Applied Mathematics 155*, 12 (2007), 1549–1561. 3.3.1

[24] GREENLAW, R., AND HOOVER, H. *Fundamentals of the Theory of Computation: Principles and Practice*. Morgan Kaufmann, 1998. 2.1.1

[25] GÜNEYSU, T., KASPER, T., NOVOTNÝ, M., PAAR, C., AND RUPP, A. Cryptanalysis with COPACOBANA. *IEEE Trans. Comput. 57*, 11 (Nov. 2008), 1498–1513, doi: 10.1109/TC.2008.80. 3.1

[26] JEROSLOW, R. G., AND WANG, J. Solving propositional satisfiability problems. *Annals of Mathematics and Artificial Intelligence 1* (1990), 167–187. 10.1007/BF01531077. 3.3.1

[27] KAHN, D. *The Codebreakers: The Comprehensive History of Secret Communication from Ancient Times to the Internet*, rev sub ed. Scribner, 1996. 2.3

[28] KARP, R. Reducibility Among Combinatorial Problems. In *Complexity of Computer Computations*, R. Miller and J. Thatcher, Eds. Plenum Press, 1972, pp. 85–103. 2.1.3, 2.1.4, 6.1

[29] KIPNIS, A., AND SHAMIR, A. Cryptanalysis of the HFE Public Key Cryptosystem by Relinearization. *Lecture Notes in Computer Science 1666* (1999), 788–788. 2.4, 3.2.1

[30] KNUTH, D. *The Art of Computer Programming Vol. 1*. Addison-Wesley, 1973. 2.1.2

[31] LANDAU, E. *Handbuch der Lehre von der Verteilung der Primzahlen*. B. G. Teubner, 1909. 2.1.2

[32] LANDAUER, R. Irreversibility and Heat Generation in the Computing Process. *IBM Journal of Research and Development 5* (1961), 183–191. 3.1

[33] LAURITZEN, N. *Concrete Abstract Algebra: From Numbers to Gröbner Bases*. Cambridge University Press, 2003. 3.2.2, 3.2.2

[34] MARQUES-SILVA, J. The Impact of Branching Heuristics in Propositional Satisfiability Algorithms. In *Proceedings of the 9th Portuguese Conference on Artificial Intelligence: Progress in Artificial Intelligence* (London, UK, 1999), EPIA '99, Springer-Verlag, pp. 62–74. 3.3.1

[35] MARQUES-SILVA, J. P., AND SAKALLAH, K. A. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Transactions on Computers 48* (1999), 506–521. 3.3.1

[36] MCDONALD, C., CHARNES, C., AND PIEPRZYK, J. Attacking Bivium with MiniSat. *ECRYPT Stream Cipher Project* (2007). 3.3

[37] MOSKEWICZ, M., MADIGAN, C., ZHAO, Y., ZHANG, L., AND MALIK, S. Chaff: Engineering an Efficient SAT solver. In *Proceedings of the 38th conference on Design automation* (2001), ACM New York, NY, USA, pp. 530–535. 3.3.1, 3.3.1

[38] NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. Data Encryption Standard. In *Federal Information Processing Standards Publication 46* (1977). 2.3

[39] NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. Triple DES. In *Federal Information Processing Standards Publication 46-3* (1999). 2.3

[40] NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. Advanced Encryption Standard. In *Federal Information Processing Standards Publication 197* (2001). 2.3

[41] PAPADIMITRIOU, CH. H. *Computational Complexity*. Addison-Wesley, 1994. 6.1

[42] RADDUM, H. MRHS Equation Systems. *Lecture Notes in Computer Science 4876* (2007), 232–245. 3.4, 3.4.3, 3.4.3

[43] RADDUM, H., AND SEMAEV, I. New Technique for Solving Sparse Equation Systems. *IACR Cryptology ePrint Archive* (2006). 3.4

[44] RADDUM, H., AND SEMAEV, I. Solving Multiple Right Hand Sides linear equations. *Designs, Codes and Cryptography 49*, 1 (2008), 147–160. 3.4.3, 3.4.3, 6.2.1

[45] RØNJOM, S., AND RADDUM, H. On the Number of Linearly Independent Equations Generated by XL. *Lecture Notes in Computer Science 5203* (2008), 239–251. 3.2.1

[46] ROTHE, J. *Complexity Theory and Cryptology. An Introduction to Cryptocomplexity*. EATCS Texts in Theoretical Computer Science. Springer, 2005. 2.1.3, 2.2.1, 6.1

[47] SCHILLING, T. E., AND RADDUM, H. Solving Equation Systems by Agreeing and Learning. *Lecture Notes in Computer Science* (2010), 151–165. 3.4.2, 6.2.1, 6.2.2

[48] SCHÖNING, U. *Theoretische Informatik — kurzgefaßt (3. Aufl.)*. Hochschultaschenbuch. Spektrum Akademischer Verlag, 1997. 2.1.1, 2.1.2

[49] SELMAN, B., KAUTZ, H. A., AND COHEN, B. Local Search Strategies for Satisfiability Testing. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science* (1996), pp. 521–532. 3.3

[50] SEMAEV, I. Sparse Algebraic Equations over Finite Fields. *SIAM Journal on Computing 39*, 2 (2009), 388–409, doi: 10.1137/070700371. 3.4, 3.4.2

[51] SHANNON, C. Communication Theory of Secrecy Systems. *Bell System Technical Journal 28* (1949), 656–715. 2.3

[52] SHANNON, C. E. A Mathematical Theory of Communication. *Bell system technical journal 27* (1948). 2.3

[53] SOOS, M., NOHL, K., AND CASTELLUCCIA, C. Extending SAT Solvers to Cryptographic Problems. In *SAT* (2009), pp. 244–257. 3.3

[54] U.S. DEPARTMENT OF COMMERCE, NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. Announcing Request for Candidate Algorithm nominations for a New Cryptograhpic Hash Algorithm (SHA–3) Family. *Federal Register 212* (2007). 2.3

[55] VAN HENTENRYCK, P., AND MICHEL, L. *Constraint-Based Local Search*. The MIT Press, 2005. 3.5

[56] WARSHALL, S. A theorem on Boolean matrices. *J. Assoc. Comput. Mach. 9* (1962). 3.5

[57] YANG, J., AND GOODMAN, J. Symmetric Key Cryptography on Modern Graphics Hardware. In *Proceedings of the Advances in Crypotology 13th international conference on Theory and application of cryptology and information security* (Berlin, Heidelberg, 2007), ASIACRYPT'07, Springer-Verlag, pp. 249–264. 3.1

[58] ZAKREVSKIJ, A., AND VASILKOVA, I. Reducing Large Systems of Boolean Equations. In *4th International Workshop on Boolean Problems* (2000), pp. 21–22. 3.5, 6.2