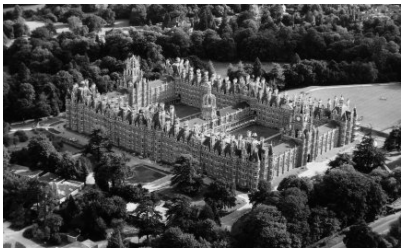
Proceedings of the ECRYPT Workshop on Tools for Cryptanalysis 2010

François-Xavier Standaert (Ed.)

22 – 23 June 2010,
Royal Holloway, University of London, Egham, UK



ECRYPT II

Organised by ECRYPT II - Network of Excellence in Cryptology II

Programme Chair

François-Xavier Standaert	Université catholique de Louvain, Belgium

Programme Committee

Martin Albrecht	Royal Holloway, University of London, UK
Frederik Armkrecht	Technical University of Darmstadt, Germany
Alex Biryukov	University of Luxembourg, Luxembourg
Joo Yeon Cho	Nokia A/S, Denmark
Christophe de Cannière	Katholiek Universiteit Leuven, Belgium
Serge Fehr	Centrum Wiskunde & Informatica, CWI, The Netherlands
Tim Güneysu	Ruhr Universität Bochum, Germany
Tanja Lange	Technische Universiteit Eindhoven, The Netherlands
Arjen Lenstra	Ecole Polytechnique Fédérale de Lausanne, Switzerland
Alexander May	Ruhr Universität Bochum, Germany
Florian Mendel	Graz University of Technology, Austria
Marine Minier	Institut National des Sciences Appliquées de Lyon, France
Ludovic Perret	LIP6-UPMC, Université de Paris 6 & INRIA, France
Gilles Piret	Oberthur Technologies, France
Damien Stehlé	CNRS, University of Sydney and Macquarie University, Australia
Michael Tunstall	University of Bristol, UK
Nicolas Veyrat-Charvillon	Université catholique de Louvain, Belgium

Preface

The focus of this workshop is on all aspects related to cryptanalysis research, mixing symmetric and asymmetric cryptography, as well as implementation issues. The workshop is a forum for presenting new software and hardware tools, including their underlying mathematical ideas and practical applications. Topics of interest include (but are not limited to):

- The automatic search of statistical trails in block ciphers.
- Lattice reduction and its application to the cryptanalysis of asymmetric encryption schemes.
- Fast HW and SW implementations for cryptanalysis (e.g. in FPGAs, graphic cards).
- Algebraic cryptanalysis (e.g. with SAT solvers, Gröbner bases).
- Sieve algorithms for integer factorization.
- Time/memory/data/key tradeoffs and birthday paradox-based attacks.
- Fourier and Hadamard-Walsh transforms and their applications in cryptanalysis.
- Tools for the fast and efficient collision search in hash functions.
- Physical (e.g. side-channel, fault) attacks, in particular their computational aspects.
- Cryptanalysis with alternative models of computation (e.g. quantum, DNA).

Invited Talks

A primer on lattice reduction

Marc Joye, Technicolor, France
Lattice basis reduction has found numerous applications in various research areas. This talk aims at giving a first introduction to the LLL algorithm. It explains how to use it to solve some simple algorithmic problems. It also describes how to break several early cryptosystems. The talk is mainly intended to practitioners wanting to use LLL as a toolbox. No mathematical background is required.

Factorization of a 768-bit RSA modulus

Paul Zimmermann, INRIA/LORIA, Nancy, France
On December 12, 2009, together with Kleinjung, Aoki, Franke, Lenstra, Thomé, Bos, Gaudry, Kruppa, Montgomery, Osvik, te Riele and Timofeev, we have completed the factorization of RSA-768 by the number field sieve. This factorization took the equivalent of about 1700 years on a 2.2Ghz AMD64 core. The talk will recall the main steps of the number field sieve, and will give the corresponding figures for RSA-768.

ASCAtoCNF - Simulating Algebraic Side-Channel Attacks

Mathieu Renauld*

UCL Crypto Group, Université catholique de Louvain, B-1348 Louvain-la-Neuve.

e-mails: mathieu.renauld@uclouvain.be

Abstract

Algebraic Side-Channel Attacks (ASCA) were recently presented ([1, 2]) as a new type of attack against block ciphers that combines side-channel information (information deduced from a physical leakage, like the power consumption of the device) and classical cryptanalysis (in this case algebraic cryptanalysis). These attacks show interesting properties. Indeed, they can exploit all available side-channel information (a standard DPA exploits only the first/last round), and thus require a smaller data complexity than other side-channel attacks. It turns out that ASCA can succeed with a data complexity of 1 (only one encryption measured), and even in an unknown plaintext/ciphertext context or against a masked implementation.

An ASCA can be divided in two phases. During the first online phase, the adversary performs measurements on the device during several encryptions. Then, during the second offline phase, the adversary translates the block cipher and the recovered side-channel information into a system of boolean equations, and tries to solve it. One of the possible techniques to solve this system is to translate it into a *satisfiability problem*, and to use a SAT solver.

ASCAtoCNF is a tool that provides the user with a quick way to simulate an ASCA with a data complexity of 1 to 9. The user specifies the target block cipher (PRESENT or the AES), the plaintext and secret key used. To simulate the side-channel recovery phase, the user chooses which operations of the block cipher are leaking information (for example: all the substitution operations from round 5 to 9). The side-channel recovery phase is assumed to be perfect (all recovered side-channel information is correct), but the user can make the attack harder by reducing the quantity of available side-channel information. The leakage model is the Hamming weight model on 8 bits: the adversary is assumed to recover the Hamming weight values of the data processed by the device during the specified leaking operations. The generated SAT problem can then be solved by a SAT solver like MiniSAT ([3]). With this tool, one can easily try various configurations of known leakages and study the impact of these configurations on the time complexity of the ASCA.

References

1. M. Renauld, F.-X. Standaert, *Algebraic Side-Channel Attacks*, in the proceedings of INSCRYPT 2009, to appear.
2. M. Renauld, F.-X. Standaert, N. Veyrat-Charvillon, *Algebraic Side-Channel Attacks on the AES: Why Time also Matters in DPA*, in the proceedings of CHES 2009, LNCS, vol. 5747, pp. 97-111, Lausanne, Switzerland, September, 2009.
3. MiniSAT, available online at http://minisat.se/.

* Work supported in part by the Walloon Region research project SCEPTIC.

Automated Algebraic Cryptanalysis

Paul Stankovski

Dept. of Electrical and Information Technology, Lund University,
P.O. Box 118, 221 00 Lund, Sweden

Abstract. We describe a simple tool for automatic algebraic cryptanalysis of a large array of stream- and block ciphers. Three tests have been implemented and the best results have led to continued work on a computational cluster. Our best results show nonrandomness in Trivium up to 1070 rounds (out of 1152), and in the full Grain-128 with 256 rounds.

Keywords: algebraic cryptanalysis, maximum degree monomial test, automated testing

The core of this work is the Maximum Degree Monomial (MDM) test [1, 2], which we use for algebraic cryptanalysis of a large array of stream and block ciphers. To facilitate time-efficient and automatic testing, we created a tool for running algebraic cryptanalysis tests. We assembled several specialized implementations that output initialization data, which is necessary for the algebraic tests. A generic interface then provides uniform access to all primitives. Algebraic tests can be implemented generically and run for each of the supported algorithms. This has been done for Trivium, Grain-128, Grain v1, Rabbit, Edon80, AES-128/256, DES, TEA, XTEA, SEED, PRESENT, SMS4, Camellia, RC5, RC6, HIGHT, CLEFIA, HC-128/256, MICKEY v2, Salsa20/12 and Sosemanuk.

We have implemented three particularly interesting tests. A greedy incarnation of the MDM test reveals inadequacies in bit mixing, and does so beautifully. This test can also point out unexpected key weight anomalies. A bit-flip test was devised to catch simple symmetry errors. Also, exhaustive search for small but optimal bit sets for the MDM test was also implemented.

The greedy approach to finding promising bit sets for the MDM test works exceptionally well for Trivium and Grain-128 (compare to [3, 4]). Using a computational cluster, we then pushed our computational limits to show weaknesses in Trivium reduced to 1070 (out of 1152) initialization rounds. The greedy strategy also works well for Grain-128, revealing nonrandomness through all 256 initialization rounds.

Our vision is that every algorithm designer should use our or other similar testing tools during algorithm development to catch algebraic weaknesses earlier than what has been possible before.

References

1. M.-J. O. Saarinen. Chosen-IV statistical attacks on eSTREAM stream ciphers. eSTREAM, ECRYPT Stream Cipher Project, Report 2006/013, 2006. http://www.ecrypt.eu.org/stream.
2. H. Englund, T. Johansson, and M. S. Turan. A framework for chosen IV statistical analysis of stream ciphers. In K. Srinathan, C. Pandu Rangan, and M. Yung, editors, *Progress in Cryptology - INDOCRYPT 2007*, volume 4859/2007 of *Lecture Notes in Computer Science*, pages 268–281. Springer-Verlag, 2007.
3. J.-P. Aumasson, I. Dinur, L. Henzen, W. Meier, and A. Shamir. Efficient FPGA Implementations of High-Dimensional Cube Testers on the Stream Cipher Grain-128. Available at http://eprint.iacr.org/2009/218/, Accessed June 17, 2009, 2009.
4. J.-P. Aumasson, I. Dinur, W. Meier, and A. Shamir. Cube Testers and Key Recovery Attacks on Reduced-Round MD6 and Trivium. In O. Dunkelman, editor, *Fast Software Encryption 2009*, volume 5665 of *Lecture Notes in Computer Science*, pages 1–22. Springer-Verlag, 2009.

Tools for Algebraic Cryptanalysis

Martin Albrecht*

Information Security Group, Royal Holloway, University of London
Egham, Surrey TW20 0EX, United Kingdom M.R.Albrecht@rhul.ac.uk

Algebraic cryptanalysis of cryptographic primitives such as block ciphers, stream ciphers and hash functions usually proceeds in two steps. (A) The algorithm is expressed as a system of multi-variate equations F over some field (usually $\mathbb{F}_2$). (B) The system F is solved using some technique such as Gröbner basis algorithms [2] , SAT solvers [3] or mixed integer programming solvers [1].

We provide scripts and tools for the mathematics software Sage [4] to construct polynomial systems of equations for various block ciphers and conversion routines from algebraic normal form (ANF) to conjunctive normal form (CNF) and mixed integer programmes. In particular we provide:

ctc.py polynomial systems for the Courtois Toy Cipher (CTC).
des.py polynomial systems for the Data Encryption Standard (DES).
katan.py polynomial systems for the KATAN/KTANTAN family of ciphers.
present.py polynomial systems for the PRESENT block cipher.
sea.py polynomial systems for the SEA block cipher.
anf2cnf.py a converter from ANF to CNF following [3].
anf2mip.py a converter from ANF to linear constraints following [1].

All scripts are available at `http://bitbucket.org/malb/algebraic_attacks`.

References

1. Julia Borghoff, Lars R. Knudsen, and Mathias Stolpe. Bivium as a Mixed-Integer Linear programming problem. In Matthew G. Parker, editor, *Cryptography and Coding – 12th IMA International Conference*, volume 5921 of *Lecture Notes in Computer Science*, pages 133–152, Berlin, Heidelberg, New York, 2009. Springer Verlag.
2. Johannes Buchmann, Andrei Pychkine, and Ralf-Philipp Weinmann. Block Ciphers Sensitive to Gröbner Basis Attacks. In *Topics in Cryptology – CT RSA'06*, volume 3860 of *Lecture Notes in Computer Science*, pages 313–331, Berlin, Heidelberg, New York, 2006. Springer Verlag. pre-print available at: `http://eprint.iacr.org/2005/200`.
3. Nicolas T. Courtois and Gregory V. Bard. Algebraic Cryptanalysis of the Data Encryption Standard. In Steven D. Galbraith, editor, *Cryptography and Coding – 11th IMA International Conference*, volume 4887 of *Lecture Notes in Computer Science*, pages 152–169, Berlin, Heidelberg, New York, 2007. Springer Verlag. pre-print available at `http://eprint.iacr.org/2006/402`.
4. William Stein et al. *SAGE Mathematics Software*. The Sage Development Team, 2008. Available at `http://www.sagemath.org`.

* This author was supported by the Royal Holloway Valerie Myerscough Scholarship.

Hybrid Approach : a Tool for Multivariate Cryptography

Luk Bettale*
joint work with Jean-Charles Faugère and Ludovic Perret

INRIA, Centre Paris-Rocquencourt, SALSA Project
UPMC, Univ. Paris 06, LIP6
CNRS, UMR 7606, LIP6
Boîte courrier 169
4, place Jussieu
75252 Paris Cedex 05, France
luk.bettale@lip6.fr

Abstract. In this paper, we present an algorithmic tool to cryptanalysis multivariate cryptosystems. The presented algorithm is a hybrid approach that mixes exhaustive search with classical Gröbner bases computation to solve multivariate polynomial systems over a finite field. Depending on the size of the field, our method is an improvement on existing techniques. For usual parameters of multivariate schemes, our method is effective. We give theoretical evidences on the efficiency of our approach as well as practical cryptanalysis of several multivariate signature schemes (TRMS, UOV) that were considered to be secure. For instance, on TRMS, our approach allow to forge a valid signature in 2^{67} operations instead of 2^{160} with exhaustive search or 2^{83} with only Gröbner bases. Our algorithm is general as its efficiency is demonstrated on random systems of equations. As the structure of the cryptosystem is not involved, our algorithm provides a generic tool to calibrate the parameters of any multivariate scheme. These results were already published in [5]. We also present an extended version of our hybrid approach, suitable for polynomials of higher degree. To easily access our tools, we provide a MAGMA package available at `http://www-salsa.lip6.fr/~bettale/hybrid.html` that provide all the necessary material to use our hybrid approach and to compute the complexities.

1 Introduction

Multivariate cryptography is a family of public key cryptosystems. The idea is to present the public key as a set of (generally quadratic) polynomials in a large number of variables. To introduce a trapdoor, a special algebraic system $F = (f_1(x_1, \ldots, x_n), \ldots, f_m(x_1, \ldots, x_n))$ is built such that it is easy to invert. The classical trapdoors are STS, UOV or HFE. To hide the structure of F, two invertible affine transformations $S, T \in \mathrm{Aff_n}(\mathbb{K})$ are chosen and the public key is the system

$$G = g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n) = T \circ F \circ S.$$

To encrypt, the system G is evaluated in the variables $m_1, \ldots, m_n$ corresponding to a message. The knowledge of the private key F, S, T allows the legitimate recipient to efficiently recover the message whereas an attacker has to solve the algebraic system G which should have no visible structure.

The problem of solving a multivariate system of equations, a.k.a. PoSSo, is known to be NP-hard, and also hard in average (exponential time). Note that PoSSo remains NP-hard even if the input polynomials are quadratics. In this case, PoSSo is also called $\mathcal{MQ}$. The security of a multivariate scheme relies directly on the hardness of solving a multivariate algebraic system of equations. In this context, it is important to have efficient tools to solve polynomial systems. When the system is considered as hard to solve as a random one (which is ideally required for a multivariate system), only general tools can be used to solve the system. We present in this paper an improved tool, namely the hybrid approach, that does not take advantage on the structure

* author partially supported by DGA/MRIS (french secretary of defense)

of the equations, but rather of the context to enhance the polynomial system solving. We use the fact that the field of coefficient is finite to perform a mix of exhaustive search and classical Gröbner bases techniques. For the parameters used in cryptography, our analysis shows that the hybrid approach brings a significant improvement over the classical methods. In [7], the authors did not succeed to attack UOV with Gröbner bases. Indeed, the parameters were unreachable using a standard zero-dimensional approach. With the hybrid approach we were able to break these parameters. Using this algorithm, we can put the security of general multivariate schemes to the proof. As our theoretic analysis allows to refine the security parameters, this make it a useful tool to design or cryptanalyze multivariate schemes.

Not only multivariate cryptography is concerned by the hybrid approach. In [15], the authors give a new method to solve the discrete logarithm problem on the group of points of an elliptic curve defined over an extension field. To do so, they have to solve a system of equations with of high degree in a quite big field. We present in this paper an extended hybrid approach which could be more suitable for this kind of problems.

Our contributions are available at `http://www-salsa.lip6.fr/~bettale/hybrid.html`

Organization of the paper

The paper is organized as follows. After this introduction, we present the general problem of solving a polynomial system as well as the classical method to address it, namely the zero-dim solving strategy using Gröbner bases. We also give the definitions of semi-regular sequences and degree of regularity, necessary to compute the complexity of our approach. In Section 3, we present the hybrid approach algorithm as well as its complexity. In Section 4, we give a generalization of the hybrid approach that uses splitted field equations. The scope of the extended hybrid approach will not be the same as the classical hybrid approach as it will be more efficient on polynomial systems of higher degree.

2 Polynomial System Solving

The general problem is to find (if any) $(z_1, \ldots, z_n) \in \mathbb{K}^n$ such that:

$$\begin{cases} f_1(z_1, \ldots, z_n) = 0 \\ \qquad \vdots \\ f_m(z_1, \ldots, z_n) = 0 \end{cases}$$

The best known method is to compute the Gröbner basis of the ideal generated by this system. We refer the reader to [1, 10] for a more thorough introduction to ideals and Gröbner bases. Informally, a Gröbner basis is a set of generators of an ideal which has "good" properties. In particular, if the system has a finite number of solution (zero-dimensional ideal), a Gröbner basis in Lex order has the following shape:

$$\{g_1(x_1), \ldots, g_2(x_1, x_2), \ldots, g_{k_1}(x_1, x_2), g_{k_1+1}(x_1, x_2, x_3), \ldots, g_{k_n}(x_1, \ldots, x_n)\}.$$

With this special structure, the system may be easily solved by successively eliminating variables, namely computing solutions of univariate polynomials and back-substituting the results.

The historical method for computing Gröbner bases was introduced by Buchberger in [8, 9]. Many improvements has been done leading to more efficient algorithms such as F_4 and F_5 due to Faugère [11, 12]. The algorithm F_4 for example is the default algorithm for computing Gröbner bases in the computer algebra softwares MAGMA and MAPLE. The F_5 algorithm[1] is even more efficient. We have mainly used this algorithm in our experiments. For our purpose, it is not necessary to describe the algorithm, but we give its complexity.

[1] available through FGb

Proposition 1. *The complexity of computing a Gröbner basis of a zero-dimensional system of m equations in n variables with F_5 is:*

$$\mathcal{O}\left(\left(m \cdot \binom{n+d_{reg}-1}{d_{reg}}\right)^{\omega}\right)$$

where d_{reg} is the degree of regularity of the system and $2 \leq \omega \leq 3$ is the linear algebra constant.

From a practical point of view, it is much faster to compute a Gröbner basis for a degree ordering such as the Degree Reverse Lexicographic (DRL) order than for a Lexicographic order (Lex). For zero-dimensional systems, it is usually less costly to first compute a DRL-Gröbner basis, and then to compute the Lex-Gröbner basis using a change ordering algorithm such as FGLM [13]. This strategy called zero-dim solving is performed blindly in modern computer algebra softwares. This is convenient for the user, but can be an issue for advanced users.

Proposition 2. *Given a Gröbner basis $G_1 \subset \mathbb{K}[x_1, \ldots, x_n]$ w.r.t. a monomial ordering $\prec_1$ of a zero-dimensional system the complexity of computing a Gröbner basis $G_2 \subset \mathbb{K}[x_1, \ldots, x_n]$ w.r.t. a monomial ordering $\prec_2$ with FGLM is:*

$$\mathcal{O}\left(n \cdot D^{\omega}\right)$$

where D is the degree of the ideal generated by G_1 (i.e. the number of solutions counted with multiplicity in the algebraic closure of $\mathbb{K}$).

We see easily that the cost of change ordering is negligible when the system has very few solutions.

For a finite field $\mathbb{K}$ with q elements, one can always add the field equations $x_1^q - x_1, \ldots, x_n^q - x_n$ to explicitly look for solutions over the ground field $\mathbb{K}$ and not in some extensions. By doing this, we will always obtain an over-defined system. This technique is widely used, and improves the computation of solutions if $q \ll n$. Otherwise, the addition of the field equations does not lead to a faster computation of a Gröbner basis. Even worse, this can slow down the computation due to the high degrees of the equations. In multivariate cryptography, some schemes use for example the field $\mathbb{F}_{2^8}$ whose elements can easily be represented with a byte. The hybrid method that we will present is especially suitable in such situation.

2.1 Semi-regular sequences

In order to study random systems, we need to formalize the definition of "random systems". To do so, the notion of regular sequences and semi-regular sequences (for over-defined systems) has been introduced in [2]. We give the definition here.

Definition 1. *Let $\{p_1, \ldots, p_m\} \subset \mathbb{K}[x_1, \ldots, x_n]$ be homogeneous polynomials of degrees $d_1, \ldots, d_m$ respectively. This sequence is semi-regular if:*

- $\langle p_1, \ldots, p_m \rangle \neq \mathbb{K}[x_1, \ldots, x_n]$
- *for all $1 \leq i \leq m$ and $g \in \mathbb{K}[x_1, \ldots, x_n]$:*

$$\deg(g \cdot p_i) < d_{reg} \text{ and } g \cdot p_i \in \langle p_1, \ldots, p_{i-1} \rangle \Rightarrow g \in \langle p_1, \ldots, p_{i-1} \rangle.$$

This notion can be extended to affine polynomials by considering their homogeneous components of highest degree. It has been proven in [2, 3] that for semi-regular sequences, the degree of regularity can be computed explicitly.

Property 1. The degree of regularity of a semi-regular sequence $p_1, \ldots, p_m$ of respective degrees $d_1, \ldots, d_m$ is given by the index of the first non-positive coefficient of:

$$\sum_{k \geq 0} c_k \cdot z^k = \frac{\prod_{i=1}^{m}(1 - z^{d_i})}{(1-z)^n}.$$

Let $D = \{d_1, \ldots, d_m\}$, we will denote the degree of regularity by $\mathrm{d_{reg}}(n, m, D)$.

This property allows us to have a very precise knowledge of the complexity of the computation of a Gröbner basis for semi-regular systems. For semi-regular systems it has been proven that the degree decreases as m goes larger. Thus, the more a system is over-defined, the faster its Gröbner basis can be computed.

For more convenience, we denote from now on the complexity of F_5 for semi-regular systems of equations of degree $d_1, \ldots, d_m$ as the function

$$\mathrm{C}_{\mathrm{F}_5}(n, m, D) = \left(m \cdot \binom{n + \mathrm{d}_{\mathrm{reg}}(n,D) - 1}{\mathrm{d}_{\mathrm{reg}}(n,D)} \right)^{\omega}$$

where D is the set $\{d_1, \ldots, d_m\}$.

3 Hybrid Approach

In many cases (especially in multivariate cryptography), the coefficient field is much bigger than the number of variables. In this case, as we have seen in Section 2, adding the field equations can dramatically slow down the computation of a Gröbner basis.

We present in this section our hybrid approach mixing exhaustive search and Gröbner bases techniques. First we will present the algorithm and discuss its complexity. Its efficiency depends on the choice of a proper trade-off. We take advantage of the behavior of semi-regular systems to find the best trade-off. After that, we give some examples coming from proposed cryptosystems as proof of concept.

3.1 Algorithm

In a finite field, one can always find all the solutions of an algebraic system by exhaustive search. The complete search should take q^n evaluations of the system if n is the number of variables and q the size of the field. The idea of the hybrid approach is to mix exhaustive search with Gröbner basis computations. Instead of computing one single Gröbner basis of the whole system, we compute the Gröbner bases of q^k subsystems obtained by fixing k variables. The intuition is that the gain obtained by solving systems with less variables may overcome the loss due to the exhaustive search on the fixed variables. Algorithm 1 describes the hybrid approach.

Algorithm 1 HybridSolving

Input: $\mathbb{K}$ is finite, $\{f_1, \ldots, f_m\} \subset \mathbb{K}[x_1, \ldots, x_n]$ is zero-dimensional, $k \in \mathbb{N}$.
Output: $\mathcal{S} = \{(z_1, \ldots, z_n) \in \mathbb{K}^n : f_i(z_1, \ldots, z_n) = 0, 1 \leq i \leq m\}$.
  $\mathcal{S} := \emptyset$
  for all $(v_1, \ldots, v_k) \in \mathbb{K}^k$ **do**
    Find the set of solutions $\mathcal{S}' \subset \mathbb{K}^{(n-k)}$ of
    $f_1(x_1, \ldots, x_{n-k}, v_1, \ldots, v_k) = 0, \ldots, f_m(x_1, \ldots, x_{n-k}, v_1, \ldots, v_k) = 0$
    using the zero-dim solving strategy.
    $\mathcal{S} := \mathcal{S} \cup \{(z'_1, \ldots, z'_{n-k}, v_1, \ldots, v_k) : (z'_1, \ldots, z'_{n-k}) \in \mathcal{S}'\}$.
  end for
  return $\mathcal{S}$.

As for the F_5 algorithm, the complexity of Algorithm 1 can be determined if the system is semi-regular. However, as the algorithm deals with sub-systems of m equations in $n - k$ variables, we will make the following assumption.

Hypothesis 1 *Let $\mathbb{K}$ be a finite field and $\{f_1, \ldots, f_m\} \subset \mathbb{K}[x_1, \ldots, x_n]$ a generic semi-regular system of equations of degree d. We will suppose that the systems*

$$\left\{ \{f_1(x_1, \ldots, x_{n-k}, v_1, \ldots, v_k), \ldots, f_m(x_1, \ldots, x_{n-k}, v_1, \ldots, v_k)\} : (v_1, \ldots, v_k) \in \mathbb{K}^k \right\}$$

are semi-regular, for all $0 \leq k \leq n$.

This hypothesis is consistent with the intuition that when some variables of a random system are fixed, the system is still random. This hypothesis has been verified with a rather large amount of random systems as well as systems coming from the applications of Section 3.2. In practice, the constructed systems may even be easier to solve than a semi-regular system. We have observed that its degree of regularity is always lower than a random system. Thus, our hypothesis can be used as it provides an upper bound on the complexity of our approach.

Proposition 3. *Let $\mathbb{K}$ be a finite field and $\{f_1, \ldots, f_m\} \subset \mathbb{K}[x_1, \ldots, x_n]$ be a semi-regular system of equations of degree $d_1, \ldots, d_m$ and $0 \leqslant k \leqslant n$. The complexity of solving the system with a hybrid approach, is bounded from above by:*

$$\mathcal{O}\left((\#\mathbb{K})^k \cdot \mathrm{C}_{\mathrm{F}_5}\left(n-k, m, D\right)\right)$$

where $D = \{d_1, \ldots, d_m\}$.

There exists a value k such that the complexity from Proposition 3 is minimal. If this value is non-trivial ($k \neq 0$ and $k \neq n$) then our method is an improvement on known techniques. In the next subsection, we give theoretical evidences that our approach is relevant on some ranges of parameters. In [5], we give for quadratic systems an asymptotic analysis of this complexity and an approximation of the best trade-off with respect to the parameters. We also show that our approach brings an improvement for quadratic systems if $\log_2(q)$ is smaller than $0.6226 \cdot \omega \cdot n$ where q is the size of the field and ω the linear algebra constant. For instance, to solve a system of 20 quadratic equations in 20 variables, the hybrid approach will bring an improvement if the field has a size below 2^{24}. Theses kind of parameters are generally found in multivariate cryptography. We show in the next section how the hybrid approach permits to break the parameters of some cryptosystems.

3.2 Applications

As proof of concept, we applied our hybrid approach to several multivariate cryptosystems. This permits to show a weakness in the choice of the parameters the TRMS [4] and UOV [14]. Our results have been given in [5]. In this paper, we don't describe the cryptosystems and only give a summary. As our approach does not depend on the structure of the systems, only the set of parameters matters to compute upper bounds. We give in Figure 1 the complexity of our approach depending on the parameter k. We see the best trade-off is to choose $k = 1$. The theoretical complexity drops from 2^{80} to 2^{67}. In practice, for TRMS we have even better results (reported in Table 1). In practice, choosing the best theoretical trade-off $k = 1$ would have taken too much memory, only $k = 2$ has been achieved.

Table 1. Experimental results on TRMS. The column m is the number of variables (and equations), $m - k$ is the number of variables left after fixing k variables. The columns T_{F_5}, Mem_{F_5}, and Nop_{F_5} are respectively the time, memory and number of operations needed to compute one Gröbner basis with the F_5 algorithm. The value T_{F_5} has to be multiplied by q^k to obtain Nop, the total number of operations of the hybrid approach.

m	$m-k$	q^k	T_{F_5}	Mem_{F_5}	Nop_{F_5}	Nop
20	18	2^{16}	51h	41.940 GB	2^{41}	2^{57}
20	17	2^{24}	2h45min	4.402 GB	2^{37}	2^{61}
20	16	2^{32}	626 s.	912 MB	2^{34}	2^{66}
20	15	2^{40}	46 s.	368 MB	2^{30}	2^{70}

Finally, our work permits to analyze the security of several multivariate schemes only by looking at their parameters. For example, in [6], the authors proposed implementations of some

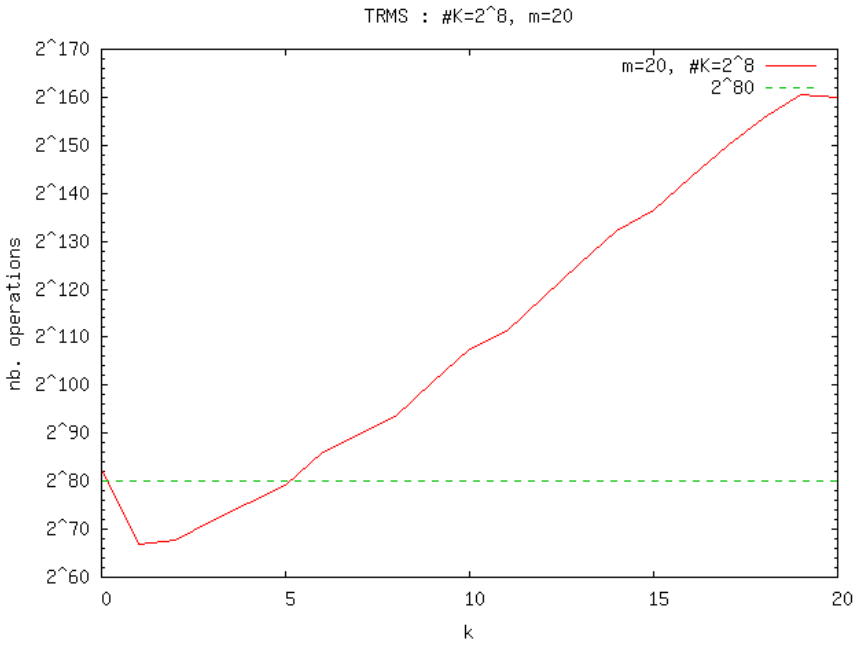


Fig. 1. TRMS: Complexity of hybrid approach depending on k

multivariate schemes. We were able to compute the minimum complexity of solving the public systems and we show that for a suitable value of k, the complexity of breaking all the proposed parameters are below 2^{80}. Our approach can be viewed as a tool to calibrate the parameters of multivariate cryptosystems.

4 Extended Hybrid Approach

In this section, we present a generalization of the hybrid approach.

4.1 Algorithm

We recall that the basic approach to find the solutions lying in the coefficient field $\mathbb{F}_q$ of a system of equations $f_1, \ldots, f_m$ is to solve the system with the field equations $x_1^q - x_1 = 0, \ldots, x_n^q - x_n = 0$. When q is too big, adding the field equations can be an issue.

The basis of the hybrid approach presented in Section 3 is to solve a set of easier systems of equations by fixing k variables $x_1, \ldots, x_k$ to some values $v_1, \ldots, v_k$. From another point of view, this means solving the original system on which we add k linear equations $x_1 - v_1 = 0, \ldots, x_k - v_k = 0$.

An idea in between could be to add "split" field equations. For a field $\mathbb{K}$ with q elements, it holds that $\prod_{e \in \mathbb{K}} x - e = x^q - x$. For a given parameter d, one could add only parts of the field equations $\prod_{i=1}^{i \leqslant d} x - e_i$ with $e_1, \ldots, e_d \in \mathbb{K}^n$. As in the hybrid approach, we could only add k equations to avoid a too big exhaustive search. The extended hybrid approach thus has two parameters. The number of split equations to be added $0 \leqslant k \leqslant n$ and their maximum degree $1 \leqslant d \leqslant q$. We remark that when $k = 0$, it becomes the classical zero-dim solving approach, when $k = n$ and $d = q$, it is the field equations approach and when $d = 1$, the approach is similar to the hybrid approach. Algorithm 2 describes the extended hybrid approach.

The complexity of Algorithm 2 can be computed in a similar way as Proposition 3.

Algorithm 2 ExtHybridSolving

Input: $\{f_1, \ldots, f_m\} \subset \mathbb{K}[x_1, \ldots, x_n]$ (zero-dim), $k, d \in \mathbb{N}$.
Output: $\mathcal{S} = \{(z_1, \ldots, z_n) \in \mathbb{K}^n : f_i(z_1, \ldots, z_n) = 0, 1 \leq i \leq m\}$.

```
S := ∅.
Let L = {h_1, ..., h_l} a factorization of the field equation x^q − x with deg(h_i) ⩽ d
for all (h_{i_1}, ..., h_{i_k}) ∈ L^k do
    Find the set of solutions S' ⊂ K^n of
    f_1 = 0, ..., f_m = 0, h_{i_1}(x_1) = 0, ..., h_{i_k}(x_k) = 0
    using the zero-dim solving strategy.
    S := S ∪ S'.
end for
return S.
```

Proposition 4. *Let $\mathbb{K}$ be a finite field and $\{f_1, \ldots, f_m\} \subset \mathbb{K}[x_1, \ldots, x_n]$ be a semi-regular system of equations of degree $d_1, \ldots, d_m$. The complexity of solving the system with the extended hybrid approach, is bounded from above by:*

$$\mathcal{O}\left(\sum_{i=0}^{k} \binom{k}{i} l^{k-i} \, \mathrm{C}_{\mathrm{F}_5}\left(n, \{d_1, \ldots, d_m, \underbrace{d, \ldots, d}_{k-i}, \underbrace{r, \ldots, r}_{i}\}\right)\right)$$

where $q = d \cdot l + r$, $0 < r \leqslant d$.

Proof. A field equation $x_i^q - x_i$ is split into l equations of degree d and 1 equation of degree r. For each subsystem, i (over k) split field equations of degree r are fixed, there are l^{k-i} possible systems. As there are $\binom{k}{i}$ possible positions for the degree r split field equations, we obtain the above result.

The above complexity can be bounded again by

$$\mathcal{O}\left(\left\lceil \frac{q}{d} \right\rceil^k \cdot \mathrm{C}_{\mathrm{F}_5}\left(n, \{d_1, \ldots, d_m, \underbrace{d, \ldots, d}_{k}\}\right)\right)$$

The two values match when $d \mid q$.

Here again, it is clear that the efficiency of this approach depends on the choice of parameters k and d. In the next section we will analyze the behavior of this approach and find out how to choose proper parameters that will bring the best trade-off between exhaustive search and Gröbner bases.

4.2 Analysis

To analyze the behavior of our approach, we have to be able to compute exactly the complexity of solving a given system. For semi-regular systems, we can know in advance its degree of regularity, and thus the complexity of the Gröbner basis computation. To perform our analysis, we use the approximation of the degree of regularity of an over-defined system (n variables, $n + k$ equations) given in [2]:

$$d_{reg} = \sum_{i=1}^{n+k} \frac{d_i - 1}{2} - \alpha_k \sqrt{\sum_{i=1}^{n+k} \frac{d_i^2 - 1}{6}} + \mathcal{O}(1)$$

when $n \to \infty$. Here, α_k is the largest root of the k-th Hermite's polynomial. To simplify the analysis, we will use the upper bound on the complexity of the extended hybrid approach.

$$C_{Hyb} = \left\lceil \frac{q}{d} \right\rceil^k \cdot \mathrm{C}_{\mathrm{F}_5}\left(n, \{d_1, \ldots, d_m, \underbrace{d, \ldots, d}_{k}\}\right).$$

Using the Stirling approximation $n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$, we can compute the logarithmic derivative of C_{Hyb} and thus find the minimum of the function, in the same way as in [5] for the hybrid approach.

The scope of the extended hybrid approach is not the same as the basic hybrid approach. While the hybrid approach was suitable for quadratic systems, the extended hybrid approach will show an improvement for system of equations of higher degree. For example, for a system of 5 equations of degree 8 in 5 variables in $\mathbb{F}_{31}$, the best theoretical trade-off is to add one split equation of degree 5 ($k = 1$, $d = 5$). From our experiments, for quadratic systems, the basic hybrid approach will always be better.

5 Conclusion

In this paper, we present a general tool to solve polynomial systems over finite fields, namely the hybrid approach. We have computed explicitly the complexity of this approach. The relevancy of our method is theoretically supported by the asymptotic analysis given in [5]. In practice, our approach is also efficient, in particular, it permits to break the parameters of several multivariate cryptosystems. We also present a generalization of this approach called the extended hybrid approach. From our analysis, this extension does not overpass the hybrid approach for quadratic systems. However, the extended hybrid approach is relevant on equations of higher degree. Finally, this paper gives a toolbox to analyze the parameters of multivariate cryptosystems. The complexity of our approaches can be used to better calibrate the parameters of multivariate cryptosystems. An implementation of the hybrid approach as well as functions to easily compute the complexity of our approach are available at `http://www-salsa.lip6.fr/~bettale/hybrid.html`.

References

1. William W. Adams and Philippe Loustaunau. *An Introduction to Gröbner Bases*, volume 3 of *Graduate Studies in Mahematics*. AMS, 1994.
2. Magali Bardet. *Étude des systèmes algébriques surdéterminés. Applications aux codes correcteurs et à la cryptographie*. PhD thesis, Université de Paris VI, 2004.
3. Magali Bardet, Jean-Charles Faugère, and Bruno Salvy. On the complexity of Gröbner basis computation of semi-regular overdetermined algebraic equations. In *Proc. International Conference on Polynomial System Solving (ICPSS)*, pages 71–75, 2004.
4. Luk Bettale, Jean-Charles Faugère, and Ludovic Perret. Cryptanalysis of the TRMS signature scheme of PKC'05. In *Progress in Cryptology – AFRICACRYPT 2008*, volume 5023 of *Lecture Notes in Computer Science*, pages 143–155. Springer, 2008.
5. Luk Bettale, Jean-Charles Faugère, and Ludovic Perret. Hybrid approach for solving multivariate systems over finite fields. *Journal of Mathematical Cryptology*, pages 177–197, 2009.
6. Andrey Bogdanov, Thomas Eisenbarth, Andy Rupp, and Christopher Wolf. Time-area optimized public-key engines: $\mathcal{MQ}$-cryptosystems as replacement for elliptic curves? In *CHES '08: Proceeding sof the 10th international workshop on Cryptographic Hardware and Embedded Systems*, pages 45–61. Springer-Verlag, 2008.
7. An Braeken, Christopher Wolf, and Bart Preneel. A study of the security of Unbalanced Oil and Vinegar signature schemes. In *Topics in Cryptology - CT-RSA 2005*, volume 3376 of *LNCS*, pages 29–43. Springer, February 2005.
8. Bruno Buchberger. *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal*. PhD thesis, University of Innsbruck, 1965.
9. Bruno Buchberger, Georges E. Collins, Rudiger G. K. Loos, and Rudolph Albrecht. Computer algebra symbolic and algebraic computation. *SIGSAM Bull.*, 16(4):5–5, 1982.
10. David A. Cox, John B. Little, and Don O'Shea. *Ideals, Varieties and Algorithms*. Springer, 2005.
11. Jean-Charles Faugère. A new efficient algorithm for computing Gröbner bases (F4). *Journal of Pure and Applied Algebra*, 139:61–88, June 1999.
12. Jean-Charles Faugère. A new efficient algorithm for computing Gröbner bases without reduction to zero (F5). In *Proceedings of the 2002 International Symposium on Symbolic and Algebraic Computation ISSAC*, pages 75–83. ACM Press, 2002.

13. Jean-Charles Faugère, Patrizia M. Gianni, Daniel Lazard, and Teo Mora. Efficient computation of zero-dimensional Gröbner bases by change of ordering. *Journal of Symbolic Computation*, 16(4):329–344, 1993.
14. Jean-Charles Faugère and Ludovic Perret. On the security of UOV. In *SCC 08*, 2008.
15. Antoine Joux and Vanessa Vitse. Elliptic curve discrete logarithm problem over small degree extension fields. `http://eprint.iacr.org/2010/157.pdf`, 2010.

Sieving for Shortest Vectors in Ideal Lattices

Michael Schneider

Technische Universität Darmstadt, Germany
mischnei@cdc.informatik.tu-darmstadt.de

Lattice based cryptography is gaining more and more importance in the cryptographic community. It is a common approach to use a special class of lattices, so-called ideal lattices, as the basis of these systems. This speeds up computations and saves storage space for cryptographic keys. The most important underlying hard problem is the computational variant of the shortest vector problem. So far there is no algorithm known that solves the shortest vector problem in ideal lattices faster than in regular lattices. Therefore, cryptosystems using ideal lattices are considered to be as secure as their regular counterparts.

In this workshop we will present Ideal Sieve, a variant of the Gauss Sieve algorithm [MV10], that is a randomized sieving algorithm that solves the shortest vector problem in lattices. Our variant makes use of the special structure of ideal lattices. We show that it is indeed possible to find a shortest vector in ideal lattices faster than in regular lattices without special structure. The speedup of our algorithm is linear in the lattice dimension, i.e., the runtime grows linearly in the dimension as well as the maximum list size decreases.

Figures 1 and 2 show experimental results of a software implementation of Ideal Sieve compared to Gauss Sieve in lattice dimension $m \leq 80$. Figure 2 shows that the number of iterations performed by the algorithm indeed decreases with a factor linearly in the lattice dimension, when exploiting the structure of ideal lattices. The number of list vectors decreases with a factor $\approx 0.4 \cdot m$. This reduces the amount of storage needed, which is one of the main bottlenecks of sieve algorithms for shortest vectors. Due to some computational overhead, the actual runtime seems to profit best in higher dimensions.

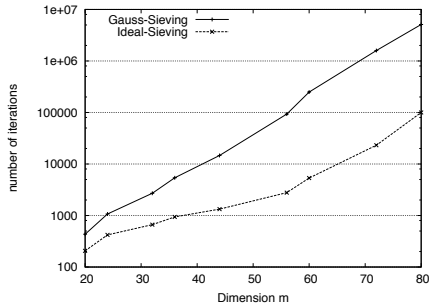


Fig. 1. Number of iterations required for sieving in cyclic lattices of dimension m.

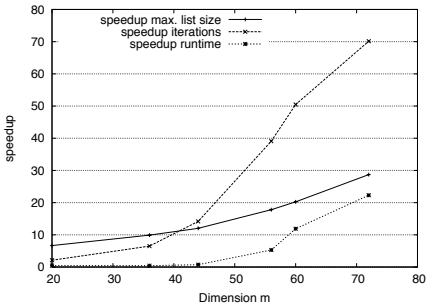


Fig. 2. Speedup factors for sieving in ideal lattices. The graph shows the data of Gauss Sieve divided by the data of our Ideal Sieve.

References

[MV10] Daniele Micciancio and Panagiotis Voulgaris. Faster exponential time algorithms for the shortest vector problem. In *SODA 2010*, pages 1468–1480, 2010.

Time-Memory and Time-Memory-Data Trade-Offs for Noisy Ciphertext

Marc Fossorier[1], Miodrag J. Mihaljević[2,4] and Hideki Imai[3,4]

[1]ENSEA, UCP, CNRS UMR-8051,
6 avenue du Ponceau, 95014 Cergy Pontoise, France
[2]Mathematical Institute, Serbian Academy of Sciences and Arts, Belgrade, Serbia
E-mail: miodragm@turing.mi.sanu.ac.rs
[3]Chuo University, Faculty of Science and Engineering, Tokyo, Japan
[4]Research Center for Information Security (RCIS), National Institute of Advanced Industrial Science and Technology (AIST), Tokyo, Japan

Abstract. In this paper, the time-memory trade-off proposed by Hellman and its generalization to the time-memory-data trade-off proposed by Biryukov and Shamir are generalized to a noisy observation of the ciphertext. This generalization exploits the inherent error correcting capability of the considered encryption scheme. Two basic approaches taking into account the effect of the sample corruption are considered and the corresponding mixture of these two approaches is proposed and analyzed. This unified approach provides a framework for comparison of the effectiveness of the two basic approaches and allows to determine the most efficient strategy when the sample is corrupted by additive noise.

Keywords: cryptography, cryptanalysis, time-memory trade-off, time-memory-data trade-off, stream ciphers.

1 Introduction

In [9], the chosen plaintext attack is considered, that is the problem of recovering a key of length $K = \log_2 N$ bits used to encrypt a particular plaintext into a sequence of L bits, $L \geq K$ when observing this encrypted (noiseless) sequence. It is shown that after proper preprocessing of complexity N, the processing time complexity T and memory M can be traded according to $M^2T = N^2$, leading to the optimum time-memory trade-off (TM-TO) $M = T = N^{2/3}$. The method of [9] is essentially an efficient implementation to invert a non linear function. In [4], this trade-off was generalized to the case where D realizations of the function are available (i.e D observations of the same plaintext encrypted by D different keys for the chosen plaintext attack), leading to the time-memory-data trade-off (TMD-TO) $M^2D^2T = N^2$. No significantly better trade-offs can be achieved for this problem [2]. Other related trade-offs such as the time data trade-offs have also been investigated [7].

The approaches of [9, 4] implicitly assume the available sample is error free. As a result, if the sample data are corrupted by noise, the TM-TO does not recover the correct argument of the non linear function it inverts. In this paper, we generalize the TM and TMD trade-offs for a noisy observation of the ciphertext. In this case, the inherent error correcting capability of the cryptosystem can be exploited. The availability of a noisy sample appears in a number of realistic scenarios, although usually the assumption is that the error free data are available for performing the cryptanalysis. A particular example for this situation is the cryptanalysis of GSM mobile telephony (see [1] for example).

2 A Brief Review of the TM and TMD Trade-Offs

In this section, an overview of the basic TM and TMD trade-off concepts is presented according to [9] and [4], respectively (related issues can be found in [6], [13] and [2]).

2.1 TM Trade-Off

Let $f(\cdot)$ denote a one-way function, and k a secret key. Computing $f(k)$ is simple, but computing k from $f(k)$ is equivalent to cryptanalysis. The TM-TO concept is based on the following two phases: a precomputation phase which should be performed only once, and a processing phase which should be performed for the reconstruction of each particular secret key.

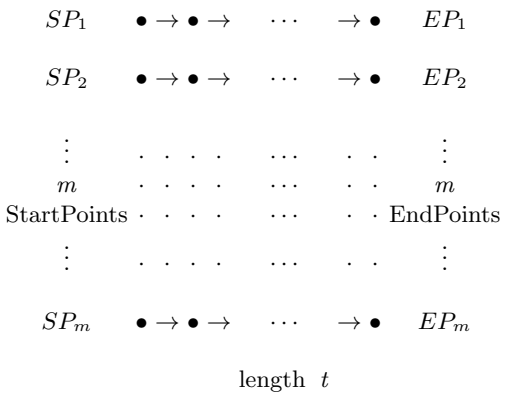


Fig. 1. The underlying matrix for time-memory trade-off.

As part of the precomputation, the cryptanalyst chooses m starting points, $SP_1, SP_2, ..., SP_m$, each an independent random variable drawn uniformly from the key space $\{1, 2, ..., N\}$. For $1 \leq i \leq m$, he lets

$$X_{i0} = SP_i \tag{1}$$

and computes

$$X_{ij} = f(X_{i,j-1}) \ , \ \ 1 \leq j \leq t \ , \tag{2}$$

following the scheme depicted in Fig. 1. The parameters m and t are chosen by the cryptanalyst to trade-off time against memory.

The last element or endpoint in the ith chain (or row) is denoted by EP_i. Clearly,

$$EP_i = f^t(SP_i) \ , \tag{3}$$

where $f^t(\cdot)$ denotes the corresponding self-composition of $f(\cdot)$ t times.

The complexity to construct the table is mt. However, to reduce memory requirements, the cryptanalyst discards all intermediate points as they are produced and sorts $\{SP_i, EP_i\}_{i=1}^m$ on the endpoints. The sorted table is stored as the result of this precomputation.

Next we explain how this table can be used for the cryptanalysis of a known encryption algorithm $E_k(\cdot)$. Suppose that the cryptanalyst has obtained the pair (Y_0, P_0) where

$$Y_0 = E_k(P_0). \tag{4}$$

and assume (4) was performed in the preprocessing stage of the table. Under this assumption, we consider the problem of recovering the secret key k when the encryption algorithm $E_k(\cdot)$ and its corresponding decryption algorithm $D_k(\cdot)$, the ciphertext Y_0, and the corresponding plaintext P_0 are known to the cryptanalyst.

Suppose that the following is valid

$$Y_1 = f(k). \tag{5}$$

The cryptanalyst can check if Y_1 is an endpoint in one "operation" because the pairs $\{(SP_i, EP_i)\}$ are sorted on the endpoints. Accordingly:

- If Y_1 is not an endpoint, the key k is not in the second to last column in Fig. 1 (if it was there, Y_1, which is its image under f, would be an endpoint).
- If $Y_1 = EP_i$, then either $k = X_{i,t-1}$ (i.e. k is in the second to last column of Fig. 1) or EP_i has more than one inverse. We refer to this latter event as a false alarm (FA). If $Y_1 = EP_i$, the cryptanalyst computes $X_{i,t-1}$ and checks whether it is the key, for example by verifying whether it deciphers Y_0 into P_0. Because all the intermediate columns in Fig. 1 have been discarded to save memory, the cryptanalyst must start at SP_i and recompute $X_{i,1}, X_{i,2}, ...,$ until he reaches $X_{i,t-1}$.
- If Y_1 is not an endpoint or a FA occurred, the cryptanalyst computes

$$Y_2 = f(Y_1) \tag{6}$$

and checks whether it is an endpoint. If it is not, the key is not in the $(t-2)$-th column of Fig. 1, while if $Y_2 = EP_i$ the cryptanalyst checks whether $X_{i,t-2}$ is the key.

– In a similar manner, the cryptanalyst computes

$$Y_3 = f(Y_2), \cdots, Y_t = f(Y_{t-1})$$

to check whether the key is in the $(t-3)$-th, $\cdots$, the 1-st column of the table in Fig. 1.

In [9], it is shown that the number of FAs per table is about mt^2/N. As a result, $mt^2 \approx \alpha N$ for $\mathrm{O}(\alpha)$ FAs per table. Since the probability of success for this approach is about mt/N per table, it follows that $N/(mt) \approx t/\alpha$ tables[1] are required for a probability of success close to 1, with $\alpha \leq t$. If P denotes the pre-processing complexity, T denotes the total processing time complexity and M denotes the total memory, we have for t/α tables: $T = (t/\alpha)\alpha t = t^2$ and $M = mt/\alpha$, so that the corresponding TM-TO for this attack satisfies $P = N$ and $TM^2 = N^2$. A typical point for this trade-off is $T = N^{2/3}$ processing (attack) time and $M = N^{2/3}$ memory space[2].

We finally notice that in this approach, it is implicitly assumed that in (4), the length L of the plaintext P_0 is the same as that of the key k, denoted K. The approach remains valid for $L \geq K$ after replacing $Y_i = f(Y_{i-1})$ by $Y_i = f(Y_{i-1}^*)$ in the recursive process depicted in Fig. 1, where Y_{i-1}^* represents any truncation of Y_{i-1} to a length-K vector.

2.2 TMD Trade-Off

Assume that for the same plaintext P_0, we now have D encrypted values

$$Y_{0_i} = E_{k_i}(P_0) \tag{7}$$

for $1 \leq i \leq D$. This is for example the case if the secret key k initializes the internal state of a stream cipher, this state having the same dimension as k, and we have D different ciphertexts obtained from the stream cipher (note that the internal state content corresponding to the beginning of each of the D blocks can be viewed as a new key drawn randomly from the previous one). In that case, the attack is successful if any one of the D given outputs can be found in the tables corresponding to Fig. 1. Accordingly the corresponding number of FAs per table remains about mt^2/N per processed data, so that for less than one FA per table on average, the probability of success for one table becomes at least Dmt/N. It follows that t/D tables are required, $t \leq D$, and since the D data are now processed at each step, we obtain $T = t/D \cdot t \cdot D = t^2$ and $M = t/D \cdot m = mt/D$.

[1] Each table is constructed employing a function which is a slight modification of $f()$ as discussed in [9].

[2] We observe that the value of α does not influence the TM-TO. Hence in the sequel of this letter, we assume $\alpha = 1$.

This suggests the TMD-TO attack proposed in [4] for stream ciphers, which satisfies $P = N/D$ and $TM^2D^2 = N^2$ for any $D^2 \leq T \leq N$. A typical point for this trade-off relation is $P = N^{2/3}$ pre-processing time, $T = N^{2/3}$ attack time, $M = N^{1/3}$ memory space, and $D = N^{1/3}$ available data.

3 The Noisy Case

In Section 2, it is assumed that the observed ciphertext is error free. However in several scenarios, this assumption may not hold and only a noisy version $\tilde{Y}_0$ of Y_0 is available to the cryptanalyst. In this section, we generalize the TM and TMD trade-offs to this noisy scenario for $K \leq L$. In this case, the encryption can be viewed as a non linear code C with inherent correcting capability e.[3] As a result, we assume

$$\tilde{Y}_0 = Y_0 + e_0 \tag{8}$$

with $w_H(e_0) \leq e$.

3.1 TM Trade-Off

Two basic approaches In the table introduced in Section 2, the $N = 2^K$ possible encryptions of P_0 can be viewed as the codewords of a non linear code C of length L. Define $\tilde{Y}_0^*$ as the truncation of $\tilde{Y}_0$ to K positions randomly selected.[4]

Two approaches can be selected to implement the TM-TO for a noisy sample:

1 The same table as in Section 2 is constructed for the noiseless case only, and all possible error patterns are added to the truncation, the first K of L bits, of the received sample for processing.
2 Tables that cover all possible noisy versions of the keys are constructed and the truncation of the noisy received sample only is used for processing.

In approach-1, the vector $\tilde{Y}_0^*$ is expanded into the $\sum_{l=0}^{e} \binom{K}{l} \approx K^e$ vectors $\tilde{Y}_l^*$ obtained by adding to $\tilde{Y}_0^*$ all possible K-tuples e_l with $w_H(e_l) \leq e$, so that

$$\tilde{Y}_l^* = \tilde{Y}_0^* + e_l. \tag{9}$$

[3] A random code of length n and rate R is likely to be able to correct all errors of weight $e \leq \lfloor (h^{-1}(1-R) - 1)/2 \rfloor$ for n large enough, where $h^{-1}()$ is the inverse of the binary entropy function $h(p) = -p \log p - (1-p)\log(1-p)$ [8].

[4] Since for a non linear code over GF(2) with 2^K codewords, an information set is defined as a minimal subset of cardinality $J \geq K$ of the positions such that no member of $\mathrm{GF}(2)^J$ is repeated in these positions [8], the selected K positions not necessarily form an information set. However this issue is implicitly considered by FAs.

The iterative process described in Section 2 is then applied to all $\sum_{l=0}^{e} \binom{K}{l}$ vectors $\tilde{Y}_l^*$ in parallel. Assume that for some row-i in the table of Fig. 1, we have

$$f^j(\tilde{Y}_l^*) = EP_i. \tag{10}$$

The corresponding SP_i is encrypted $t-j$ times and the candidate is selected if

$$d_H(f^{t-j}(SP_i), \tilde{Y}_0) \leq e. \tag{11}$$

Since the code C is assumed to correct all errors of weight at most e, only one $f^{t-j-1}(SP_i)$ should satisfy (11).

In approach-2, K^e tables are constructed together by associating to each table a particular error pattern e_l with $w_H(e_l) \leq e$. The table associated with e_l is constructed based on the recursion

$$Y_i = f(Y_{i-1}^* + e_l). \tag{12}$$

Then only $\tilde{Y}_0^*$ needs to be processed.

It should be noted that in approach-1, only the key space is covered by preprocessing while in approach-2, preprocessing covers the cross product of the key space and noise space of interest. Approach-2 is equivalent to the attack of [5] which addresses key recovery of stream ciphers using a publicly known initial value. For this problem, approach-1 is unfeasible due to the non linear use of the initial value, as opposed to the additivity of the noise. A similar approach was employed in [11] for cryptanalysis in certain broadcast encryption scenarios. Approach-1 can be viewed as an exhaustive coverage of all error patterns and consequently, increases the complexity of the attack for the noiseless case by a factor proportional to the number of errors to cover, as suggested in [3, p.58]. However it should be noted that this approach increases the time complexity of the noiseless attack only, so that a better TM-TO should be achievable if this complexity increase is considered in the initial TM-TO of the noiseless attack.

A mixed approach In this section, a unified approach which allows the joint consideration of approach-1 and approach-2 is developed. This unified approach provides a framework for comparison of the effectiveness of these two approaches and allows to determine the most efficient strategy for the TM-TO and TMD-TO when the sample is corrupted by additive noise. We define

$$K^e = K^{e_1} K^{e_2} \tag{13}$$

and construct K^{e_1} tables as in approach-2. Then $\tilde{Y}_0^*$ is expanded into K^{e_2} vectors $\tilde{Y}_l^*$.[5] As a result, approach-1 is obtained for $e_1 = 0$ and $e_2 = e$, while approach-2 corresponds to $e_1 = 1$ and $e_2 = 0$. The TM-TO for this mixed approach is specified by the following theorem.

[5] We implicitply assume there exists a partition which allows to cover all K^e based on (13). Since the final result does not depend on this partition, we do not explicitly define it.

Theorem 31 *The TM-TO for cryptanalysis of a system with key length K, plaintext length L, $L \geq K$, and able to correct up to e errors in the ciphertext with a preprocessing covering K^{e_1} error patterns, $e_1 \leq e$, is given by*

$$T_{tot} M_{tot}^2 \approx K^{2e_1+e_2} 2^{2K} \tag{14}$$

where T_{tot} and M_{tot} represent the total time complexity and total memory, respectively. It can be achieved with

$$M_{tot} = T_{tot} \approx K^{(2e_1+e_2)/3} 2^{2K/3} \tag{15}$$

Proof: For each of the K^{e_1} error patterns to be covered by preprocessing, an $m \times t$ table is constructed as in [5]. Assuming the error pattern is one of the K^{e_1} error patterns covered by preprocessing, the expected number of FAs in the corresponding table is given by approximatively $2^{-K} mt^2$, while this table covers the key with probability $2^{-K} mt$. Hence for O(1) FA per table, about $K^{e_1} t$ tables are needed. The corresponding time complexity is $T \approx t^2$ and the total memory required is $M \approx mtK^{e_1}$. It follows that

$$M^2 T \approx K^{2e_1} 2^{2K}. \tag{16}$$

To cover all possible K^e error patterns, K^{e_2} values $\tilde{Y}_l^*$ are processed so that the total processing time and total memory are given by

$$\begin{aligned} T_{tot} &= K^{e_2} T \\ M_{tot} &= M, \end{aligned} \tag{17}$$

respectively. Combining (16) and (17) provides (14), from which Theorem 31 follows.

The following corollary indicates that approach-1, obtained from $e_1 = 0$ and $e_2 = e$ in Theorem 31, is the most efficient. Interesting, it also minimizes the associated preprocessing $P = K^{e_1} 2^K$.

Corollary 31 *The TM-TO for cryptanalysis of a system with key length K, plaintext length L, $L \geq K$, and able to correct up to e errors in the ciphertext is given by*

$$T_{tot} M_{tot}^2 \approx K^e 2^{2K} \tag{18}$$

where T_{tot} and M_{tot} represent the total time complexity and total memory, respectively. It can be achieved with

$$M_{tot} = T_{tot} \approx K^{e/3} 2^{2K/3} \tag{19}$$

It can be noted that the approach is valid for any K positions defining $\tilde{Y}_0^*$ from $\tilde{Y}_0$. Consequently, assuming α non intersecting sets of such K positions have been identified, $1 \leq \alpha \leq \lfloor L/K \rfloor$, there exists at least one such set with at most $\lceil e/\alpha \rceil$ errors in it. Applying the previous approach to all α sets with $w_H(e_l) \leq \lceil e/\alpha \rceil$ is then sufficient to succeed, providing the following corollary.

Corollary 32 *The TM-TO for cryptanalysis of a system with key length K, and plaintext length L, $L \geq K$, and able to correct up to e errors in the ciphertext is given by*

$$T_{tot} = M_{tot} \approx K^{\lceil e/\alpha \rceil/3} 2^{2K/3} \tag{20}$$

if α non intersecting sets of K positions are considered.

While an additive noisy ciphertext has been assumed in this study, a similar approach holds for a ciphertext with erasures. In that case, the location of the erasures is known and for ϵ erasures, all 2^ϵ hypotheses can be covered based on approach-1. We obtain the following corollary.

Corollary 33 *The TM-TO for cryptanalysis of a system with key length K, plaintext length L, $L \geq K$, and able to correct up to ϵ erasures in the ciphertext is given by*

$$T_{tot} M_{tot}^2 \approx 2^\epsilon 2^{2K} \tag{21}$$

where T_{tot} and M_{tot} represent the total time complexity and total memory, respectively. It can be achieved with

$$M_{tot} = T_{tot} \approx 2^{\epsilon/3} 2^{2K/3} \tag{22}$$

3.2 TMD Trade-Off

As in Section 2.2, Theorem 31 can be generalized as follows based on a preprocessing of $P = 2^K/D$.

Theorem 32 *The TMD-TO for cryptanalysis of a system with key length K, plaintext length L, $L \geq K$, able to correct up to e errors and with D encryptions of the same plaintext available is given by*

$$T_{tot} M_{tot}^2 D^2 \approx K^e 2^{2K} \tag{23}$$

where T_{tot} and M_{tot} represent the total time complexity and total memory, respectively.

Consequently the basic TMD-TO (over an error free sample) and the proposed TMD-TO approach mounted over data with errors appear as powerful ones when the target is to recover the internal state of a keystream generator of a moderate size.

It can be noted that for stream ciphers for example, this scenario introduces an interesting trade-off between D, L and e as for a fixed data length LD, the larger L is, the more powerful the code is (i.e. the larger e can be) but the fewer data blocks D are available.

4 Concluding Remarks

In this paper, a hybrid approach obtained from a mixture of two basic approaches referred to as approaches 1 and 2 have been presented for mounting TM-TO and TMD-TO over noisy data. In approach-1, the same table as for the noiseless case is constructed, and all possible error patterns associated to the truncation of the received sample to the key size are considered for processing. In approach-2, the tables that cover all possible noisy versions of the keys are constructed and the truncation of the noisy received sample is used only for processing. Particularly note that approach-1 does not require an exhaustive search over all possible error vectors which corrupt the segment employed for cryptanalysis, but only the fraction of these error-patterns that corrupt the first K bits. This appears as a consequence of the error-correction capability introduced by expanding the K-bit input vector into the L-bit output one (its corrupted version is available for cryptanalysis) which can be considered as a nonlinear error-correction code. The error-capability of the underlying code determines the level of data corruption which can be processed and this feature depends on the encryption algorithm considered.

The proposed hybrid approach provides a unified framework to compare the effectiveness of the basic two approaches. The analysis implies that approach-1 is the most efficient one for this cryptanalytic scenario. The main reason is due to the fact that the TM-TO and TMD-TO corresponding to approach-1 include (only) the additional time complexity to process error vectors with respect to the noiseless case (which becomes the special case that processes the all-0 error vector only).

The reported results can provide a background for a number of further (research) directions including the following ones: (i) consideration of the error-correction capability of particular encryption techniques, and (ii) employment of the TM-TO and TMD-TO over noisy data for cryptanalysis of certain encryption schemes obtained by suitable approximation of the original ones where the given sample for cryptanalysis appears as a noisy version of the sample which generates the approximated scheme (see [12], for example).

References

1. E. Barkan and E. Biham, "Conditional Estimators: An Effective Attack on A5/1," SAC2005, *Lecture Notes in Computer Science*, vol. 3897, pp. 1-19, 2006.
2. E. Barkan, E. Biham and A. Shamir, "Rigorous Bounds on Cryptanalytic Time/Memory Tradeoffs," CRYPTO 2006, *Lecture Notes in Computer Science*, vol. 4117, pp. 1-21, Aug. 2006.
3. E. Barkan, *Cryptanalysis of Ciphers and Protocols*, PhD Thesis, Technion, Israel, 2006.
4. A. Biryukov and A. Shamir, "Cryptanalytic Time/Memory/Data Tradeoffs for Stream Ciphers," ASIACRYPT 2000, *Lecture Notes in Computer Science*, vol. 1976, pp. 1-13, 2000.

5. O. Dunkelman and N. Keller, "Treatment of the Initial Value in Time-Memory-Data Tradeoff Attacks on Stream Ciphers," *Information Processing Letters*, vol. 107, pp. 133-137, 2008.
6. A. Fiat and M. Naor, "Rigorous Time/Space Trade-Offs for Inverting Functions," *SIAM J. Computing*, vol. 29, pp. 790-803, 1999.
7. J.D. Golić, "Cryptanalysis of Three Mutually Clock-Controlled Stop/Go Shift Registers," *IEEE Trans. Inform. Theory*, vol. 46, pp. 1081-1089, May 2000.
8. J.I. Hall, *Notes on Coding Theory*, 2003, available at http://www.mth.msu.edu/ jhall.
9. M.E. Hellman, "A Cryptanalytic Time-Memory Trade-Off," *IEEE Trans. Inform. Theory*, vol. 26, pp. 401-406, July 1980.
10. A.J. Menezes, P.C. van Oorschot and S.A. Vanstone, *Handbook of Applied Cryptography*, Boca Roton: CRC Press, 1997.
11. M. J. Mihaljević, M. P. C. Fossorier and H. Imai, "Security Evaluation of Certain Broadcast Encryption Schemes Employing a Generalized Time-Memory-Data Trade-Off," *IEEE Communications Letters*, vol.11, pp. 988-990, Dec. 2007.
12. M. J. Mihaljević, M. P. C. Fossorier and H. Imai, "A General Formulation of Algebraic and Fast Correlation Attacks Based on Dedicated Sample Decimation," AAECC2006, *Lecture Notes in Computer Science*, vol. 3857, pp. 203-214, Feb. 2006.
13. P. Oechslin, "Making a Faster Cryptanalytic Time-Memory Trade-Off," CRYPTO 2003, *Lecture Notes in Computer Science*, vol. 2729, pp. 617-630, 2003.

Algebraic Precomputations in Differential Cryptanalysis

Martin Albrecht*[1], Carlos Cid[1], Thomas Dullien[2], Jean-Charles Faugère[3], and Ludovic Perret[3]

[1] Information Security Group, Royal Holloway, University of London
Egham, Surrey TW20 0EX, United Kingdom
{M.R.Albrecht,carlos.cid}@rhul.ac.uk
[2] Lehrstuhl für Kryptologie und IT-Sicherheit, Ruhr-Universität Bochum
44780 Bochum, Germany
Thomas.Dullien@ruhr-uni-bochum.de
[3] SALSA Project - INRIA (Centre Paris-Rocquencourt)
UPMC, Univ Paris 06 - CNRS, UMR 7606, LIP6
104, avenue du Président Kennedy 75016 Paris, France
jean-charles.faugere@inria.fr, ludovic.perret@lip6.fr

Abstract. Algebraic cryptanalysis is a general tool which permits one to assess the security of a wide range of cryptographic schemes. Algebraic techniques have been successfully applied against a number of multivariate schemes and stream ciphers. Yet, their feasibility against block ciphers remains the source of much speculation. At FSE 2009 Albrecht and Cid proposed to combine differential cryptanalysis with algebraic attacks against block ciphers. The proposed attacks required Gröbner basis computations during the online phase of the attack. In this work we take a different approach and only perform Gröbner basis computations in a pre-computation (or offline) phase. In other words, we study how we can improve "classical" differential cryptanalysis using algebraic tools. We apply our techniques against the block ciphers PRESENT and KTANTAN32.

1 Introduction

Algebraic cryptanalysis is a general tool which permits one to assess the security of a wide range of cryptographic schemes [2, 19, 18, 17, 15, 16, 21, 23, 22, 20]. As pointed out in the report [11], "*the recent proposal and development of algebraic cryptanalysis is now widely considered an important breakthrough in the analysis of cryptographic primitives*". It is a powerful technique that applies potentially to a wide range of cryptosystems – amongst them block ciphers, which are the main concern of this paper.

The basic principle of algebraic cryptanalysis is to model a cryptographic primitive by a set of algebraic equations. The system of equations is constructed in such a way as to have a correspondence between the solutions of this system, and a secret information of the cryptographic primitive (for instance, the secret key of a block cipher). The secret can thus be derived by solving the equation system.

Algebraic techniques have been successfully applied against a number of multivariate schemes and in stream cipher cryptanalysis. On the other hand, their feasibility against block ciphers remains the source of much speculation [13, 12, 2, 17]. The sizes of the resulting equation systems are usually beyond the capabilities of current solving algorithms. Furthermore, the complexity estimates are complicated as the algebraic systems are highly structured; a situation where known complexity bounds are no longer valid.

While it is currently infeasible to *cryptanalyse* a block cipher by algebraic means alone, these techniques nonetheless have practical implications for block cipher cryptanalysis. For instance,

* This author was supported by the Royal Holloway Valerie Myerscough Scholarship.

Albrecht and Cid [1] proposed at FSE 2009 to combine differential cryptanalysis with algebraic attacks against block ciphers and demonstrated the feasibility of their techniques against reduced-round versions of the block cipher PRESENT [5]. In this approach, the key recovery was approached by solving (or showing lack of solutions in) equation systems that were much simpler than the full cipher.

In this paper, we elaborate on this approach. That is, we further shift the focus away from attempting to solve the full system of equations. It turns out that significant information can be gained without solving the equation system in the classical sense. Additionally to PRESENT, we also apply these concepts to the block cipher KTANTAN32 [9].

We recall that differential cryptanalysis was formally introduced by Eli Biham and Adi Shamir at Crypto'90 [4], and has since been successfully used to attack a wide range of block ciphers. In its basic form, the attack can be used to distinguish an n-bit block cipher from a random permutation. By considering the distribution of output *differences* for the non-linear components of the cipher (e.g. the S-Box), the attacker may be able to construct *differential characteristics* $P^{'} \oplus P^{''} = \Delta P \rightarrow \Delta C = C^{'} \oplus C^{''}$ for a number of rounds N that are valid with probability p. If $p \gg 2^{-n}$, then by querying the cipher with a large number of plaintext pairs with prescribed difference ΔP, the attacker may be able to distinguish the cipher from a random permutation by counting the number of pairs with the output difference predicted by the characteristic. A pair for which the characteristic holds is called a *right pair*.

By modifying the attack, one can use it to recover key information. Instead of characteristics for the full N-round cipher, the attacker considers characteristics valid for r rounds only ($r = N - R$, with $R > 0$). If such characteristics exist with non-negligible probability the attacker can guess some key bits of the last rounds, partially decrypt the known ciphertexts, and verify if the result matches the one predicted by the characteristic. Candidate (last round) keys are counted, and as random noise is expected for wrong key guesses, eventually a peak may be observed in the candidate key counters, pointing to the correct round key.

The number of right pairs that are needed to distinguish the right candidate key depends on the probability p of the characteristic, the number k of simultaneous subkey bits that are used for the practical decryptions and counted, the average count α of how many keys are suggested per analysed pair (excluding the wrong pairs than can be discarded before the counting), and the fraction β of the analysed pairs among all the pairs. If an attacker is looking for k subkey bit, they can count the number of occurrences of the 2^k possible key values in 2^k counters. The counters contain an average of $(m \cdot \alpha \cdot \beta)/2^k$ counts, where m is the number of pairs, $m \cdot \beta$ is the expected number of pairs to analyse, and α is the number of suggested keys on average. Since these suggestions are spread across 2^k counters, we divide by 2^k. The right subkey value is counted $m \cdot p$ times by the right pairs, plus the random counts for all the possible subkeys. The *signal to noise ratio* is therefore:

$$S/N = \frac{m \cdot p}{m \cdot \alpha \cdot \beta / 2^k} = \frac{2^k \cdot p}{\alpha \cdot \beta}.$$

Note that it would be sufficient to consider the probability p of the differential – i.e. the sum of all p_i for all characteristics with $\Delta P \rightarrow \Delta C$ – instead of the probability of the characteristic. However, in practice authors often work with the probabilities of characteristics because it is easier to estimate them.

Albrecht and Cid proposed in [1] three techniques (so-called *Attack-A*, *Attack-B* and *Attack-C*) which require Gröbner basis computations during the online phase of the attack. This limitation prevented them from applying their techniques to PRESENT-80 with more than 16 rounds, since computation time would exceed exhaustive key search. In this work we take a different approach and only perform Gröbner basis computations in a pre-computation (or offline) phase. That is, we

study how we can improve "classical" differential cryptanalysis using the algebraic tools available to us. More specifically, we aim to increase the signal to noise ratio S/N using algebraic techniques.

The paper is organised as follows. In Section 2 we establish the notation used throughout the paper. In Section 3 we provide a high level description of the main idea behind this work. In Section 4 we briefly describe the ciphers which we use to demonstrate our ideas. These ideas are applied to reduce the noise in Section 5 and to improve the signal in Section 6. Experimental results are also presented in Sections 5 and 6.

2 Notation

We consider N-round block ciphers with a B_s-bit blocksize and K_s-bit key size. When we consider substitution-permutation networks (SPN) we denote the inputs to the S-Box layer as X and the outputs as Y. We always consider the parallel encryption of two plaintexts $P' = (P'_0, \ldots, P'_{B_s-1})$ and $P'' = (P''_0, \ldots, P''_{B_s-1})$ which are related by the input difference $\Delta P = (\Delta P_0, \ldots, \Delta P_{B_s-1})$. Thus we have $P'_i \oplus P''_i = \Delta P_i$ for $0 \leq i < B_s$. We consider $r < N$ round differential characteristics and have that $N - r = R$. If a differential characteristic predicts that the j-th inputs to the i-th S-Box layer application are related by the difference $\Delta X_{i,j}$, then given the plaintext difference ΔP, we have that $X'_{i,j} \oplus X''_{i,j} = \Delta X_{i,j}$ is true with some non-negligible probability. The characteristic also predicts that the j-th outputs of the i-th S-box layer application are related by the difference $\Delta Y_{i,j}$.

Finally, we denote the equation system encoding the encryption of P' to C' under the key K as F' and the ideal spanned by the $f \in F'$ as I' (similarly for P'' and C''). If we write $X_{i,j}$ we refer to both $X'_{i,j}$ and $X''_{i,j}$ (similarly for $Y_{i,j}$). In general we start counting at zero, except for the rounds, which we start counting at one.

3 Main Idea

The main idea involves shifting the emphasis of previous algebraic attacks away from attempting to solve a equation system towards *using ideal membership as implication*. Instead of trying to solve an equation system arising from the cipher, we use Gröbner basis methods to calculate what a particular differential pattern *implies*.

To explain the main idea we start with a small example. Consider the 4-bit S-Box of PRESENT [5]. The S-Box can be completely described by a set of polynomials that express each output bit in terms of the input bits. One can consider a *pair* of input bits $X'_{1,0}, \ldots, X'_{1,3}$ and $X''_{1,0}, \ldots, X''_{1,3}$ and their respective output bits $Y'_{1,0}, \ldots, Y'_{1,3}$ and $Y''_{1,0}, \ldots, Y''_{1,3}$. Since the output bits are described as polynomials in the input bits, it is easy to build a set of polynomials describing the parallel application of the S-Box to the pair of input bits. Assume the fixed input difference of $(0, 0, 0, 1)$ holds for this S-Box. This can be described algebraically by adding the polynomials $X'_{1,3}+X''_{1,3} = 1$, $X'_{1,j} + X''_{1,j} = 0$ for $0 \leq j < 3$ to the set. As usual, the field equations are also added.

The set of equations now forms a description of the parallel application of the S-Box to two inputs with a fixed input difference. The ideal I spanned by these polynomials contains *all* polynomials that are *implied* by the set. If all equations in the generating set of the ideal evaluate to zero, it is clear that any element of I evaluates to zero. This means that *any equation in the ideal will always vanish* if it is assigned values generated by applying the S-Box to a pair of inputs with the above-mentioned input difference.

From a cryptographic point of view, it is important to understand what relations between output bits will hold for a particular input difference. As a consequence, we are looking for polynomials in *just the output bits* which are contained in I. Algebraically, we are trying to find elements of the ideal $I_Y = I \bigcap \mathbb{F}_2[Y'_{1,0}, \ldots, Y'_{1,3}, Y''_{1,0}, \ldots, Y''_{1,3}]$ where I is the ideal spanned by our original equations. A *deglex* Gröbner basis G_Y of this ideal can be computed using standard elimination techniques [3, p.168]. For this, we can set up a block or product ordering where all output variables are lexicographically smaller than any other variable in the system. In addition, we fix the *deglex* ordering among the output variables. Computing the Gröbner basis with respect to such an ordering gives us the Gröbner basis G_Y. We note that G_Y will contain the relations of lowest degree of I_Y due to the choice of term ordering. In our example, we obtain:

$$
\begin{aligned}
G_Y = [&Y'_{1,3} + Y''_{1,3} + 1,\\
&Y'_{1,0} + Y'_{1,2} + Y''_{1,0} + Y''_{1,2} + 1,\\
&Y''_{1,0}Y''_{1,2} + Y'_{1,2} + Y''_{1,0} + Y''_{1,1} + Y''_{1,3},\\
&Y''_{1,0}Y''_{1,1} + Y''_{1,0}Y''_{1,3} + Y''_{1,1}Y''_{1,2} + Y''_{1,2}Y''_{1,3} + Y'_{1,1} + Y''_{1,0} + Y''_{1,1},\\
&Y'_{1,2}Y''_{1,2} + Y''_{1,1}Y''_{1,2} + Y''_{1,2}Y''_{1,3},\\
&Y'_{1,2}Y''_{1,0} + Y''_{1,1}Y''_{1,2} + Y''_{1,2}Y''_{1,3} + Y'_{1,1} + Y'_{1,2} + Y''_{1,0} + Y''_{1,3},\\
&Y'_{1,1}Y''_{1,2} + Y'_{1,2}Y''_{1,1} + Y'_{1,2}Y''_{1,3} + Y''_{1,1}Y''_{1,2} + Y'_{1,1} + Y'_{1,2} + Y''_{1,1},\\
&Y'_{1,1}Y''_{1,1} + Y'_{1,1}Y''_{1,3} + Y''_{1,1}Y''_{1,2} + Y''_{1,1}Y''_{1,3} + Y''_{1,2}Y''_{1,3} + Y''_{1,1},\\
&Y'_{1,1}Y''_{1,0} + Y'_{1,2}Y''_{1,1} + Y'_{1,2}Y''_{1,3} + Y''_{1,0}Y''_{1,3} + Y''_{1,1}Y''_{1,2} + Y''_{1,2}Y''_{1,3} + Y'_{1,1} + Y''_{1,3},\\
&Y'_{1,1}Y'_{1,2} + Y'_{1,2}Y''_{1,3} + Y''_{1,1}Y''_{1,2} + Y''_{1,2}Y''_{1,3} + Y'_{1,2}].
\end{aligned}
$$

There is no other linear or quadratic polynomial $p \in I_Y$ which is not a simple algebraic combination of the polynomials in G_Y.

Of course, we can ask different questions instead of looking for low degree equations. For instance, we can query whether there are equations in the bits $Y'_{1,3}, Y''_{1,2}, Y''_{1,3}$ induced by the input difference by setting up the appropriate term ordering.

In order to formalise this idea, consider a function E (for example a block cipher). Assume that E can be expressed as a set of algebraic equations F over $\mathbb{F}$. If one application of the function can be described as a set of equations, d parallel applications to d different inputs (which we denote $P_0, \ldots, P_{d-1}$) can also be described as a set of equations. We call the set of equations relating the i-th input and output E_i and the matching polynomial system F_i. The outputs of these equations are called $C_0, \ldots, C_{d-1}$. Furthermore, assume any property Λ which holds on $P_0, \ldots, P_{d-1}$ and which can be expressed in a set of algebraic equations F_Λ. A natural question to ask is: How do properties on $P_0, \ldots, P_{d-1}$ affect properties on $C_0, \ldots, C_{d-1}$? We combine the sets of polynomials $\overline{F} = F_\Lambda \cup (\bigcup_{i=0}^{d-1} F_i)$ and consider the Ideal $I = \langle \overline{F} \rangle$ spanned by $\overline{F}$. Next, we compute the unique reduced Gröbner basis G_C of the ideal $I_C = I \cap \mathbb{F}[C_0, \ldots, C_{d-1}]$. Now G_C contains all "relevant" polynomials in $C_0, \ldots, C_{d-1}$, where "relevant" is determined by the term ordering.

As soon as we can compute the Gröbner basis G_C for the function E then we only need to collect the right polynomials from G_C. However, for many functions E computing G_C seems infeasible using current Gröbner basis techniques, implementations and computing power. Thus we have to relax some conditions hoping that we still can recover some equations using a similar technique. We provide below a few heuristics and techniques which still allow recovering *some* relevant equations.

Early Abort. To recover some properties we might not need to compute the complete Gröbner basis, instead we may opt to stop the computation at some degree D.

Replacing Symbols by Constants. It is possible to replace the symbols $P_0, \ldots, P_{d-1}$ by some constants satisfying the constraint Λ which further simplifies the computation. Of course any

polynomial recovered from such a computation would have to be checked against other values to verify that it actually holds in general or with high probability.

Choosing a Different Term Ordering. Instead of computing with respect to an elimination ordering, which is usually more expensive than a degree compatible ordering, we may choose to perform our computations with respect to an easier ordering such as *degrevlex*. Used together with **Early Abort**, we have no assurances about the uniqueness and completeness of the recovered system. However, we might still be able to recover some information.

Computing Normal Forms Only. We can also compute equations by computing normal forms only. For many ciphers it is possible to construct a Gröbner basis for the round transformation [7, 8] with respect to some elimination ordering without any polynomial reductions. These constructions exploit the fact that a system of polynomials is a Gröbner basis if each polynomial has a leading term which is pairwise prime with every other leading term. Using this property, we may construct a Gröbner basis for some elimination ordering for the inverse of the cipher, i.e. the decryption process, such that the input variables are lexicographically bigger than the output variables of some round. Furthermore, we construct the term ordering such that the variables of round $i-1$ are lexicographically bigger than the variables for round i. Furthermore, the symbols C_i are the lexicographically smallest.

If G' is such a Gröbner basis for r rounds for the first encryption and G'' such a Gröbner basis for r rounds for the second encryption, we can combine these bases to $G = G' \cup G''$ which still is a Gröbner basis. Now we can compute the normal form of $X'_i + X''_i + \Delta X_i$ with respect to G. This will eliminate all variables $> C_i$ as much as possible by construction. If this computation does not give equations in the C_i only, we may opt to perform an interreduction on several such equations hoping that this way the remaining key bits are eliminated. For example, such a Gröbner basis for one application of the PRESENT S-Box is

$$\begin{aligned}
&X_{1,0} + Y_{1,0}Y_{1,1}Y_{1,3} + Y_{1,1}Y_{1,2}Y_{1,3} + Y_{1,2}Y_{1,3} + Y_{1,0} + Y_{1,1} + Y_{1,2} + Y_{1,3},\\
&X_{1,1} + Y_{1,0}Y_{1,1}Y_{1,3} + Y_{1,0}Y_{1,2}Y_{1,3} + Y_{1,1}Y_{1,2}Y_{1,3} + Y_{1,0}Y_{1,2} + Y_{1,0}Y_{1,3}\\
&\quad + Y_{1,1}Y_{1,2} + Y_{1,1}Y_{1,3} + Y_{1,2}Y_{1,3} + Y_{1,0} + 1,\\
&X_{1,2} + Y_{1,0}Y_{1,1}Y_{1,3} + Y_{1,0}Y_{1,2}Y_{1,3} + Y_{1,1}Y_{1,2}Y_{1,3} + Y_{1,0}Y_{1,1} + Y_{1,0}Y_{1,2}\\
&\quad + Y_{1,1}Y_{1,3} + Y_{1,0} + Y_{1,2} + Y_{1,3},\\
&X_{1,3} + Y_{1,0}Y_{1,2} + Y_{1,1} + Y_{1,3} + 1
\end{aligned}$$

The normal form of the equation $X'_{1,3} + X''_{1,3} + 1$ with respect to G (i.e. two of such systems for X', Y' and X'', Y'') is $Y'_{1,0}Y'_{1,2} + Y''_{1,0}Y''_{1,2} + Y'_{1,1} + Y'_{1,3} + Y''_{1,1} + Y''_{1,3} + 1$.

4 Case Studies

To demonstrate our techniques, we consider the block ciphers PRESENT and KTANTAN32.

PRESENT [5] was proposed at CHES 2007 as an ultra-lightweight block cipher, enabling a very compact implementation in hardware, and therefore particularly suitable for RFIDs and similar devices. There are two variants of PRESENT: one with 80-bit keys and one with a 128-bit keys, denoted as PRESENT-80 and PRESENT-128 respectively.

PRESENT is an SP-network with a blocksize of 64 bits and both versions have 31 rounds. Each round of the cipher has three layers of operations: `keyAddLayer`, `sBoxLayer` and `pLayer`. The operation `keyAddLayer` is a simple subkey addition to the current state, while the `sBoxLayer` operation consists of 16 parallel applications of a 4-bit S-Box. The operation `pLayer` is a permutation of wires.

The PRESENT authors give a security analysis of their cipher by showing resistance against well-known attacks such as differential and linear cryptanalysis [5]. The best published differential attacks are for 16 rounds of PRESENT-80 [27] and 17 (and possibly up to 19) rounds [1] for PRESENT-128. Results on linear cryptanalysis for up to 26 rounds are available in [10, 24]. Bit-pattern based integral attacks [28] are successful up to seven rounds of PRESENT. A new type of attack, called statistical saturation attack, was proposed in [14] and shown to be applicable up to 24 rounds of PRESENT.

KTANTAN32 [9] was proposed at CHES 2009 and is the smallest cipher in a family of block ciphers proposed in [9]. It allows a very compact implementation in hardware. It has a blocksize of 32 bits and accepts an 80-bit key. The input is loaded into two registers L_2 and L_1 of 19 and 13 bit length respectively. A round transformation is then applied to these registers 254 times. This round function updates two bits using a quadratic function and performs rotations on the registers. After 254 rounds the content of L_2 and L_1 is outputed as the ciphertext.

The designers of KTANTAN consider a wide range of attacks in their security argument and show the cipher secure against differential, linear, impossible differential, algebraic attacks and some combined attacks. So far, there are no third-party security analyses of the cipher of which the authors are aware.

5 Reducing the Noise

Recall that in order to discard wrong pairs, [1] proposed a technique referred to as *Attack-C*. In this context, the attacker considers an equation system modelling only the rounds $> r$. The attacker is left with R rounds for each plaintext–ciphertext pair to consider. These are related by the output difference predicted by the differential. If we denote the equation system for the last R rounds of the encryption of P' to C' or P'' to C'' as F'_R or F''_R respectively. The algebraic part of *Attack-C* is a Gröbner basis computation on the polynomial system

$$F = F'_R \cup F''_R \cup \{X'_{r+1,i} + X''_{r+1,i} + \Delta X_{r+1,i} \mid 0 \leq i < B_s\}.$$

Whenever the Gröbner basis is equivalent to $\{1\}$, we know that the analysed pair could not have been a right pair. Thus the pair can be discarded. However, no strong assurances are given in [1] as to how many pairs are actually discarded by this technique[4].

In the this work, we consider the same system of equations as in *Attack-C* but replace the tuples of constants C' and C'' by tuples of symbols. By computing a Gröbner basis for the right elimination ordering (cf. Section 3), we can recover equations in the variables C' and C'' which must evaluate to zero on the actual ciphertext values as soon as the input difference for round $r+1$ holds. Once we recovered such equations we can calculate the probability that all these polynomials evaluate to zero for random values for C' and C''. This gives an estimate about the quality of the filter. Furthermore, if the equations in C' and C'' are of sufficiently small degree, this filter is much faster than *Attack-C* since no Gröbner basis has to be computed for each pair.

5.1 Case Study: PRESENT

We consider the differential from [27] and construct filters for PRESENT reduced to $14+R$ rounds. The same filter applies also to $10+R$, $6+R$ and $2+R$ rounds since the characteristic is iterative with a period of four rounds. The explicit polynomials in this section do not differ for PRESENT-80 and PRESENT-128.

[4] Note that *Attack-B* in [1] is guaranteed to distinguish right pairs eventually.

PRESENT 1R. We consider the polynomial ring $P =$

$$\begin{aligned}\mathbb{F}_2[\ & K_{0,0},\ldots,K_{0,79},\ K_{1,0},\ldots,K_{1,3},\\ & Y'_{1,0},\ldots,Y'_{1,63},\quad Y''_{1,0},\ldots,Y''_{1,63},\quad X'_{1,0},\ldots,X'_{1,63},\quad X''_{1,0},\ldots,X''_{1,63},\\ & \ldots,\qquad\qquad K_{15,0},\ldots,K_{15,3},\\ & Y'_{15,0},\ldots,Y'_{15,63},\ Y''_{15,0},\ldots,Y''_{15,63},\ X'_{15,0},\ldots,X''_{15,63},\ X''_{15,0},\ldots,X''_{15,63},\\ & C'_0,\ldots,C'_{63},\qquad C''_0,\ldots,C''_{63}]\end{aligned}$$

and attach the following block ordering:

$$\underbrace{K_{0,0},\ldots,X''_{15,63}}_{\text{degrevlex}},\underbrace{C'_0,\ldots,C''_{63},C''_0,\ldots,C''_{63}}_{\text{degrevlex}}.$$

We set up an equation system as in *Attack-C* of [1], except that the ciphertext bits are symbols (C'_i and C''_i). Then, we compute the Gröbner basis up to degree $D = 3$ using PolyBoRi 0.6.3 [6] (as shipped with the Sage [26] mathematics software) with the option `deg_bound=3` and filter out any polynomial that contains non-ciphertext variables.

This computation returns 60 polynomials of which 58 are linear. These 58 linear polynomials are of the form $C'_i + C''_i$ for

$$i \in \{0,\ldots,6,8,\ldots 14,16,\ldots 22,24,\ldots 30,32,\ldots 38,40,\ldots 46,48,\ldots,63\}.$$

The remaining two polynomials are $(C'_{23} + C''_{23} + 1)(C'_7 + C'_{39} + C''_7 + C''_{39} + 1)$ and $(C'_{31} + C''_{31} + 1)(C'_{15} + C'_{47} + C''_{15} + C''_{47} + 1)$. The probability that all polynomials evaluate to zero on a random point is approximately $2^{-58.83}$.

PRESENT 2R. We extend the ring and the system from the 1R experiment in the obvious way and perform the same computation as before. This computation returns 65 polynomials of which 46 are linear. Forty linear polynomials are of the form $C'_i + C''_i$ and encode the information that the last round output difference of 10 S-Boxes must be zero (cf. [27]). The remaining 24 polynomials split into two sets F_0, F_2 of 12 polynomials in 24 variables each and the F_j do not share any variables with each other or the first 40 linear polynomials. The systems F_j are listed in Figure 2 in the Appendix. The probability that all polynomials evaluate to zero for a random point is $\approx 2^{-50.669}$.

For comparison, we construct random pairs C', C'' which pass this polynomial filter and notice that for *Attack-C* from [1] roughly every second such pair for Present-80 and 317 out of 512 for Present-128 will pass. Thus we expect *Attack-C* to pass with probability $\approx 2^{-51.669}$ for Present-80 and with probability $\approx 2^{-51.361}$ for Present-128. Finally, we recall that Wang's filter from [27] passes with probability $2^{-40} \cdot (5/16)^6 \approx 2^{-50.07}$.

PRESENT 3R. We extend the ring and the block ordering in the obvious way and compute a Gröbner basis with degree bound 3. The computation returns 28 polynomials of which 16 are linear. The linear polynomials have the form $C'_i + C''_i$ for

$$i \in \{3,7,11,15,19,23,27,31,35,39,43,47,51,55,59,63\}.$$

The remaining 12 polynomials are quadratic and cubic (cf. Figure 3 in the Appendix). The probability that all polynomials evaluate to zero on a random point is $\approx 2^{-18.296}$. To compare with *Attack-C*, we construct random pairs C', C'' which pass this polynomial filter. *Attack-C* will accept roughly 6 in 1024 pairs for Present-80 and 9 out of 1024 pairs for Present-128. Thus, we expect *Attack-C* to pass with probability $\approx 2^{-25.711}$ for Present-80 and $2^{-25.126}$ for Present-128.

PRESENT 4R. We extend the ring and the block ordering in the obvious way. With a degree bound $D = 3$ we recover

$$(C'_{32+j} + C''_{32+j} + 1)(C'_j + C''_j + 1)(C'_{16+j} + C'_{48+j} + C''_{16+j} + C''_{48+j})$$

for $0 \leq j < 16$. The probability that all polynomials evaluate to zero on a random point is $\approx 2^{-3.082}$. We verified experimentally that this bound is optimal by using the SAT solver CryptoMiniSat [25] on *Attack-C* systems in a 4R attack against Present-80-14. The solver returned an assignment which satisfies the equation system with probability $\approx 2^{-3}$. Thus, we conclude that *Attack-C* will roughly accept 1 in 8 pairs.

5.2 Case Study: KTANTAN

In Table 1 we give our results against Ktantan32. We used the best differential for 42 rounds as provided by the KTANTAN designers and extended it to 71 rounds. The characteristic has a probability of 2^{-31}. Below we present results for the degree bounded at four and at five respectively. For each degree bounds we give the number of degree 1-5 polynomials (denoted as $d = *$) we found. In the last column of each experiment we give the approximate probability that all the equations we found evaluate to zero for random values (denoted $\log_2 p$).

	degree bound = 4						degree bound = 5					
N	$d=1$	$d=2$	$d=3$	$d=4$	$d=5$	$\log_2 p$	$d=1$	$d=2$	$d=3$	$d=4$	$d=5$	$\log_2 p$
72	32	0	0	0	0	-32.0	32	0	0	0	0	-32.0
74	32	0	0	0	0	-32.0	32	0	0	0	0	-32.0
76	32	0	0	0	0	-32.0	32	0	0	0	0	-32.0
78	31	3	0	0	0	-32.0	31	3	0	0	0	-32.0
80	28	11	0	0	0	-31.4	28	11	0	0	0	-31.4
82	25	23	0	0	0	-31.0	25	23	0	0	0	-31.0
84	20	32	4	8	0	-29.0	20	32	4	32	0	-29.0
86	16	44	19	8	0	-25.7	16	46	23	75	106	< -24
88	12	39	54	96	0	-24.0	12	51	103	371	745	< -23
90	8	41	129	287	0	-23.0	8	42	133	612	1762	< -22
92	4	28	113	285	0	-20.0	4	33	133	743	2646	-20.4
94	1	20	94	244	0	-16.3	1	25	124	662	2345	-18.5
96	0	8	38	96	0	-12.8	0	8	52	287	1264	-14.3
98	0	3	8	29	0	-7.0	0	3	10	46	156	-9.1
100	0	1	3	13	0	-3.7	0	1	3	18	47	-4.6
102	0	0	0	2	0	-0.8	0	0	0	4	9	-0.9
103	0	0	0	1	0	-0.4	0	0	0	2	4	-0.4
104	0	0	0	0	0	0.0	N/A	N/A	N/A	N/A	N/A	N/A

Table 1. Decreasing the noise for Ktantan32.

6 Increasing the Signal

In this section, we consider the problem of increasing the amount of correct data that has to agree and is always suggested by a right pair. Increasing this value usually has considerable costs attached to it. First, more data needs to be managed and thus usually the counter tables get bigger. On average, we can expect each additional bit considered to double the size of these tables. Second,

in order to generate more data, more partial decryptions must be performed which increases the computation time. Additionally, the number of key bits that can be trial decrypted might be limited by the number of rounds R we can consider because of the quality of the filter.

In this work we use (non-linear) data available from the first few rounds instead of the last R rounds. Assume that we have an SP-network, a differential characteristic $\Delta = (\Delta P, \Delta Y_1, \ldots, \Delta Y_r)$ valid for r rounds with probability p, and (P', P'') a right pair for Δ (so that $\Delta P = P' \oplus P''$ and ΔY_r holds for the output of round r). For simplicity, let us assume that only one S-Box is active in round 1, with input $X'_{1,j}$ and $X''_{1,j}$ (restricted to this S-Box) for the plaintext P' and P'' respectively, and that there is a key addition immediately before the S-Box operation, that is

$$S(P'_j \oplus K_{0,j}) = S(X'_{1,j}) = Y'_{1,j} \text{ and } S(P''_j \oplus K_{0,j}) = S(X''_{1,j}) = Y''_{1,j}.$$

The S-Box operation S can be described by a (vectorial) Boolean function, expressing each bit of the output $Y'_{1,j}$ as a polynomial function (over $\mathbb{F}_2$) on the input bits of $X'_{1,j}$ and $K_{0,j}$. If (P', P'') is a right pair, then the polynomial equations arising from the relation $\Delta Y_{1,j} = Y'_{1,j} \oplus Y''_{1,j} = S(P'_j \oplus K_{0,j}) \oplus S(P''_j \oplus K_{0,j})$ gives us a very simple equation system to solve, with only the key variables $K_{0,j}$ as unknowns (and which do not vanish identically because we are considering nonzero differences). Consequently, right pairs suggest additional information about the key from the first round difference. In particular, if ΔY_1 holds with probability 2^{-b} then we can recover b bits of information about the key, if we found a right pair.

There is no *a priori* reason to restrict this argument from [1] to the first round. Let Δ, r, P', P'' be as before. We setup two equation systems F' and F'' involving P', C' and P'', C'' respectively and discard any polynomials from the rounds $> s$ where s is a small integer > 0. Previously we had $s = 1$. Finally, we add linear equations as suggested by the characteristic and use this system to recover information about the key from the first s rounds.

In order to avoid the potentially costly Gröbner basis computation for every candidate pair replace the tuples of constants P' and P'' by tuples of symbols. According to Section 3, we can compute polynomials involving only key variables and the newly introduced plaintext variables P' and P''. Assume that we can indeed compute the Gröbner basis with P' and P'' symbols for the first s rounds and the linear equations arising from the characteristic added. Assume further that the probability that the characteristic restricted to s rounds holds is 2^{-b} and that we computed m_s polynomials in the variables K_0, P' and P''. This means that we recover b bits of information when we evaluate all m_s polynomials such that we replace P' and P'' by their actual values.

This means that we have b bits of extra information and thus can write $S/N = \frac{2^{k+b} \cdot p}{\alpha \cdot \beta}$ without the overhead of performing any partial decryptions. However, we have to perform m_s polynomial evaluations (where we replace P' and P'' by their actual values) of relatively small low degree polynomials.

Case Study: PRESENT. We consider the first two encryption rounds and the characteristic from [27]. We set up a polynomial ring with two blocks such that the variables P_i and K_i are lexicographically smaller than any other variables. Within the blocks we chose a degree lexicographical term ordering. We set up an equation system covering the first two encryption rounds and added the linear equations suggested by the characteristic. Then, we eliminated all linear leading terms which are not in the variables P_i and K_i and computed a Gröbner basis up to degree five. This computation returned 22 linear and quadratic polynomials (we give the Gröbner basis for these polynomials in Figure 4). This system gives 8 bits of information about the key. Note that the first two rounds of the characteristic pass with probability 2^{-8}.

Case Study: KTANTAN32. We consider the first 24 rounds of KTANTAN32 and compute the full Gröbner basis. This computation recovers 39 polynomials. We list an excerpt in Figure 1 in the Appendix. As expected we observe that the characteristic also imposes restrictions on the plaintext. These eight equations allow us to recover up to four bits (depending on the value of P'_{19}) of information about the key.

Acknowledgements

We would like to thank William Stein for allowing us to run our experiments on his computers[5]. We would also like to thank anonymous referees for helpful comments.

Conclusion

In this work, we have introduced a novel application for the algebraic cryptanalysis of block ciphers. We propose a method which can improve "classical" differential cryptanalysis, by applying algebraic tools in a pre-computation phase. As such, we shift the focus from attempting to solve large systems of polynomial equations to recovering symbolic information about the underlying cipher. Although the use of algebraic techniques in general, and Gröbner basis methods in particular, in block cipher cryptanalysis has received some criticism within the cryptographic community, as it has been often the case that "simpler" techniques can perform favourably in many situations, we stress that the rich algebraic structure of Gröbner basis can offer many advantages and may give one a more subtle insight of the cipher structure. This can in turn be used in the cryptanalysis of the cipher. We note that *in principle* our techniques can recover an optimal amount of information and that in almost all cases considered in this work we were (almost) able to accomplish this. We expect that this approach is applicable to other cryptanalytical techniques such as linear and higher-order differential cryptanalysis and consider applying it as an area of future work.

References

1. Martin Albrecht and Carlos Cid. Algebraic Techniques in Differential Cryptanalysis. In *Fast Software Encryption 2009*, Lecture Notes in Computer Science, Berlin, Heidelberg, New York, 2009. Springer Verlag.
2. Gwenole Ars. *Applications des bases de Gröbner à la cryptographie*. PhD thesis, Université de Rennes I, 2005.
3. Thomas Becker and Volker Weispfenning. *Gröbner Bases - A Computational Approach to Commutative Algebra*. Springer Verlag, Berlin, Heidelberg, New York, 1991.
4. Eli Biham and Adi Shamir. Differential Cryptanalysis of DES-like Cryptosystems. In *Advances in Cryptology — CRYPTO 1990*, volume 537 of *Lecture Notes in Computer Science*, pages 3–72, Berlin, Heidelberg, New York, 1991. Springer Verlag.
5. Andrey Bogdanov, Lars R. Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew J. B. Robshaw, Yannick Seurin, and C. Vikkelsoe. PRESENT: An ultra-lightweight block cipher. In *Cryptographic Hardware and Embedded Systems - CHES 2007*, volume 7427 of *Lecture Notes in Computer Science*, pages 450–466, Berlin, Heidelberg, New York, 2007. Springer Verlag. Available at `http://www.crypto.rub.de/imperia/md/content/texte/publications/conferences/present_ches2007.pdf`.
6. Michael Brickenstein and Alexander Dreyer. PolyBoRi: A framework for Gröbner basis computations with Boolean polynomials. In *Electronic Proceedings of MEGA 2007*, 2007. Available at `http://www.ricam.oeaw.ac.at/mega2007/electronic/26.pdf`.

[5] Purchased under National Science Foundation Grant No. DMS-0821725.

7. Johannes Buchmann, Andrei Pychkine, and Ralf-Philipp Weinmann. A Zero-dimensional Gröbner Basis for AES-128. In *Fast Software Encryption 2006*, volume 4047 of *Lecture Notes in Computer Science*, pages 78–88. Springer Verlag, 2006.
8. Stanislav Bulygin and Michael Brickenstein. Obtaining and solving systems of equations in key variables only for the small variants of AES. *Mathematics in Computer Science*, 2008. conditionally accepted for special issue "Symbolic Computation and Cryptography".
9. Christophe De Cannière, Orr Dunkelman, and Miroslav Knežević. KATAN and KTANTAN — a family of small and efficient hardware-oriented block ciphers. In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems - CHES 2009*, volume 5747 of *Lecture Notes in Computer Science*, pages 272–288. Springer Verlag, 2009.
10. Joo Yeon Cho. Linear cryptanalysis of reduced-round PRESENT. Cryptology ePrint Archive, Report 2009/397, 2009. available at `http://eprint.iacr.org/2009/397`.
11. Carlos Cid. D.STVL.7 algebraic cryptanalysis of symmetric primitives, 2008. available at `http://www.ecrypt.eu.org/ecrypt1/documents/D.STVL.7.pdf`.
12. Carlos Cid and Gaëtan Leurent. An Analysis of the XSL Algorithm. In *Advances in Cryptology — ASIACRYPT 2005*, volume 3788 of *Lecture Notes in Computer Science*, pages 333–352, Berlin, Heidelberg, New York, 2005. Springer Verlag.
13. Carlos Cid, Sean Murphy, and Matthew Robshaw. Small Scale Variants of the AES. In *Fast Software Encryption 2005*, volume 3557 of *Lecture Notes in Computer Science*, pages 145–162, Berlin, Heidelberg, New York, 2005. Springer Verlag. Available at `http://www.isg.rhul.ac.uk/~sean/smallAES-fse05.pdf`.
14. Baudoin Collard and Francois-Xavier Standaert. A Statistical Saturation Attack against the Block Cipher PRESENT. In *Topics in Cryptology – CT-RSA 2009*, pages 195–210, 2009.
15. Nicolas T. Courtois. Fast algebraic attacks on stream ciphers with linear feedback. In *Advances in Cryptology – CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 176–194, 2003.
16. Nicolas T. Courtois. Higher order correlation attacks, XL algorithm and cryptanalysis of Toyocrypt. In P.J. Lee and C.H. Lim, editors, *Information Security and Cryptology - ICISC 2002: 5th International Conference*, volume 2587 of *Lecture Notes in Computer Science*, Berlin, Heidelberg, New York, 2003. Springer Verlag.
17. Nicolas T. Courtois and Gregory V. Bard. Algebraic Cryptanalysis of the Data Encryption Standard. In Steven D. Galbraith, editor, *Cryptography and Coding – 11th IMA International Conference*, volume 4887 of *Lecture Notes in Computer Science*, pages 152–169, Berlin, Heidelberg, New York, 2007. Springer Verlag. pre-print available at `http://eprint.iacr.org/2006/402`.
18. Nicolas T. Courtois and Willi Meier. Algebraic attacks on stream ciphers with linear feedback. In Eli Biham, editor, *Advances in Cryptology — EUROCRYPT 2003*, volume 2656 of *Lecture Notes in Computer Science*, pages 345–359, Berlin, Heidelberg, New York, 2003. Springer Verlag.
19. Nicolas T. Courtois and Josef Pieprzyk. Cryptanalysis of block ciphers with overdefined systems of equations. In Yuliang Zheng, editor, *Advances in Cryptology — ASIACRYPT 2002*, volume 2501 of *Lecture Notes in Computer Science*, pages 267–287, Berlin, Heidelberg, New York, 2002. Springer Verlag.
20. Jean-Charles Faugère, Françoise Levy dit Vehel, and Ludovic Perret. Cryptanalysis of MinRank. In *Advances in Cryptology – CRYPTO 2008*, volume 5157 of *Lecture Notes in Computer Science*, pages 280–296, 2008.
21. Jean-Charles Faugère and Antoine Joux. Algebraic cryptanalysis of hidden field equation (hfe) cryptosystems using gröbner bases. In *CRYPTO*, pages 44–60, 2003.
22. Jean-Charles Faugère and Ludovic Perret. Cryptanalysis of $2r^-$ schemes. In *Advances in Cryptology – CRYPTO 2006*, volume 4117 of *Lecture Notes in Computer Science*, pages 357–372, 2006.
23. Jean-Charles Faugère and Ludovic Perret. Polynomial equivalence problems: Algorithmic and theoretical aspects. In *Advances in Cryptology – EUROCRYPT 2006*, volume 4004 of *Lecture Notes in Computer Science*, pages 30–47, 2006.
24. Jorge Nakahara Jr, Pouyan Seperhdad, Bingsheng Zhang, and Meiqin Wang. Linear (hull) and algebraic cryptanalysis of the block cipher PRESENT. In *The 8th International Conference on Cryptoplogy and Network Security - CANS 2009*, 2009.
25. Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT solvers to cryptographic problems. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009*, volume 5584 of *Lecture Notes in Computer Science*, pages 244–257, Berlin, Heidelberg, New York, 2009. Springer Verlag.

26. William Stein et al. *SAGE Mathematics Software*. The Sage Development Team, 2008. Available at `http://www.sagemath.org`.
27. Meiqin Wang. Differential Cryptanalysis of reduced-round PRESENT. In Serge Vaudenay, editor, *Africacrypt 2008*, volume 5023 of *Lecture Notes in Computer Science*, pages 40–49, Berlin, Heidelberg, New York, 2008. Springer Verlag.
28. Muhammad Reza Z'Aba, Håvard Raddum, Matt Henricksen, and Ed Dawson. Bit-pattern based integral attacks. In Kaisa Nyberg, editor, *Fast Software Encryption 2008*, number 5086 in Lecture Notes In Computer Science, pages 363–381, Berlin, Heidelberg, New York, 2008. Springer Verlag.

A Explicit Polynomials

$$
\begin{aligned}
&(P'_{19}+1)(P'_3P'_8+P'_{10}P'_{12}+K_3+K_{53}+P'_7+P'_{18}+P'_{23}),\\
&P'_8P'_{10}P'_{19}+K_8P'_{19}+P'_3P'_8+P'_6P'_{19}+P'_{10}P'_{12}\\
+&P'_{16}P'_{19}+K_3+K_{53}+P'_7+P'_{18}+P'_{19}+P'_{23},\\
&P'_{19}P'_{22}+K_1+K_{11}+P'_6+P'_{11}+P'_{17}+P'_{21}+P'_{26},\\
&P'_{23}P'_{26}+K_{65}+P'_{21}+P'_{25}+P'_{30},\\
&P'_1+1,P'_2,P'_5+1,P'_9+1
\end{aligned}
$$

Fig. 1. Polynomials for the first two rounds of KTANTAN32.

$$
\begin{aligned}
&(C'_{57+j}+C''_{57+j})(C'_{53+j}+C''_{53+j}+1)(C'_{17+j}+C''_{17+j}),\\
&(C'_{57+j}+C''_{57+j})(C'_{53+j}+C''_{53+j}+1)(C'_{33+j}+C''_{33+j}),\\
&(C'_{57+j}+C''_{57+j}+1)(C'_{25+j}+C''_{25+j}),\\
&(C'_{57+j}+C''_{57+j}+1)(C'_{41+j}+C''_{41+j}),\\
&(C'_{53+j}+C''_{53+j}+1)(C'_{21+j}+C''_{21+j}),\\
&(C'_{53+j}+C''_{53+j}+1)(C'_{37+j}+C''_{37+j}),\\
&(C'_{53+j}+C''_{53+j}+1)(C'_{49+j}+C'_{57+j}+C''_{49+j}+C''_{57+j}+1),\\
&(C'_{49+j}+C''_{49+j}+1)(C'_{17+j}+C''_{17+j}),\\
&(C'_{49+j}+C''_{49+j}+1)(C'_{33+j}+C''_{33+j}),\\
&C'_{1+j}+C'_{33+j}+C'_{49+j}+C''_{1+j}+C''_{33+j}+C''_{49+j},\\
&C'_{5+j}+C'_{37+j}+C'_{53+j}+C''_{5+j}+C''_{37+j}+C''_{53+j},\\
&C'_{9+j}+C'_{41+j}+C'_{57+j}+C''_{9+j}+C''_{41+j}+C''_{57+j},
\end{aligned}
$$

Fig. 2. 2R polynomials for PRESENT with $j \in \{0,2\}$.

$$(C'_{36}+C''_{36})((C'_4+C''_4)(C'_{20}+C'_{52}+C''_{20}+C''_{52}+1)+(C'_{20}+C''_{20}+1)(C'_{52}+C''_{52}+1)),$$
$$(C'_{37}+C''_{37})((C'_5+C''_5)(C'_{21}+C'_{53}+C''_{21}+C''_{53}+1)+(C'_{21}+C''_{21}+1)(C'_{53}+C''_{53}+1)),$$
$$(C'_{40}+C''_{40})((C'_8+C''_8)(C'_{24}+C'_{56}+C''_{24}+C''_{56}+1)+(C'_{24}+C''_{24}+1)(C'_{56}+C''_{56}+1)),$$
$$(C'_{41}+C''_{41})((C'_9+C''_9)(C'_{25}+C'_{57}+C''_{25}+C''_{57}+1)+(C'_{25}+C''_{25}+1)(C'_{57}+C''_{57}+1)),$$
$$(C'_{45}+C''_{45})((C'_{13}+C''_{13})(C'_{29}+C'_{61}+C''_{29}+C''_{61}+1)+(C'_{29}+C''_{29}+1)(C'_{61}+C''_{61}+1)),$$
$$(C'_{46}+C''_{46})((C'_{14}+C''_{14})(C'_{30}+C'_{62}+C''_{30}+C''_{62}+1)+(C'_{30}+C''_{30}+1)(C'_{62}+C''_{62}+1)),$$
$$(C'_{06}+C''_{06})((C'_{22}+C''_{22})(C'_{38}+C'_{54}+C''_{38}+C''_{54}+1)+(C'_{38}+C''_{38}+1)(C'_{54}+C''_{54}+1)),$$
$$(C'_{10}+C''_{10})((C'_{26}+C''_{26})(C'_{42}+C'_{58}+C''_{42}+C''_{58}+1)+(C'_{42}+C''_{42}+1)(C'_{58}+C''_{58}+1)),$$
$$(C'_{12}+C''_{12})((C'_{28}+C''_{28})(C'_{44}+C'_{60}+C''_{44}+C''_{60}+1)+(C'_{44}+C''_{44}+1)(C'_{60}+C''_{60}+1)),$$
$$(C'_{52}+C''_{52}+1)(C'_{20}+C''_{20}+1)(C'_4+C'_{36}+C''_4+C''_{36}),$$
$$(C'_{60}+C''_{60}+1)(C'_{28}+C''_{28}+1)(C'_{12}+C'_{44}+C''_{12}+C''_{44}),$$
$$(C'_{10}+C'_{42}+C'_{58}+C''_{10}+C''_{42}+C''_{58})(C'_2+C'_{34}+C'_{50}+C''_2+C''_{34}+C''_{50}).$$

Fig. 3. 3R polynomials for PRESENT.

$$(K_1+P'_1+1)(K_0+K_3+K_{29}+P'_0+P'_3),$$
$$(K_2+P'_2)(K_0+K_3+K_{29}+P'_0+P'_3),$$
$$K_1K_2+K_1P'_2+K_2P'_1+P'_1P'_2+K_0+K_1+K_3+K_{29}+P'_0+P'_1+P'_3,$$
$$(K_9+P'_9+1)(K_8+K_{11}+K_{31}+P'_8+P'_{11}),$$
$$(K_{10}+P'_{10})(K_8+K_{11}+K_{31}+P'_8+P'_{11}),$$
$$K_9K_{10}+K_9P'_{10}+K_{10}P'_9+P'_9P'_{10}+K_8+K_9+K_{11}+K_{31}+P'_8+P'_9+P'_{11},$$
$$(K_{49}+P'_{49}+1)(K_{41}+K_{48}+K_{51}+P'_{48}+P'_{51}),$$
$$(K_{50}+P'_{50})(K_{41}+K_{48}+K_{51}+P'_{48}+P'_{51}),$$
$$K_{49}K_{50}+K_{49}P'_{50}+K_{50}P'_{49}+P'_{49}P'_{50}+K_{41}+K_{48}+K_{49}+K_{51}+P'_{48}+P'_{49}+P'_{51},$$
$$(K_{57}+P'_{57}+1)(K_{43}+K_{56}+K_{59}+P'_{56}+P'_{59}),$$
$$(K_{58}+P'_{58})(K_{43}+K_{56}+K_{59}+P'_{56}+P'_{59}),$$
$$K_{57}K_{58}+K_{57}P'_{58}+K_{58}P'_{57}+P'_{57}P'_{58}+K_{43}+K_{56}+K_{57}+K_{59}+P'_{56}+P'_{57}+P'_{59},$$
$$K_5+K_7+P'_5+P'_7,$$
$$K_6+K_7+P'_6+P'_7,$$
$$K_{53}+K_{55}+P'_{53}+P'_{55},$$
$$K_{54}+K_{55}+P'_{54}+P'_{55}$$

Fig. 4. Polynomials for the first two rounds of PRESENT.

SYMAES: A Fully Symbolic Polynomial System Generator for AES-128*

Vesselin Velichkov[1,2,**], Vincent Rijmen[1,2,3], and Bart Preneel[1,2]

[1] Department of Electrical Engineering ESAT/SCD-COSIC,
Katholieke Universiteit Leuven. Kasteelpark Arenberg 10, B-3001 Heverlee, Belgium.
vesselin.velichkov@esat.kuleuven.be
[2] Interdisciplinary Institute for BroadBand Technology (IBBT), Belgium.
[3] Graz University of Technology.

SYMAES is a software tool that generates a system of polynomials in GF(2), corresponding to the round transformation and key schedule of the block cipher AES-128 [1].

Most of the existing polynomial system generators for AES are typically used under the assumption that the plaintext and ciphertext bits are known, and therefore are treated as constants. Although some of the generators, such as the AES (SR) Polynomial System Generator [2,3], can also be used when this assumption is not made, the instructions to do this are not always very natural. SYMAES is specifically designed to address the case in which (some of) the plaintext and ciphertext bits are unknown and are therefore treated as symbolic variables. Such a scenario is realistic and arises during the algebraic cryptanalysis of AES-based constructions, where only parts of the plaintext and/or ciphertext are known. An example of such a construction is the stream cipher LEX [4], a small-scale version of which has been analysed using SYMAES [5].

The inputs to SYMAES are the bits of the plaintext and the bits of the original key, represented as symbolic variables in GF(2). The output is a system of equations describing the output bits of one round of AES as a function of the input bits and the key. SYMAES also generates symbolic equations for the AES key schedule. Then, the bits of the round keys are expressed as polynomials in the bits of the original key.

As a final note we would like to stress that SYMAES should not be seen as a competitor to existing AES polynomial system generators, but rather as an addition to them. SYMAES achieves in a more natural way what can also be achieved using SR [3]. Similarly to SR, SYMAES is also written in Python and is used within the open source computer algebra Sage [6]. This makes possible a future integration of the SYMAES code into SR.

This submission is accompanied by an appendix containing the SYMAES source code and usage instructions.

References

1. Joan Daemen, Vincent Rijmen, The Design of Rijndael: AES - The Advanced Encryption Standard, Springer-Verlag, 2002.
2. Carlos Cid, Sean Murphy, Matthew J. B. Robshaw: Small Scale Variants of the AES. FSE 2005: 145-162.

* This work was supported in part by the IAP Program P6/26 BCRYPT of the Belgian State (Belgian Science Policy), and in part by the European Commission through the ICT program under contract ICT-2007-216676 ECRYPT II.

** DBOF Doctoral Fellow, K.U.Leuven, Belgium.

3. Martin Albrecht, Small Scale Variants of the AES (SR) Polynomial System Generator, available at `http://www.sagemath.org/doc/reference/sage/crypto/mq/sr.htm`.
4. Alex Biryukov, The Design of a Stream Cipher Lex. Selected Areas in Cryptography 2006:67-75.
5. V. Velichkov, V. Rijmen, and B. Preneel, Algebraic Cryptanalysis of a Small-Scale Version of Stream Cipher LEX, IET Information Security Journal, 16 pages, 2009, *to appear*.
6. Stein, William, *Sage: Open Source Mathematical Software (Version 3.1.4)*, The Sage Group, 2008, `http://www.sagemath.org`.

Efficient Decomposition of Dense Matrices over GF(2)

Martin R. Albrecht*[1] and Clément Pernet[2]

[1] Information Security Group, Royal Holloway, University of London
Egham, Surrey TW20 0EX, United Kingdom
M.R.Albrecht@rhul.ac.uk
[2] INRIA-MOAIS LIG, Grenoble Univ. ENSIMAG, Antenne de Montbonnot 51,
avenue Jean Kuntzmann, F-38330 MONTBONNOT SAINT MARTIN, France
clement.pernet@imag.fr

Abstract. In this work we describe an efficient implementation of a hierarchy of algorithms for the decomposition of dense matrices over the field with two elements ($\mathbb{F}_2$). Matrix decomposition is an essential building block for solving dense systems of linear and non-linear equations and thus much research has been devoted to improve the asymptotic complexity of such algorithms. In this work we discuss an implementation of both well-known and improved algorithms in the M4RI library. The focus of our discussion is on a new variant of the M4RI algorithm – denoted MMPF in this work – which allows for considerable performance gains in practice when compared to the previously fastest implementation. We provide performance figures on x86_64 CPUs to demonstrate the viability of our approach.

1 Introduction

We describe an efficient implementation of a hierarchy of algorithms for PLS decomposition of dense matrices over the field with two elements ($\mathbb{F}_2$). The PLS decomposition is closely related to the well-known PLUQ and LQUP decompositions. However, it offers some advantages in the particular case of $\mathbb{F}_2$. Matrix decomposition is an essential building block for solving dense systems of linear and non-linear equations (cf. [11, 10]) and thus much research has been devoted to improve the asymptotic complexity of such algorithms. In particular, it has been shown that various matrix decompositions such as PLUQ, LQUP and LPS are essentially equivalent and can be reduced to matrix-matrix multiplication (cf. [13]). Thus, we know that these decompositions can be achieved in $\mathcal{O}(n^\omega)$ where ω is the exponent of linear algebra[3]. In this work we focus on matrix decomposition in the special case of $\mathbb{F}_2$ and discuss an implementation of both well-known and improved algorithms in the M4RI library [2]. The M4RI library implements dense linear algebra over $\mathbb{F}_2$ and is used by the Sage [16] mathematics software and the PolyBoRi [9] package for computing Gröbner bases. It is also the linear algebra library used in [15, 14].

Our implementation focuses on 64-bit x86 architectures (x86_64), specifically the Intel Core 2 and the AMD Opteron. Thus, we assume in this chapter that each native CPU word has 64 bits. However it should be noted that our code also runs on 32-bit CPUs and on non-x86 CPUs such as the PowerPC.

* This author was supported by the Royal Holloway Valerie Myerscough Scholarship.

[3] For practical purposes we set $\omega = 2.807$.

Element-wise operations over $\mathbb{F}_2$ are relatively cheap compared to loads from and writes to memory. In fact, in this work we demonstrate that the two fastest implementations for dense matrix decomposition over $\mathbb{F}_2$ (the one presented in this work and the one found in Magma [8] due to Allan Steel) perform worse for sparse matrices despite the fact that fewer field operations are performed. This indicates that counting raw field operations is not an adequate model for estimating the running time in the case of $\mathbb{F}_2$.

This work is organised as follows. We will start by giving the definitions of reduced row echelon forms (RREF), PLUQ and PLS decomposition in Section 2 and establish their relations. We will then discuss Gaussian elimination and the M4RI algorithm in Section 3 followed by a discussion of cubic PLS decomposition and the MMPF algorithm in 4. We will then discuss asymptotically fast PLS decomposition in Section 5 and implementation issues in Section 6. We conclude by giving empirical evidence of the viability of our approach in Section 7.

2 RREF and PLS

Proposition 1 (PLUQ decomposition). *Any $m \times n$ matrix A with rank r, can be written $A = PLUQ$ where P and Q are two permutation matrices, of dimension respectively $m \times m$ and $n \times n$, L is $m \times r$ unit lower triangular and U is $r \times n$ upper triangular.*

Proof. See [13].

Proposition 2 (PLS decomposition). *Any $m \times n$ matrix A with rank r, can be written $A = PLS$ where P is a permutation matrix of dimension $m \times m$, L is $m \times r$ unit lower triangular and S is an $r \times n$ matrix which is upper triangular except that its columns are permuted, that is $S = UQ$ for U $r \times n$ upper triangular and Q is a $n \times n$ permutation matrix.*

Proof. Write $A = PLUQ$ and set $S = UQ$.

Another way of looking at PLS decomposition is to consider the $A = LQUP$ decomposition [12]. We have $A = LQUP = LSP$ where $S = QU$. We can also write $A = LQUP = SUP$ where $S = LQ$. Applied to A^T we then get $A = P^T U^T S^T = P' L' S'$. Finally, a proof for Proposition 2 can also be obtained by studying any one of the Algorithms 9, 3 or 4.

Definition 1 (Row Echelon Form). *An $m \times n$ matrix A is in echelon form if all zero rows are grouped together at the last row positions of the matrix, and if the leading coefficient of each non zero row is one and is located to the right of the leading coefficient of the above row.*

Proposition 3. *Any $m \times n$ matrix can be transformed into echelon form by matrix multiplication.*

Proof. See [13]

Note that while there are many PLUQ decompositions of any matrix A there is always also a decomposition for which we have that $S = UQ^T$ is a row echelon form of A. In

this work we compute $A = PLS$ such that S is in row echelon form. Thus, a proof for Proposition 3 can also be obtained by studying any one of the Algorithms 9, 3 or 4.

Definition 2 (Reduced Row Echelon Form). *An $m \times n$ matrix A is in reduced echelon form if it is in echelon form and each leading coefficient of a non zero row is the only non zero element in its column.*

3 Gaussian Elimination and M4RI

Gaussian elimination is the classical, cubic algorithm for transforming a matrix into (reduced) row echelon form using elementary row operations only. The "Method of the Four Russians" Inversion (M4RI) [7] reduces the number of additions required by Gaussian elimination by a factor of $\log n$ by using a caching technique inspired by Kronrod's method for matrix-matrix multiplication.

3.1 The "Method of the Four Russians" Inversion (M4RI)

The "Method of the Four Russians" inversion was introduced in [5] and later described in [6] and [7]. It inherits its name and main idea from the misnamed "Method of the Four Russians" multiplication [4, 1].

To give the main idea consider for example the matrix A of dimension $m \times n$ in Figure 3.1. The $k \times n$ ($k = 3$) submatrix on the top has full rank and we performed Gaussian elimination on it. Now, we need to clear the first k columns of A for the rows below k (and above the submatrix in general if we want the reduced row echelon form). There are 2^k possible linear combinations of the first k rows, which we store in a table T. We index T by the first k bits (e.g., $011 \to 3$). Now to clear k columns of row i we use the first k bits of that row as an index in T and add the matching row of T to row i, causing a cancellation of k entries. Instead of up to k additions this only costs one addition due to the pre-computation. Using Gray codes (or similar techniques) this pre-computation can be performed in 2^k vector additions and the overall cost is $2^k + m - k + k^2$ vector additions in the worst case (where k^2 accounts for the Gauss elimination of the $k \times n$ submatrix). The naive approach would cost $k \cdot m$ row additions in the worst case to clear k columns. If we set $k = \log m$ then the complexity of clearing k columns is $\mathcal{O}\left(m + \log^2 m\right)$ vector additions in contrast to $\mathcal{O}(m \cdot \log m)$ vector additions using the naive approach.

This idea leads to Algorithm 1. In this algorithm the subroutine GaussSubmatrix (cf. Algorithm 8) performs Gauss elimination on a $k \times n$ submatrix of A starting at position (r, c) and searches for pivot rows up to m. If it cannot find a submatrix of rank k it will terminate and return the rank $\overline{k}$ found so far. Note the technicality that the routine GaussSubmatrix and its interaction with Algorithm 1 make use of the fact that all the entries in a column below a pivot are zero if they were considered already.

The subroutine MakeTable (cf. Algorithm 7) constructs the table T of all 2^k linear combinations of the k rows starting a row r and a column c, i.e. it enumerates all elements of the vector space $\mathsf{span}(r, ..., r + \overline{k} + 1)$ spanned by the rows $r, \ldots, r + \overline{k} - 1$. Finally, the subroutine AddRowsFromTable (cf. Algorithm 6) adds the appropriate row from T – indexed by k bits starting at column c – to each row of A with index $i \notin \{r, \ldots, r + \overline{k} - 1\}$. That is, it adds the appropriate linear combination of the rows $\{r, \ldots, r + \overline{k} - 1\}$ onto a row i in order to clear k columns.

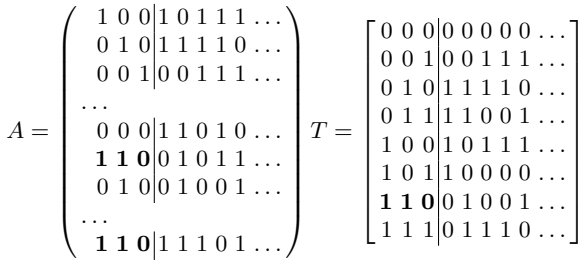


Fig. 1. M4RI Idea

Note that the relation between the index id and the row j in T is static and known *a priori* because GaussSubmatrix puts the submatrix in reduced row echelon form. In particular this means that the $\overline{k} \times \overline{k}$ submatrix starting at (r, c) is the identity matrix.

Input: A – a $m \times n$ matrix
Input: k – an integer $k > 0$
Result: A is in reduced row echelon form.
begin
  $r, c \longleftarrow 0, 0$;
  while $c < n$ **do**
    if $c + k > n$ **then** $k \leftarrow n - c$;
    $\overline{k} \longleftarrow$ GaussSubmatrix(A, r, c, k, m);
    if $\overline{k} > 0$ **then**
      $T, L \longleftarrow$ MakeTable$(A, r, c, \overline{k})$;
      AddRowsFromTable$(A, 0, r, c, \overline{k}, T, L)$;
      AddRowsFromTable$(A, r + \overline{k}, m, c, \overline{k}, T, L)$;
    end
    $r, c \longleftarrow r + \overline{k}, c + \overline{k}$;
    if $k \neq \overline{k}$ **then** $c \leftarrow c + 1$;
  end
end

Algorithm 1: M4RI

When studying the performance of Algorithm 1, we expect the function MakeTable to contribute most. Instead of performing $\overline{k}/2 \cdot 2^{\overline{k}} - 1$ additions MakeTable only performs $2^{\overline{k}} - 1$ vector additions. However, in practice the fact that $\overline{k}$ columns are processed in each loop iteration of AddRowsFromTable contributes signficiantly due to the better cache locality. Assume the input matrix A does not fit into L2 cache. Gaussian elimination would load a row from memory, clear one column and likely evict that row from cache in order to make room for the next few rows before considering it again for the next column. In the M4RI algorithm more columns are cleared per load.

We note that our presentation of M4RI differs somewhat from that in [6]. The key difference is that our variant does not throw an error if it cannot find a pivot within

the first $3k$ rows in GaussSubmatrix. Instead, our variant searches all rows and consequently the worst-case complexity is cubic. However, on average for random matrices we expect to find a pivot within $3k$ rows and thus expect the average-case complexity to be $\mathcal{O}(n^3/\log n)$.

4 M4RI and PLS Decomposition

In order to recover the PLS decomposition of some matrix A, we can adapt Gaussian elimination to preserve the transformation matrix in the lower triangular part of the input matrix A and to record all permutations performed. This leads to Algorithm 9 in the Appendix which modifies A such that it contains L in below the main diagonal, S above the main diagonal and returns P and Q such that $PLS = A$ and $SQ^T = U$.

The main differences between Gaussian elimination and Algorithm 9 are:

- No elimination is performed above the currently considered row, i.e. the rows $0, \ldots, r-1$ are left unchanged. Instead elimination starts below the pivot, from row $r+1$.
- Column swaps are performed at the end of Algorithm 9 but not in Gaussian elimination. This step compresses L such that it is lower triangular.
- Row additions are performed starting at column $r+1$ instead of r to preserve the transformation matrix L. Over any other field we would have to rescale $A[r, r]$ for the transformation matrix L but over $\mathbb{F}_2$ this is not necessary.

4.1 The Method of Many People Factorisation (MMPF)

In order to use the M4RI improvement over Gaussian elimination for PLS decomposition, we have to adapt the M4RI algorithm.

Column Swaps Since column swaps only happen at the very end of the algorithm we can modify the M4RI algorithm in the obvious way to introduce them.

U vs. I Recall, that the function GaussSubmatrix generates small $\overline{k} \times \overline{k}$ identity matrices. Thus, even if we remove the call to the function AddRowsFromTable$(A, 0, r, c, \overline{k}, T)$ from Algorithm 1 we would still eliminate up to $\overline{k} - 1$ rows above a given pivot and thus would fail to produce U. The reason the original specification [5] of the M4RI requires $\overline{k} \times \overline{k}$ identity matrices is to have a *a priori* knowledge of the relationship between id and j in the function AddRowsFromTable. On the other hand the rows of any $\overline{k} \times n$ upper triangular matrix also form a basis for the $\overline{k}$-dimensional vector space $\mathsf{span}(r, \ldots, r+\overline{k}-1)$. Thus, we can adapt GaussSubmatrix to compute the upper triangular matrix instead of the identity. Then, in MakeTable1 we can encode the actual relationship between a row j of T and id in the lookup table L.

Preserving L In Algorithm 9 preserving the transformation matrix L is straight forward: addition starts in column $c+1$ instead of c. On the other hand, for M4RI we need to fix the table T to update the transformation matrix correctly; For example, assume $\overline{k} = 3$ and that the first row of the $\overline{k} \times n$ submatrix generated by GaussSubmatrix has the first $\overline{k}$ bits equal to `[1 0 1]`. Assume further that we want to clear $\overline{k}$ bits of a a row which also starts with `[1 0 1]`. Then – in order to generate L – we need to encode that

this row is cleared by adding the first row only, i.e. we want the first $\overline{k} = 3$ bits to be `[1 0 0]`. Recall that in the M4RI algorithm the *id* for the row j starting with `[1 0 0]` is `[1 0 0]` if expressed as a sequence of bits. Thus, to correct the table, we add the $\overline{k}$ bits of the *a priori id* onto the first $\overline{k}$ entries in T (starting at c) as in MakeTable1.

Other Bookkeeping Recall that GaussSubmatrix's interaction with Algorithm 1 uses the fact that processed columns of a row are zeroed out to encode whether a row is "done" or not. This is not true anymore if we compute the PLS decomposition instead of the upper triangular matrix in GaussSubmatrix since we store L below the main diagonal. Thus, we explicitly encode up to which row a given column is "done" in PlsSubmatrix (cf. Algorithm 10). Finally, we have to take care not to include the transformation matrix L when constructing T.

Input: A – a $m \times n$ matrix
Input: r_{start} – an integer $0 \leq r_{\text{start}} < m$
Input: c_{start} – an integer $0 \leq c_{\text{start}} < n$
Input: k – an integer $k > 0$
Result: Retuns an $2^k \times n$ matrix T and the translation table L
begin
  $T \longleftarrow$ the $2^k \times n$ zero matrix;
  for $1 \leq i < 2^k$ **do**
    $j \longleftarrow$ the row index of A to add according to the Gray code;
    add row j of A to the row i of T starting at c_{start};
  end
  $L \longleftarrow$ an integer array with 2^k entries;
  for $1 \leq i < 2^k$ **do**
    $id = \sum_{j=0}^{k} T[i, c_{\text{start}} + j] \cdot 2^{k-j-1}$;
    $L[id] \longleftarrow i$;
  end
  for $1 \leq i < 2^k$ **do**
    $b_0, \ldots, b_{\overline{k}-1} \longleftarrow$ bits of *a priori id* of the row i;
    for $0 \leq j < \overline{k}$ **do**
      $T[i, c_{\text{start}} + j] \longleftarrow T[i, c_{\text{start}} + j] + b_j$;
    end
  end
  return T, L;
end

Algorithm 2: MakeTable1

These modifications lead to Algorithm 3 which computes the *PLS* decomposition of A in-place, that is L is stored below the main diagonal and S is stored above the main diagonal of the input matrix. Since none of the changes to the M4RI algorithm affect the asymptotical complexity, Algorithm 3 is cubic in the worst case and has complexity $\mathcal{O}\left(n^3 / \log n\right)$ in the average case.

Input: A – a $m \times n$ matrix
Input: P – a permutation vector of length m
Input: Q – a permutation vector of length n
Input: k – an integer $k > 0$
Result: PLS decomposition of A
begin
  $r, c \longleftarrow 0, 0$;
  for $0 \leq i < n$ **do** $Q[i] \longleftarrow i$;
  for $0 \leq i < m$ **do** $P[i] \longleftarrow i$;
  while $r < m$ **and** $c < n$ **do**
    if $c + k > n$ **then** $k \longleftarrow n - c$;
    $\overline{k}, d_r \longleftarrow$ PlsSubmatrix(A, r, c, k, P, Q);
    $U \longleftarrow$ the $\overline{k} \times n$ submatrix starting at $(r, 0)$ where every entry prior to the upper triangular matrix starting at (r, c) is zeroed out;
    if $\overline{k} > 0$ **then**
      $T, L \longleftarrow$ MakeTable1$(U, 0, c, \overline{k})$;
      AddRowsFromTable$(A, d_r + 1, m, c, \overline{k}, T, L)$;
      $r, c \leftarrow r + \overline{k}, c + \overline{k}$;
    else
      `// skip zero column`
      $c \leftarrow c + 1$;
    end
  end
  `// Now compress L`
  for $0 \leq j < r$ **do** swap the columns j and $Q[j]$ starting at row j;
  return r;
end

Algorithm 3: MMPF

5 Asymptotically Fast PLS Decomposition

It is well-known that PLUQ decomposition can be accomplished in-place and in time complexity $\mathcal{O}(n^\omega)$ by reducing it to matrix-matrix multiplication (cf. [13]). We give a slight variation of the recursive algorithm from [13] in Algorithm 4. We compute the PLS instead of the PLUQ decomposition.

In Algorithm 4 the routine SubMatrix(r_s, c_s, r_e, c_e) returns a "view" (cf. [3]) into the matrix A starting at row and column r_s and c_s resp. and ending at row and column r_e and c_e resp. We note that that the step $A_{NE} \longleftarrow L_{NW}^{-1} \times A_{NE}$ can be reduced to matrix-matrix multiplication (cf. [13]). Thus Algorithm 4 can be reduced to matrix-matrix multiplication and has complexity $\mathcal{O}(n^\omega)$. Since no temporary matrices are needed to perform the algorithm, except maybe in the matrix-matrix multiplication step, the algorithm is in-place.

6 Implementation

Similarly to matrix multiplication (cf. [3]) it is beneficial to call Algorithm 4 until some "cutoff" bound and to switch to a base-case implementation (in our case Algorithm 3) once this bound is reached. We perform the switch over if the matrix fits into 4MB or

Input: A – a $m \times n$ matrix
Input: P – a permutation vector of length m
Input: Q – a permutation vector of length n
Result: PLS decomposition of A
begin
    $n_0 \longleftarrow$ pick some integer $0 \leq n_0 < n$; `//` $n_0 \approx n/2$
    $A_0 \longleftarrow$ SubMatrix$(A, 0, 0, m, n_0)$;
    $A_1 \longleftarrow$ SubMatrix$(A, 0, n_0, m, n)$;
    $Q_0 \longleftarrow Q[0, \ldots, n_0]$;
    $r_0 \longleftarrow$ PLS(A_0, P, Q_0); `// first recursive call`
    for $0 \leq i \leq n_0$ **do** $Q[i] \leftarrow Q_0[i]$;
    $A_{NW} \longleftarrow$ SubMatrix$(A, 0, 0, r_0, r_0)$;
    $A_{SW} \longleftarrow$ SubMatrix$(A, r_0, 0, m, r_0)$;
    $A_{NE} \longleftarrow$ SubMatrix$(A, 0, n_0, r_0, n)$;
    $A_{SE} \longleftarrow$ SubMatrix(A, r_0, n_0, m, n);
    if r_1 **then**
        `// Compute of the Schur complement`
        $A_1 \longleftarrow P \times A_1$;
        $L_{NW} \longleftarrow$ the lower left triangular matrix in A_{NW};
        $A_{NE} \longleftarrow L_{NW}^{-1} \times A_{NE}$;
        $A_{SE} \longleftarrow A_{SE} + A_{SW} \times A_{NE}$;
    end
    $P_1 \longleftarrow P[r_0, \ldots, m]$;
    $Q_1 \longleftarrow Q[n_0, \ldots, n]$;
    $r_1 \longleftarrow$ PLS(A_{SE}, P_1, Q_1); `// second recursive call`
    $A_{SW} \longleftarrow P \times A_{SW}$;
    `// Update P & Q`
    for $0 \leq i < m - r_0$ **do** $P[r_0 + 1] = P_1[i] + r_0$;
    for $0 \leq i < n - n_0$ **do** $Q[n_0 + i] \leftarrow Q_1[i] + n_0$;
    $j \leftarrow r_0$;
    for $n_0 \leq i < n_0 + r_1$ **do** $Q[j] \leftarrow Q[i]$; $j \leftarrow j + 1$;
    `// Now compress L`
    $j \leftarrow n_0$;
    for $r_0 \leq i < r_0 + r_1$ **do** swap the columns i and j starting at row i;
    return $r_0 + r_1$;
end

Algorithm 4: PLS Decomposition

in L2 cache, whichever is smaller. These values seem to provide the best performance on our target platforms.

The reason we are considering the PLS decomposition instead of either the LQUP or the PLUQ decomposition is that the PLS decomposition has several advantages over $\mathbb{F}_2$, in particular when the flat row-major representation is used to store entries.

- We may choose where to cut with respect to columns in Algorithm 4. In particular, we may choose to cut along word boundaries. For LQUP decomposition, where roughly all steps are transposed, column cuts are determined by the rank r_0.
- In Algorithm 3 rows are added instead of columns. Row operations are much cheaper than column operations in row-major representation.
- Column swaps do not occur in the main loop of either Algorithm 4 or 3, but only row swaps are performed. Column swaps are only performed at the end. Column swaps are much more expensive than row swaps (see below).
- Fewer column swaps are performed for PLS decomposition than for PLUQ decomposition since U is not compressed.

One of the major bottleneck are column swaps. In Algorithm 5 a simple algorithm for swapping two columns a and b is given with bit-level detail. In Algorithm 5 we assume that the bit position of a is greater than the bit position of b for simplicity of presentation. The advantage of the strategy in Algorithm 5 is that it uses no conditional jumps in the inner loop, However, it still requires 9 instructions per row. On the other hand, we can add two rows with $9 \cdot 128 = 1152$ entries in 9 instructions if the SSE2 instruction set is available. Thus, for matrices of size 1152×1152 it takes roughly the same number of instructions to add two matrices as it does to swap two columns. If we were to swap every column with some other column once during some algorithm it thus would be as expensive as a matrix multiplication for matrices of these dimensions.

Input: A – a $m \times n$ matrix
Input: a – an integer $0 \leq a < b < n$
Input: b – an integer $0 \leq a < b < n$
Result: Swaps the columns a and b in A
begin
  $M \longleftarrow$ the memory where A is stored;
  $a_w, b_w \longleftarrow$ the word index of a and b in M;
  $a_b, b_b \longleftarrow$ the bit index of a and b in a_w and b_w;
  $\Delta \longleftarrow a_b - b_b$;
  $a_m \longleftarrow$ the bit-mask where only the a_bth bit is set to 1;
  $b_m \longleftarrow$ the bit-mask where only the b_bth bit is set to 1;
  for $0 \leq i < m$ **do**
    $R \longleftarrow$ the memory where the row i is stored;
    $R[a_w] \longleftarrow R[a_w] \oplus ((R[b_w] \odot b_m) >> \Delta)$;
    $R[b_w] \longleftarrow R[b_w] \oplus ((R[a_w] \odot a_m) << \Delta)$;
    $R[a_w] \longleftarrow R[a_w] \oplus ((R[b_w] \odot b_m) >> \Delta)$;
  end
end

Algorithm 5: Column Swap

Another bottleneck for relatively sparse matrices in dense row-major representation is the search for pivots. Searching for a non-zero element in a row can be relatively expensive due to the need to identify the bit position. However, the main performance penalty is due to the fact that searching for a non-zero entry in one column is in a row-major representation is very cache unfriendly.

Indeed, both our implementation and the implementation available in Magma suffer from performance regression on relatively sparse matrices as shown in Figure 2. We stress that this is despite the fact that the theoretical complexity of matrix decomposition is rank sensitive, that is, strictly less field operations have to be performed for low rank matrices. While the penalty for relatively sparse matrices is much smaller for our implementation than for Magma, it clearly does not achieve the theoretical possible performance. Thus, we also consider a hybrid algorithm which starts with M4RI and switches to PLS-based elimination as soon as the (approximated) density reaches 15%, denoted as 'M+P 0.15'.

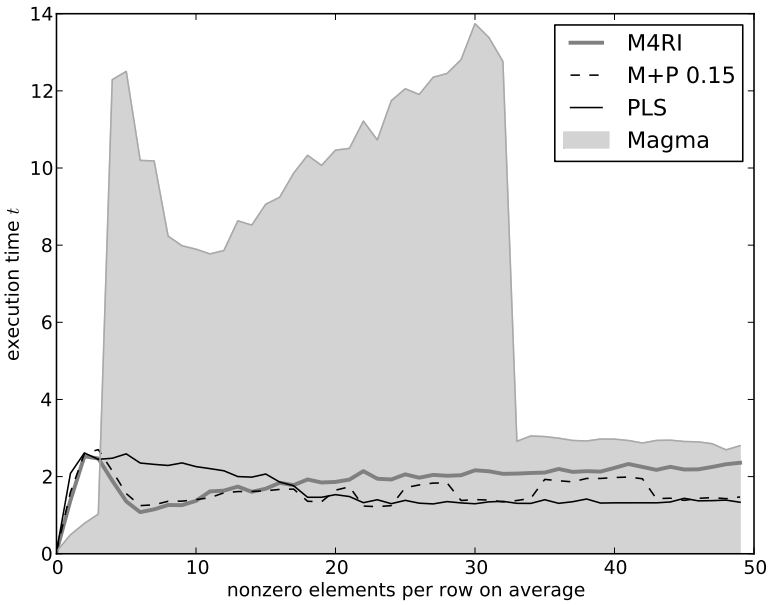


Fig. 2. Sensitivity to density for $n = 10^4$ on 2.6Ghz Opteron

7 Results

In Table 1 we give average running time over ten trials for computing reduced row echelon forms of dense random $n \times n$ matrices over $\mathbb{F}_2$. We compare the asymptotically fast implementation due to Allan Steel in Magma, the cubic Gaussian elimination implemented

by Victor Shoup in NTL, and both our implementations. Both the implementation in Magma and our PLS decomposition reduce matrix decomposition to matrix multiplication. A discussion and comparison of matrix multiplication in the M4RI library and in Magma can be found in [3]. In Table 1 the column 'PLS' denotes the complete running time for first computing the PLS decomposition and the computation of the reduced row echelon form from PLS.

	64-bit Linux, 2.6Ghz Opteron				64-bit Linux, 2.33Ghz Xeon (E5345)			
n	Magma 2.15-10	NTL 5.4.2	M4RI 20090105	PLS 20100324	Magma 2.16-7	NTL 5.4.2	M4RI 20100324	PLS 20100324
$10,000$	3.351s	18.45s	2.430s	1.452s	2.660s	12.05s	1.360s	0.864s
$16,384$	11.289s	72.89s	10.822s	6.920s	8.617s	54.79s	5.734s	3.388s
$20,000$	16.734s	130.46s	19.978s	10.809s	12.527s	100.01s	10.610s	5.661s
$32,000$	57.567s	479.07s	83.575s	49.487s	41.770s	382.52s	43.042s	20.967s
$64,000$	373.906s	2747.41s	537.900s	273.120s	250.193s	–	382.263s	151.314s

Table 1. RREF for random matrices

In Table 2 we give running times for matrices as they appear when solving non-linear systems of equations. The matrices HFE 25, 30 and 35 were contributed by Michael Brickenstein and appear during a Gröbner basis computation of HFE systems using PolyBoRi. The Matrix MXL was contributed by Wael Said and appears during an execution of the MXL2 algorithm [15] for a random quadratic system of equations. We consider these matrices within the scope of this work since during matrix elimination the density quickly increases and because even the input matrices are dense enough such that we expect one non-zero element per 128-bit wide SSE2 XOR on average. The columns 'M+P 0.xx' denote the hybrid algorithms which start with M4RI and switch over to PLS based echelon form computation once the density of the remaining part of the matrix reaches 15% or 20% respectively. We note that the relative performance of the M4RI and the PLS algorithm for these instances depends on particular machine configuration. To demonstrate this we give a set of timings for the Intel Xeon X7460 machine `sage.math`[4] in Table 2. Here, PLS always is faster than M4RI, while on a Xeon E5345 M4RI wins for all HFE examples. We note that Magma is not available on the machine `sage.math`. The HFE examples show that the observed performance regression for sparse matrices does have an impact in practice and that the hybrid approach does look promising for these instances.

8 Acknowledgments

We would like to thank anonymous referees for helpful comments on how to improve our presentation.

[4] Purchased under National Science Foundation Grant No. DMS-0821725.

			64-bit Fedora Linux, 2.33Ghz Xeon (E5345)				
Problem	Matrix Dimension	Density	Magma 2.16-7	M4RI 20100324	PLS 20100324	M+P 0.15 20100429	M+P 0.20 20100429
HFE 25	$12,307 \times 13,508$	0.076	3.68s	1.94s	2.09s	2.33s	2.24s
HFE 30	$19,907 \times 29,323$	0.067	23.39s	11.46s	13.34s	12.60s	13.00s
HFE 35	$29,969 \times 55,800$	0.059	–	49.19s	68.85s	66.66s	54.42s
MXL	$26,075 \times 26,407$	0.185	55.15	12.25s	9.22s	9.22s	10.22s
			64-bit Ubuntu Linux, 2.66Ghz Xeon (X7460)				
Problem	Matrix Dimension	Density		M4RI 20100324	PLS 20100324	M+P 0.15 20100429	M+P 0.20 20100429
HFE 25	$12,307 \times 13,508$	0.076		2.24s	2.00s	2.39s	2.35s
HFE 30	$19,907 \times 29,323$	0.067		27.52s	13.29s	13.78s	22.9s
HFE 35	$29,969 \times 55,800$	0.059		115.35s	72.70s	84.04s	122.65s
MXL	$26,075 \times 26,407$	0.185		26.61s	8.73s	8.75s	13.23s
			64-bit Debian/GNU Linux, 2.6Ghz Opteron)				
Problem	Matrix Dimension	Density	Magma 2.15-10	M4RI 20100324	PLS 20100324	M+P 0.15 20100429	M+P 0.20 20100429
HFE 25	$12,307 \times 13,508$	0.076	4.57s	3.28s	3.45s	3.03s	3.21s
HFE 30	$19,907 \times 29,323$	0.067	33.21s	23.72s	25.42s	23.84s	25.09s
HFE 35	$29,969 \times 55,800$	0.059	278.58s	126.08s	159.72s	154.62s	119.44s
MXL	$26,075 \times 26,407$	0.185	76.81s	23.03s	19.04s	17.91s	18.00s

Table 2. RREF for matrices from practice.

References

1. A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.
2. Martin Albrecht and Gregory V. Bard. *The M4RI Library – Version 20091104.* The M4RI Team, 2009. http://m4ri.sagemath.org.
3. Martin Albrecht, Gregory V. Bard, and William Hart. Algorithm 898: Efficient multiplication of dense matrices over GF(2). *ACM Transactions on Mathematical Software*, 37(1), 2009. pre-print available at http://arxiv.org/abs/0811.1714.
4. V. Arlazarov, E. Dinic, M. Kronrod, and I. Faradzev. On economical construction of the transitive closure of a directed graph. *Dokl. Akad. Nauk.*, 194(11), 1970. (in Russian), English Translation in Soviet Math Dokl.
5. Gregory V. Bard. Accelerating cryptanalysis with the Method of Four Russians. Cryptology ePrint Archive, Report 2006/251, 2006. Available at http://eprint.iacr.org/2006/251.pdf.
6. Gregory V. Bard. *Algorithms for Solving Linear and Polynomial Systems of Equations over Finite Fields with Applications to Cryptanalysis.* PhD thesis, University of Maryland, 2007.
7. Gregory V. Bard. Matrix inversion (or LUP-factorization) via the Method of Four Russians, in $\theta(n^3/\log(n))$ time. In Submission, 2008.
8. Wieb Bosma, John Cannon, and Catherine Playoust. The MAGMA Algebra System I: The User Language. In *Journal of Symbolic Computation 24*, pages 235–265. Academic Press, 1997.
9. Michael Brickenstein and Alexander Dreyer. PolyBoRi: A framework for Gröbner basis computations with Boolean polynomials. In *Electronic Proceedings of MEGA 2007*, 2007. Available at http://www.ricam.oeaw.ac.at/mega2007/electronic/26.pdf.
10. Nicolas T. Courtois, Alexander Klimov, Jacques Patarin, and Adi Shamir. Efficient algorithms for solving overdefined systems of multivariate polynomial equations. In *Advances*

in Cryptology — EUROCRYPT 2000, volume 1807 of *Lecture Notes in Computer Science*, pages 392–407, Berlin, Heidelberg, New York, 2000. Springer Verlag.
11. Jean-Charles Faugère. A new efficient algorithm for computing Gröbner basis (F4). *Journal of Pure and Applied Algebra*, 139(1-3):61–88, 1999.
12. O.H. Ibarra, S. Moran, and R. Hui. A generalization of the fast LUP matrix decomposition algorithm and applications. *Journal of Algorithms*, 3:45–56, 1982.
13. Claude-Pierre Jeannerod, Clément Pernet, and Arne Storjohann. Fast gaussian elimination and the PLUQ decomposition. *in submission*, 2010.
14. Mohamed Saied Emam Mohamed, Daniel Cabarcas, Jintai Ding, Johannes Buchmann, and Stanislav Bulygin. Mxl3: An efficient algorithm for computing gröbner bases of zero-dimensional ideals. In *12th International Conference on Information Security and Cryptology (ICISC)*, 2009.
15. Mohamed Saied Emam Mohamed, Wael Said Abd Elmageed Mohamed, Jintai Ding, and Johannes Buchmann. Mxl2: Solving polynomial equations over gf(2) using an improved mutant strategy. In *Proceedings of Post-Quantum Cryptography 2008*, 2008. pre-print available at http://www.cdc.informatik.tu-darmstadt.de/reports/reports/MXL2.pdf.
16. William Stein et al. *SAGE Mathematics Software (Version 4.3)*. The Sage Development Team, 2008. Available at http://www.sagemath.org.

A Support Algorithms

Input: A – a $m \times n$ matrix
Input: r_{start} – an integer $0 \leq r_{\text{start}} < m$
Input: r_{end} – an integer $0 \leq r_{\text{start}} \leq r_{\text{end}} < m$
Input: c_{start} – an integer $0 \leq c_{\text{start}} < n$
Input: k – an integer $k > 0$
Input: T – a $2^k \times n$ matrix
Input: L – an integer array of length 2^k
begin
    for $r_{\text{start}} \leq i < r_{\text{end}}$ **do**
        $id = \sum_{j=0}^{k} A[i, c_{\text{start}} + j] \cdot 2^{k-j-1}$;
        $j \longleftarrow L[id]$;
        add row j from T to the row i of A starting at column c_{start};
    end
end

Algorithm 6: AddRowsFromTable

Input: A – a $m \times n$ matrix
Input: r_{start} – an integer $0 \leq r_{\text{start}} < m$
Input: c_{start} – an integer $0 \leq c_{\text{start}} < n$
Input: k – an integer $k > 0$
Result: Retuns an $2^k \times n$ matrix T
begin
    $T \longleftarrow$ the $2^k \times n$ zero matrix;
    for $1 \leq i < 2^k$ **do**
        $j \longleftarrow$ the row index of A to add according to the Gray code;
        add row j of A to the row i of T starting at column c_{start};
    end
    $L \longleftarrow$ integer array allowing to index T by k bits starting at column c_{start};
    return T, L;
end

Algorithm 7: MakeTable

Input: A – a $m \times n$ matrix
Input: r – an integer $0 \leq r < m$
Input: c – an integer $0 \leq c < n$
Input: k – an integer $k > 0$
Input: r_{end} – an integer $0 \leq r \leq r_{\text{end}} < m$
Result: Returns the rank $\overline{k} \leq k$ and puts the $\overline{k} \times (n - c)$ submatrix starting at $A[r, c]$ in reduced row echelon form.

```
begin
    r_s ⟵ r;
    for c ≤ j < c + k do
        found ⟵ False;
        for r_s ≤ i < r_end do
            for 0 ≤ l < j − c do // clear the first columns
                if A[i, c + l] ≠ 0 then  add row r + l to row i of A starting at column c + l;
            end
            if A[i, j] ≠ 0 then // pivot?
                Swap the rows i and r_s in A;
                for r ≤ l < r_s do // clear above
                    if A[l, j] ≠ 0 then  add row r_s to row l in A starting at column j;
                end
                r_s ⟵ r_s + 1;
                found ⟵ True;
                break;
            end
        end
        if found = False then
            return j - c;
        end
    end
    return j - c;
end
```

Algorithm 8: GaussSubmatrix

```
Input: A – a m × n matrix
Input: P – a permutation vector of length m
Input: Q – a permutation vector of length n
Result: PLS decomposition of A. Returns the rank of A.
begin
    r, c ← 0, 0;
    while r < m and c < n do
        found ⟵ False;
        for c ≤ j < n do // search for some pivot
            for r ≤ i < m do
                if A[i, j] then found ← True and break;
            end
            if found then break;
        end
        if found then
            P[r], Q[r] ⟵ i, j;
            swap the rows r and i in A;
            // clear below but preserve transformation matrix
            if j + 1 < n then
                for r + 1 ≤ l < m do
                    if A[l, j] then
                        add the row r to the row l starting at column j + 1;
                    end
                end
            end
            r, c ⟵ r + 1, j + 1;
        else
            break;
        end
    end
    for r ≤ i < m do P[i] ⟵ i ;
    for r ≤ i < n do Q[i] ⟵ i ;
    // Now compress L
    for 0 ≤ j < r do swap the columns j and Q[j] starting at row j;
    return r;
end
```

Algorithm 9: Gaussian PLS Decomposition

Input: A – a $m \times n$ matrix
Input: s_r – an integer $0 \leq s_r < m$
Input: s_c – an integer $0 \leq s_c < n$
Input: k – an integer $k > 0$
Input: P – a permutation vector of length m
Input: Q – a permutation vector of length n
Result: Returns the rank $\overline{k} \leq k$ and d_r – the last row considered.
Also puts the $\overline{k} \times (n - c)$ submatrix starting at (r, c) in PLS decomposition form.

```
begin
    done ⟵ all zero integer array of length k;
    for 0 ≤ r < k do
        found ⟵ False;
        for s_r + r ≤ i < m do // search for some pivot
            for 0 ≤ l < r do // clear before
                if done[l] < i then
                    if A[i, s_c + l] ≠ 0 then
                        add row s_r + l to row i in A starting at column s_c + l + 1;
                    end
                    done[l] ⟵ i;
                end
            end
            if A[i, s_c + r] ≠ 0 then
                found ⟵ True;
                break;
            end
        end
        if found = False then break;
        P[s_r + r], Q[s_r + r] ⟵ i, s_c + r;
        swap the rows s_r + r and i in A;
        done[r] ⟵ i;
    end
    d_r ⟵ max({done[i] | i ∈ {0, ..., k̄ − 1}});
    for 0 ≤ c_2 < k̄ and r + c_2 < n − 1 do // finish submatrix
        for done[c_2] < r_2 ≤ d_r do
            if A[r_2, r + c_2] ≠ 0 then
                add row r + c_2 to row r_2 in A starting at column r + c_2 + 1;
            end
        end
    end
    return r, d_r;
end
```

Algorithm 10: PLSSUBMATRIX

Breaking Elliptic Curves Cryptosystems using Reconfigurable Hardware

Junfeng Fan[†], Daniel V. Bailey[‡*] , Lejla Batina[†◁], Tim Güneysu[‡], Christof Paar[‡] and Ingrid Verbauwhede[†]

[†] ESAT/SCD-COSIC, Katholieke Universiteit Leuven and IBBT
Kasteelpark Arenberg 10, B-3001 Leuven-Heverlee, Belgium
{Firstname.Lastname}@esat.kuleuven.be
[‡] Horst Görtz Institute for IT Security, Ruhr University Bochum, Germany
{gueneysu, cpaar}@crypto.rub.de
[*] RSA, the Security Division of EMC, USA
dbailey@rsa.com
[◁] Radboud University Nijmegen, Netherlands

Abstract. This paper reports a new speed record for FPGAs in cracking Elliptic Curve Cryptosystems. We conduct a detailed analysis of different $\mathbf{F}_{2^m}$ multiplication approaches in this application. A novel architecture using optimized normal basis multipliers is proposed to solve the Certicom challenge ECC2K-130. We compare the FPGA performance against CPUs, GPUs, and even the Sony PlayStation 3. Our implementations show low-cost FPGAs outperform even multicore desktop processors and graphics cards by a factor of 2.

1 Motivation: Attacking ECC2K-130

Cryptosystems ensure the security, authenticity and privacy of data and users in most products nowadays. Elliptic-Curve Cryptosystems (ECC), independently invented by Miller [18] and Koblitz [15], are now commonplace in both the academic literature and practical deployments. These systems allow shorter keylengths, ciphertexts, and signatures than other conventional cryptosystems, *e.g.*, RSA. In addition, thanks to these smaller operands, ECC offer higher performance and lower power when compared with other systems. Especially in pervasive computing applications, ECC admit valuable optimizations in computing and communication complexity.

The security of ECC relies on the difficulty of Elliptic Curve Discrete Logarithm Problem (ECDLP) [3]. Briefly speaking, ECDLP is to find an integer n for two points P and Q on an elliptic curve E such that $Q \equiv [n]P$. To use ECC in the real world, practitioners need to know: how big should the parameters (or, colloquially, the "key size") be to avoid practical attacks? Choosing parameters too small allows computational attackers to solve the ECDLP instance, while choosing parameters too large wastes time, communication, and storage. To encourage investigation of these issues, researchers at Certicom Corp. published a list of ECDLP challenges in 1997 [7].

Smaller members of the list of Certicom ECDLP challenge problems have been solved. Escott *et al.* report on their successful attack on ECCp-97, an ECDLP in a group of roughly 2^{97} elements [8]. A larger instance, ECC2-109 was solved by Monico *et al.* [6]. Bos *et al.* analyze and solve the ECDLP in a group of roughly 2^{112} elements using PS3 [4].

This paper reports on our effort to solve one of the Certicom ECDLP challenge problems using FPGAs. We focus on the ECC2K-130 challenge. ECC2K-130 is a Koblitz curve challenge over $\mathbf{F}_{2^{131}}$. Compared to previous attacks, which are mostly implemented in software on general-purpose workstations, our work obtains a much higher performance-cost ratio by using FPGA platforms.

The rest of the paper is organized as follows. Section 2 summarizes related work. In Section 3 we give a short description of the function that we implement on FPGA. Section 4 and 5 explore different algorithms and architectures. Section 6 reports on the results, and finally we conclude the paper in Section 7.

2 Related Work

The strongest known attacks against the ECDLP are generic attacks based on Pollard' rho method [20, 5]. Further improvements including parallelization and the use of group automorphisms were made by Wiener and Zuccherato [26], van Oorschot *et al.* [25], and Gallant *et al.* [9]. The parallelized Pollard rho method consists of parallel loops that search for distinguished points. Each loop starts from a random point on E and ends when a distinguished point is hit. The core function is thus the update function, also known as iteration function. Our work implements this function along with its improvements.

FPGAs have been applied to the Pollard rho method in several previous works. Güneysu *et al.* analyze ECDLPs over fields of odd-prime characteristic [11, 10], targeting a machine with 128 low-cost FPGAs. They extrapolate that to break an ECDLP in a group of roughly 2^{131} elements using this machine as taking over a thousand years.

Similarly, Meurice de Dormale *et al.* apply FPGAs to the ECDLP [17]. Here, they use characteristic-two finite fields, but restrict their inquiry to polynomial basis. Although conventional wisdom has held that low-weight polynomial basis is a better choice, in our application we can take advantage of the free repeated squarings (2^n-th powers) offered in normal basis. In addition, recent progress on normal-basis multiplication by Shokrollahi *et al.* [24] and Bernstein and Lange [2] further improve the prospects for normal basis. Our work is the first FPGA implementation of parallelized Pollard rho method using normal basis multiplication.

This work is part of a global distributed effort to break the largest ECDLP ever solved [1]. While that paper summarizes the overall effort, here we focus on the FPGA implementation.

The Contribution of This Paper. As part of this effort, this paper explores FPGA implementation options for the core finite-field arithmetic operations as well as architectures. An in-depth comparison between polynomial basis

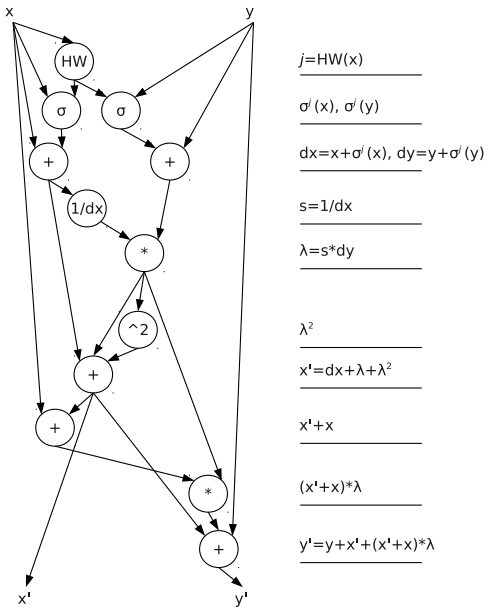


Fig. 1: Dataflow of the iteration function: $P_{i+1} = \sigma^j(P_i) \oplus P_i$. (Note here $\sigma^j(x) = x^{2^j}$)

multiplier, Type-II normal basis multiplier and Shokrollahi's multiplier is given. Especially, this is the first FPGA implementation of Shokrollahi's multiplication algorithm. Our work proves the superiority of an FPGA platform over other specialized architectures and its suitability for the tasks that are computationally demanding. Results in this paper are relevant both to the cryptanalytic community as well as those interested in fast cryptographic implementations in normal basis.

3 Iteration function

We briefly describe the iteration function in this section. The general attack strategy and the design rationale behind of the iteration function can be found in [1]. The iteration function is implemented on FPGA.

Our condition for a point $P_i(x, y)$ to be a distinguished point is that $\mathrm{HW}(x) \leq 34$, where x is represented using type-II normal-basis and $\mathrm{HW}(x)$ returns the Hamming weight of x. Our iteration function is defined as

$$P_{i+1} = \sigma^j(P_i) \oplus P_i, \tag{1}$$

where $\sigma^j(P_i) = (x^{2^j}, y^{2^j})$ and $j = ((\mathrm{HW}(x_{P_i})/2) \bmod 8) + 3$. To solve ECC2K-130, about $2^{60.9}$ iterations are expected in total [1].

An efficient implementation of the iteration function is thus the key step towards a fast attack. Given $P_i(x, y)$, the iteration function computes $P_{i+1}(x', y')$ using Eq.(1). Fig. 1 shows the data flow of the iteration function.

4 Optimizing Finite-Field Operations

The iteration function consists of two multiplications, one inversion and several squarings in $\mathbf{F}_{2^m}$. Thus, fast finite-field arithmetic is essential to optimize the attack.

A vast body of literature exists on finite field arithmetic, and we are free to choose from a variety of representations and algorithms. For example, we can use either polynomial basis or normal basis for element representation, and an iteration can be performed using eithr Extended Euclidean Algorithm (EEA) or Fermat's little theorem. This leads to an important question: which configuration (basis, multiplication algoirthm, inversion algorithm) ensures the most efficient implementation of the aforementioned iteration function? We try to answer this question with complexity analysis and design-space exploration.

4.1 Multiplication

An element of $\mathbf{F}_{2^{131}}$ can be represented in both polynomial basis $\mathbf{P}$ and Type-II normal basis $\mathbf{N}$, where

$$\mathbf{P} = \{\ 1, w, w^2, \cdots, w^{130}\},$$
$$\mathbf{N} = \{\ \gamma + \gamma^{-1}, \gamma^2 + \gamma^{-2}, \gamma^{2^2} + \gamma^{-2^2}, \cdots, \gamma^{2^{130}} + \gamma^{-2^{130}}\}.$$

Here w is a root of an irreducible polynomial of degree 131, while γ is a primitive 263^{rd} root of unity. Multiplication in polynomial basis has long been considered more efficient than normal basis. On the other hand, squaring in normal basis is simply a circular shift. Moreover, computing any power α^{2^n} can be performed by circularly-shifting by n positions. We implemented both options for comparison.

Besides conventional multiplication algorithms in polynomial and normal basis, we also implemented a recently reported hybrid algorithm, due to Shokrollahi [24]. This algorithm uses only $\mathrm{O}(m \log m)$ operations for the basis conversion. When multiplication is needed, two field elements are converted to polynomial basis, a polynomial-basis multiplication is carried out, then the results are converted back to normal basis and reduced. This paper includes the first FPGA implementation of this method.

Polynomial-Basis Multiplier Algorithms for multiplication in polynomial basis consist of two steps, polynomial multiplication and modular reduction. They can be carried out separately or interleaved. Given two elements $A(w) = \sum_{i=0}^{m-1} a_i w^i$ and $B(w) = \sum_{i=0}^{m-1} b_i w^i$, a bit-serial modular multiplication algorithm is shown in Alg. 1.

Algorithm 1 Bit-serial modular multiplication in $\mathbf{F}_{2^m}$

Input: $A(w) = \sum_{i=0}^{m-1} a_i w^i$, $B(w) = \sum_{i=0}^{m-1} b_i w^i$ and $P(w)$.
Output: $A(w)B(w) \bmod P(w)$.

1: $C(w)(= \sum_{i=0}^{m} c_i w^i) \leftarrow 0$;
2: **for** $i = m-1$ to 0 **do**
3: $\quad C(w) \leftarrow w(C(w) + c_m P(w) + b_i A(w))$;
4: **end for**

Return: $C(w)/w$.

It is well known that one way to reduce area complexity is to use a polynomial $P(w)$ with special form, such as low weight. For $\mathbf{F}_{2^{131}}$ there exists an irreducible pentanomial $P(w) = w^{131} + w^{13} + w^2 + w + 1$. Thus, the complexity of step 3 in Alg. 1 is $(m + 4)$ XOR and $(m + 4)$ AND operations.

One can also compute $C(w) = A(w)B(w) = \sum_{i=0}^{2m-2} c_i w^i$ first, and then reduce it with $P(w)$. In this case, the Karatsuba method [14] can be used to reduce the complexity of polynomial multiplication. The reduction phase requires $\mathrm{O}(m)$ AND and XOR operations when low-weight $P(w)$ exists. For example, when $P(w)$ is a pentanomial, reducing $C(w)$ requires around $4m$ AND and $4m$ XOR operations. The overall complexity of a modular multiplication is $\mathrm{M}(m)+\mathrm{O}(m)$, where $\mathrm{M}(m)$ is the complexity of an m-bit polynomial multiplication.

Normal-Basis Multiplier The normal-basis multiplier by Sunar and Koç [23] employs the fact that an element $(\gamma^{2^i} + \gamma^{2^{-i}})$ for $i \in [1, m]$ can be written as $(\gamma^j + \gamma^{-j})$ for some $j \in [1, m]$. As a result, the following basis **pN** is equivalent to **N**:

$$\mathbf{pN} = \{\gamma + \gamma^{-1}, \gamma^2 + \gamma^{-2}, \gamma^3 + \gamma^{-3}, \cdots, \gamma^{131} + \gamma^{-131}\}.$$

pN is also known as permuted normal basis. Let $\beta_i = (\gamma^i + \gamma^{-i})$, then an element T in $GF(2^m)$ is represented as $T = \sum_{i=1}^{m} t_i \beta_i$. One multiplication of A and B represented with **pN** requires m^2 AND and $3m(m-1)/2$ two-input XORs.

This algorithm is then adapted by Kwon [16] to deduce a systolic multiplier. Compared to the Sunar-Koç multiplier, Kwon's architecture shown in Fig. 2, is highly regular and thus can be implemented in a digit-serial manner. On the other hand, it has higher complexity: $2m$ AND and $2m$ XOR gates for a bit-serial multiplier.

Shokrollahi's multiplier Shokrollahi discovered an efficient algorithm for basis conversion between permuted normal basis and polynomial basis. Later, Bernstein and Lange proposed further optimizations to this approach including a

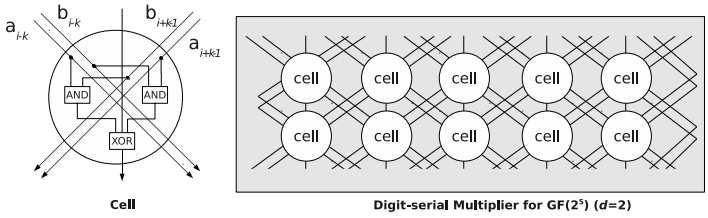


Fig. 2: Modular multiplier in $GF(2^m)$ using Kwon's algorithm

more straight-forward conversion function. More details on this multiplication method can be found in their recent work [2]. The new polynomial basis (**nP**) is defined in [2] as Type-II polynomial basis.

$$\mathbf{nP} = \{(\gamma + \gamma^{-1}), (\gamma + \gamma^{-1})^2, \cdots, (\gamma + \gamma^{-1})^m\}$$

which leads to a hybrid normal-basis multiplication algorithm. We denote A_{nP} and A_{pN} the representation of A using **nP** and **pN**, respectively. A multiplication then proceeds as follows.

1. converting to polynomial basis: $A_{nP} \leftarrow A_{pN}$, $B_{nP} \leftarrow B_{pN}$,
2. polynomial multiplication: $C_{nP} \leftarrow A_{nP} B_{nP}$,
3. converting back to normal basis ($2m$-bit conversion): $C_{pN} \leftarrow C_{nP}$,
4. reduction (folding).

Let S_{pN2nP} be a transformation function that converts A_{pN} to A_{nP}. The essential observation by Shokrollahi is that basis conversion can be recursively broken down to half-length transformations. Let f_{pN} and f_{nP} be corresponding representations of f in **pN** and **nP**, respectively,

$$f_{pN} = [f_1\ f_2\ \cdots\ f_8] \odot [(\gamma + \gamma^{-1})\ (\gamma^2 + \gamma^{-2})\ \cdots\ (\gamma^8 + \gamma^{-8})]^T,$$
$$f_{nP} = [g_1\ g_2\ \cdots\ g_8] \odot [(\gamma + \gamma^{-1})\ (\gamma + \gamma^{-1})^2\ \cdots\ (\gamma + \gamma^{-1})^8]^T,$$

Converting f_{pN} to f_{nP} can be then performed with two 4-bit transformations:

$$\{g_1, g_2, g_3, g_4\}_{\mathbf{nP}} \xleftarrow{S_{pN2nP}} \{f_1 + f_7, f_2 + f_6, f_3 + f_5, f_4\}_{\mathbf{pN}}$$
$$\{g_5, g_2, g_7, g_8\}_{\mathbf{nP}} \xleftarrow{S_{pN2nP}} \{f_5, f_6, f_7, f_8\}_{\mathbf{pN}}$$

The **pN** $\rightarrow$ **nP** and **nP** $\rightarrow$ **pN** conversion in $\mathbf{F}_{2^m}$ takes $(m/2)\log_2(m/4)$ operations each. In total, one field multiplication takes about $\mathrm{M}(m) + m\log_2 m + m\log_2(m/4)$ operations. A more detailed discussion on the complexity can be found in [2].

Based on the analysis above, we can draw the following conclusions:

– A bit-serial multiplier using polynomial basis has a lower area complexity than one using normal basis.

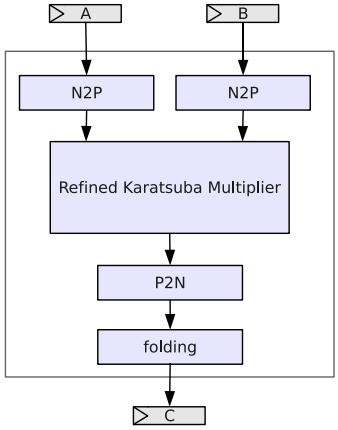


Fig. 3: Shokrollahi multiplier

– When low-weight polynomials exist, Shokrollahi's multiplication algorithm is likely to have a higher complexity than convential polynomial basis multiplication since the base conversion step is more complex than the polynomial reduction.

Though it seems that polynomial basis should be used, normal basis offers several advantages in this specific application. First, the Pollard rho iteration function requires the Hamming weight of x-coordinate represented in normal basis. In fact, thanks to the Frobenius endomorphism, checking HW of x in normal basis checks 131 points simultaneously. This brings a speedup of $\sqrt{131}$ to the attack [1]. Second, the iteration function includes two $\sigma(x, j) = x^{2^j}$ routines (known as m-squaring). In normal basis, σ is essentially a circular shift of j bits, and thus can be performed in one cycle. Gains in m-squaring compensate the loss in multiplications.

4.2 Inversion

Inversion is the most costly of the four basic field operations. Two broad approaches are found in the literature: the Extended Euclidean Algorithm (EEA) and Fermat's Little Theorem (FLT). In polynomial basis, the binary variant of EEA is generally faster, while the variant of FLT attributed to Itoh and Tsujii [13] is the better choice in normal basis because squaring is free. Itoh-Tsujii reduces the problem of extension-field inversion to exponentiation and inversion in the subfield. In polynomial basis, exponentiation is generally quite expensive owing to the need to explicitly compute squares, making EEA a better choice. Itoh-Tsujii raises an element to the exponent $r - 1 = 2 + 2^2 + \cdots + 2^{m-1}$, using the fact that in normal basis, squaring is free. In addition, this algorithm uses an addition chain to reduce the number of multiplications: in $\mathbf{F}_{2^{131}}$, our addi-

Algorithm 2 Simultaneous inversion (Batch size = 3)

Input: $\alpha_1, \alpha_2, \alpha_3$.
Output: α_1^{-1}, α_2^{-1} and α_3^{-1}.

1: $d_1 \leftarrow \alpha_1$
2: $d_2 \leftarrow d_1\alpha_2$
3: $d_3 \leftarrow d_2\alpha_3$
4: $u \leftarrow d_3^{-1}$
5: $t_3 \leftarrow ud_2$, $u \leftarrow u\alpha_3$
6: $t_2 \leftarrow ud_1$, $u \leftarrow u\alpha_2$
7: $t_1 \leftarrow u$

Return: t_1, t_2, t_3.

tion chain has length nine (1, 2, 4, 8, 16, 32, 64, 128, 130). The net result is a complexity of eight field multiplications to compute an inverse.

To further reduce the computation costs for inverses, we employ Montgomery's trick that batches multiple inversions by trading inversions for multiplications [19]. Alg.2 shows this method to invert three inputs. Indeed, we can trade one inversion for three extra multiplications. As a result, one iteration function uses 5M + $(1/n)$I where n is the batch size.

5 Architecture Exploration

The architecture of the engine has a fundamental impact on the overall throughput. Among all the design options the following three are of great importance.

1. Multiplier architecture
2. Memory architecture
3. Inverter architecture

As an architecture exploration, we implemented three different architectures using different types of multipliers.

5.1 Architecture I: Load-Store

As a starting point, we take a programmable elliptic-curve coprocessor as the platform. A digit-serial polynomial multiplier (see [21] for details) is used. A dedicated squarer is included for squaring. In each loop, the x-coordinate is converted into its normal basis representation, and its Hamming weight is counted. This adds a base conversion block and a Hamming weight computation block.

On this platform, squaring or addition takes two clock cycles, while multiplication takes $\lfloor n/d \rfloor + 1$ cycles given a digit-size d. The design is synthesized using ISE 11.2 and the target FPGA is Xilinx Spartan-3 XC3S5000 (4FG676). Implementation results show that $d = 22$ gives the best trade-off in terms of area-delay product.

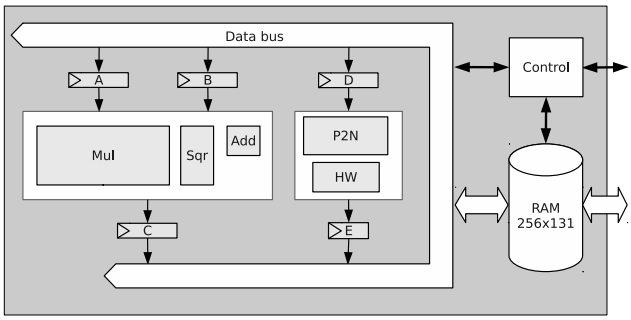


Fig. 4: *Archi-I*: ECC processor using polynomial basis multiplier

The design consumes 3,656 slices, including 1,468 slices for the multiplier, 75 slices for the squarer, 1,206 slices for the base conversion, 117 slices for Hamming weight calculation.

One Pollard rho iteration takes 71 cycles, among them 35 cycles are used for multiplication. The design achieves a maximum clock frequency of 101 MHz, and one iteration takes 704 ns. The m-squaring is performed with m successive squarings. Obviously, this architecture is not efficient. The m-squaring operations can be largely sped up when normal basis is used.

5.2 Architecture II: Type-II Normal Basis Multiplier

Archi-II uses a digit-serial normal basis multiplier. The structure of the multiplier is shown in Fig. 2. When m is small, a full systolic architecture can be used, performing one multiplication per cycle. However, a systolic array for $\mathbf{F}_{2^{131}}$ is too large (more than 20,000 slices on Spartan-3). Thus, a digit-serial architecture is used. Implementation results show that $d = 13$ gives the lowest area-delay product. The multiplier alone uses 2,093 slices.

The basis-conversion component in *Archi-I* is no longer needed in *Archi-II*, saving 1,468 slices. In total, the design uses 2,578 slices. On this platform, one Pollard rho iteration takes 81 cycles, including 55 cycles used for multiplication. Compared to *Archi-I*, the m-squaring operation is largely improved. However, the multiplier becomes much slower than that in *Archi-I*. The design achieves a maximum clock frequency of 125 MHz, and one iteration takes 648 ns.

5.3 Architecture III: Fully Pipelined Iteration Function

Archi-III unrolls the Pollard rho iteration such that a throughput of one iteration per cycle is achieved. Remember that 5 multiplications are required for each iteration, as a result, five normal basis multipliers are used. The design is fully pipelined. Since additions and squarings are embedded in the pipeline, it increases the delay of one iteration but does not affect the throughput.

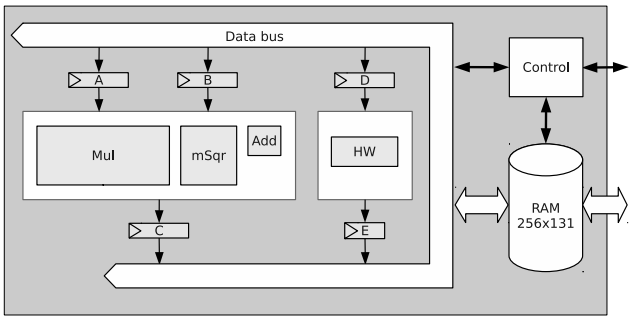


Fig. 5: *Archi-II*: ECC processor using normal basis

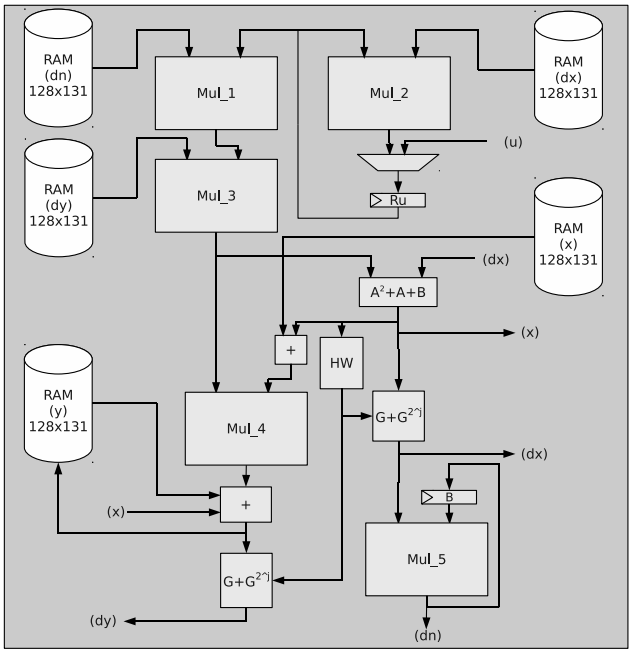


Fig. 6: *Archi-III*: pipelined processor using Shokrollahi multipliers

At the first glance, fully expanding the iteration function seams impossible due to the inversion in each iteration. Indeed, after dx is generated in Fig.1, inverting dx will take too much area to fit one FPGA. The solution is to start the pipeline after the real inversion ($u \leftarrow d_n^{-1}$) is performed.

Fig 6 shows the architecture that supports the expanded iteration function. In total, five multipliers are used. Before the starting of the pipeline, x,y,dx and dy of P_i are stored in RAM (x), (y), (dx) and (dy), respectively. RAM (dn)

keeps the intermediate data d_i of Alg. 2, and u is ready in register Ru. After the starting of the pipeline, the five multipliers perform the following operations.

- Mul_1: $t_i \leftarrow ud_{i-1}$
- Mul_2: $u \leftarrow u\alpha_i$
- Mul_3: $\lambda \leftarrow dy(1/dx)$
- Mul_4: $\lambda(x' + x)$
- Mul_5: $d'_i \leftarrow d'_i dx'$

Mul_1, Mul_2 and Mul_5 are used by batch inversion (Alg. 2), while Mul_3 and Mul_4 are used for point addition (Fig. 1).

The inversion ($u \leftarrow d_n^{-1}$) is performed by another multiplier together with a squarer. In order to keep full use of the engine, we interleave two groups of iteration function. When the engine is executing one group, the inverter is performing inversion for the other group.

The implementation of *Archi-III* consumes 22,195 slices and 20 block RAMs (RAMB16s) on Xilinx Spartan-3 XC3S5000 FPGA. One fully pipelined Shokrollahi's multiplier uses 4,391 slices. The inverter itself uses 4,761 slices. In total, the design uses 26,731 slices. The post placing-and-routing results show that this design can achieve a maximum clock frequency of 111 MHz.

6 Results and Analysis

The ECC2K-130 attack using FPGAs is conducted using an improved version of the COPACOBANA cluster described in [10, 12], also known as RIVYERA [22]. It is populated with 128 Spartan-3 XC3S5000 FPGAs and an optional 32MB memory per FPGA combined in one 19" housing. All FPGAs are connected with two opposite directed, systolic ring networks that directly interface with the PC (which is integrated in the same housing) via two individual PCI Express communication controllers. Although this setup can obviously provide a significant amount of bandwidth due to its local communication paths, the ECC2K-130 attack design actually requires only moderate communication performance.

Table 1 summarizes the implementation results on a Spartan-3 XC3S5000 FPGA. Based on the available resources (33,280 slices and 104 BRAMs) of each XC3S5000 FPGA, we also estimated that at most 9 clones of *Archi-I* or 12 clones of *Archi-II* can be implemented on a single FPGA. For *Archi-III*, one clone uses 80% of the available resources of one FPGA.

Table 1: Cost for one Pollard's rho iteration for various architectures

	Digit	Area		Freq.	Delay per Step		Throughput
	size	#slice	#BRAM	[MHz]	Cycles	[ns]	per FPGA
Archi-I: Polynomial basis	22	3,656	4	101	71	703	12.8×10^6
Archi-II: Type-II ONB	13	2,578	4	125	81	648	18.5×10^6
Archi-III: Shokrollahi's	-	26,731	20	111	23 (stages)	206	111×10^6

The throughput per engine, T_e is computed as $T_e = \frac{Freq.}{Cycles\ per\ step}$, and the throughput per FPGA T_c is computed as $T_c = T_e * l$. Here, l is the number of engines on a single FPGA. Compared with *Archi-I*, *Archi-II* has smaller area and shorter delay. In other words, Type-II optimal normal basis has significant advantages for this application. On the other hand, *Archi-III* achieves a 8.6 times speedup over *Archi-II*. The improvement mainly comes from the efficient field arithmetic and the special architecture. The use of Shokrollahi's algorithm significantly improved the throughput of a multiplier, while the expansion of the iteration function hides delays caused by addition and squaring in the pipeline.

Table 2: Performance comparison

Source	Platform	Challenge	Frq. [MHz]	Throughput [$\times 10^6$]
[17]	Xilinx FPGA S3E1200-4	ECC2-131	100	10.0
[1]	Cell CPU 6 SPEs, 1 PPE	ECC2K-130	3,200	27.7
[1]	Graphics Card GTX 295	ECC2K-130	1,242	54.0
[1]	Core 2 Extreme Q6850, 4 cores	ECC2K-130	3,000	22.5
This work (*Archi-III*)	Xilinx FPGA XC3S5000	ECC2K-130	111	111

In Table 2 we compare our results with similar implementations on different platforms.

To our knowledge, this is the first FPGA implementation using fast normal-basis multiplication to attack ECDLP. As a point of comparison, we look into the work of Meurice de Dormale, et al. [17]. They do not specifically target Koblitz curves and they are using different FPGAs, which makes a fair comparison difficult.

On the other hand, there is an interesting comparison between implementations of the same attack (and thus iteration function) on different platforms other than FPGAs. Within the whole project [1] efforts have been made to speed-up the iteration function on CPUs, GPUs and PlayStation 3. These platforms are state-of-the-art processors supporting massive parallelism. Compared to these platforms, our FPGA implementation is at least 2 times faster in terms of throughput.

The whole complexity of this attack is around $2^{60.9}$ iterations. We estimate that given five RIVYERA FPGA clusters, the ECC2K-130 challenge can be solved in one year.

7 Conclusion

A new efficiency record for FPGAs in cracking public-key cryptosystems based on elliptic curves is reported. We conduct a detailed comparison of different architectures for normal-basis multipliers suited this application. The comparison includes the first FPGA implementation report for one of these architectures. Our results show that even low-cost FPGAs outperform CPUs, the PlayStation 3 platform and even GPUs.

Acknowledgements

This work was supported in part by K.U. Leuven-BOF (OT/06/40), by the IAP Programme P6/26 BCRYPT of the Belgian State (Belgian Science Policy), by FWO project G.0300.07, and by the European Commission through the ICT programme under contract ICT-2007-216676 ECRYPT II.

References

1. D. V. Bailey, L. Batina, D. J. Bernstein, P. Birkner, J. W. Bos, H. Chen, C. Cheng, G. van Damme, G. de Meulenaer, L. J. D. Perez, J. Fan, T. Güneysu, F. Gurkaynak, T. Kleinjung, T. Lange, N. Mentens, R. Niederhagen, C. Paar, F. Regazzoni, P. Schwabe, L. Uhsadel, A. Van Herrewege, and B. Yang. Breaking ECC2K-130. Cryptology ePrint Archive, Report 2009/541, 2009. `http://eprint.iacr.org/`.
2. D.J. Bernstein and T. Lange. Type-II Optimal Polynomial Bases. to appear in WAIFI 2010 - `http://binary.cr.yp.to/opb-20100209.pdf`.
3. I. Blake, G. Seroussi, and N. Smart. *Advances in Elliptic Curve Cryptography (London Mathematical Society Lecture Note Series)*. Cambridge University Press, New York, NY, USA, 2005.
4. J. W. Bos, M. E. Kaihara, T. Kleinjung, A. K. Lenstra, and P. L. Montgomery. On the security of 1024-bit RSA and 160-bit elliptic curve cryptography: version 2.1. Cryptology ePrint Archive, Report 2009/389, 2009. `http://eprint.iacr.org/2009/389`.
5. R. P. Brent and J. M. Pollard. Factorization of the eighth Fermat number. *Mathematics of Computation*, 36:627–630, 1981.
6. Certicom. Certicom ECC Challenge. `http://www.certicom.com/images/pdfs/cert_ecc_challenge.pdf`, 1997.
7. Certicom. ECC Curves List. `http://www.certicom.com/index.php/curves-list`, 1997.
8. A. Escott, J. Sager, A. Selkirk, and D. Tsapakidis. Attacking elliptic curve cryptosystems using the parallel Pollard rho method. *CryptoBytes (The technical newsletter of RSA Laboratories)*, 4:15–19, 1998.
9. R. P. Gallant, R. J. Lambert, and S. A. Vanstone. Improving the parallelized Pollard lambda search on anomalous binary curves. *Mathematics of Computation*, 69(232):1699–1705, 2000.
10. T. Güneysu, T. Kasper, M. Novotný, C. Paar, and A. Rupp. Cryptanalysis with COPACOBANA. *IEEE Transactions on Computers*, 57(11):1498–1513, November 2008.

11. T. Güneysu, C. Paar, and J. Pelzl. Attacking elliptic curve cryptosystems with special-purpose hardware. In *Proceedings of the 2007 ACM/SIGDA 15th international symposium on Field programmable gate arrays*, page 215. ACM, 2007.
12. T. Güneysu, G. Pfeiffer, C. Paar, and M. Schimmler. Three Years of Evolution: Cryptanalysis with COPACOBANA. In *Workshop on Special-purpose Hardware for Attacking Cryptographic Systems - SHARCS 2009*, September 9-10 2009.
13. T. Itoh and S. Tsujii. A Fast Algorithm for Computing Multiplicative Inverses in $GF(2^m)$ Using Normal Bases. *Inf. Comput.*, 78(3):171–177, 1988.
14. A. Karatsuba and Y. Ofman. Multiplication of many-digital numbers by automatic computers. 145:595–596, 7 1963.
15. N. Koblitz. Elliptic curve cryptosystems. *Mathematics of computation*, 48(177):203–209, 1987.
16. S. Kwon. A low complexity and a low latency bit parallel systolic multiplier over GF(2^m) using an optimal normal basis of type II. In *IEEE Symposium on Computer Arithmetic - ARITH-16*, pages 196–202, 2003.
17. G. Meurice de Dormale, P. Bulens, and J. J. Quisquater. Collision Search for Elliptic Curve Discrete Logarithm over GF(2^m) with FPGA. In *Workshop on Cryptographic Hardware and Embedded Systems (CHES 2007)*, pages 378–393. Springer, 2007.
18. V.S. Miller. Use of elliptic curves in cryptography. In *Advances in Cryptology CRYPTO 85 Proceedings*, pages 417–426, 1986.
19. P. L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48:243–264, 1987.
20. J. M. Pollard. Monte Carlo methods for index computation (mod p). *Mathematics of Computation*, 32:918–924, 1978.
21. K. Sakiyama, L. Batina, B. Preneel, and I. Verbauwhede. Multicore Curve-Based Cryptoprocessor with Reconfigurable Modular Arithmetic Logic Units over GF(2^n). *IEEE Trans. Computers*, 56(9):1269–1282, 2007.
22. SciEngines GmbH. RIVYERA S3-5000, 2010. `http://www.sciengines.com/joomla/index.php?option=com_content&view=article&id=60&Itemid=74`.
23. B. Sunar and Ç.K. Koç. An Efficient Optimal Normal Basis Type II Multiplier. *IEEE Transactions on Computers*, 50:83–87, 2001.
24. J. v. z. Gathen, A. Shokrollahi, and J. Shokrollahi. Efficient multiplication using type 2 optimal normal bases. In Claude Carlet and Berk Sunar, editors, *WAIFI*, volume 4547 of *Lecture Notes in Computer Science*, pages 55–68. Springer, 2007.
25. P. C. van Oorschot and M. J. Wiener. Parallel collision search with cryptanalytic applications. *Journal of Cryptology*, 12(1):1–28, 1999.
26. M. J. Wiener and R. J. Zuccherato. Faster attacks on elliptic curve cryptosystems. In *Selected Areas in Cryptography*, volume 1556 of *LNCS*, pages 190–200, 1998.

Fast Exhaustive Search for Polynomial Systems in $\mathbb{F}_2$

Charles Bouillaguet[1], Hsieh-Chung Chen[2], Chen-Mou Cheng[3],
Tung Chou[3], Ruben Niederhagen[3,4], Adi Shamir[1,5], and Bo-Yin Yang[2]

[1] Ecole Normale Supérieure, Paris, France, charles.bouillaguet@ens.fr
[2] Institute of Information Science, Academia Sinica, Taipei, Taiwan, {kc,by}@crypto.tw
[3] National Taiwan University, Taipei, Taiwan, {doug,blueprint}@crypto.tw
[4] Technische Universiteit Eindhoven, the Netherlands, ruben@polycephaly.org
[5] Weizmann Institute of Science, Israel, adi.shamir@weizmann.ac.il

Abstract. We analyze how fast we can solve general systems of multivariate equations of various low degrees over $\mathbb{F}_2$; this is a well known hard problem which is important both in itself and as part of many types of algebraic cryptanalysis. Compared to the standard exhaustive search technique, our improved approach is more efficient both asymptotically and practically. We implemented several optimized versions of our techniques on CPUs and GPUs. Our technique runs more than 10 times faster on modern graphic cards than on the most powerful CPU available. Today, we can solve 48+ quadratic equations in 48 binary variables on a 500-dollar NVIDIA GTX 295 graphics card in 21 minutes. With this level of performance, solving systems of equations supposed to ensure a security level of 64 bits turns out to be feasible in practice with a modest budget. This is a clear demonstration of the computational power of GPUs in solving many types of combinatorial and cryptanalytic problems.
Keywords: multivariate polynomials, solving systems of equations, exhaustive search, parallelization, Graphic Processing Units (GPUs)

1 Introduction

Solving a system of m nonlinear polynomial equations in n variables over $\mathbb{F}_q$ is a natural mathematical problem that has been investigated by various research communities. The cryptographers are among the interested parties since an NP-complete problem whose random instances seem hard could be used to design cryptographic primitives, as witness the development of multivariate cryptography in the last few decades, using one-way trapdoor functions such as HFE, SFLASH, and QUARTZ [13, 22, 23], as well as stream ciphers such as QUAD [5].

Conversely, in "algebraic cryptanalysis" one distills from a cryptographic primitive a system of multivariate polynomial equations with the secret among the variables. This does not break AES as first advertised, but does break KeeLoq [12], for a recent example, and find a faster collision on 58-round SHA-1 [26].

Since the pioneering work by Buchberger [10], Gröbner-basis techniques have been the most prominent tool for this problem, especially after the emergence of faster algorithms such as $\mathbf{F_4}$ or $\mathbf{F_5}$ [16, 17], which broke the first HFE challenge [18]. The cryptographic community independently rediscovered some of the ideas underlying efficient Gröbner-basis algorithms as of the XL algorithm [14] and its variants. They also introduced techniques to deal with special cases, particularly that of sparse systems [1,25].

In this paper we take a different path, namely improving the standard and seemingly well-understood exhaustive search algorithm. When the system consists of n randomly chosen quadratic equations in n variables, all the known solution techniques have exponential complexity. In particular, Gröbner-basis methods have an advantage on very overdetermined systems (with many more equations than unknowns) and systems with certain algebraic "weaknesses", but were shown to be exponential on "generic" enough systems in [2, 3]. In addition, the computation of a Gröbner basis is often a memory-bound process; since memory is more expensive than time at the scale of interest, such sophisticated techniques can be inferior in practice when compared to simple testing of all the possible solutions, which uses almost no memory.

For "generic" quadratic systems, experts believe [2, 27] that Gröbner basis methods will go up to degree D_0, which is the minimum possible D where the coefficient of t^D in $(1+t)^n(1+t^2)^{-m}$ goes negative, and then require the solution of a system of linear equations with $T \gtrsim \binom{n}{D_0-1}$ variables. This will take at least $\mathrm{poly}(n) \cdot T^2$ bit-operations, assuming we can afford a sufficiently large amount of memory and that we can solve such a linear system of equations with non-negligible probability in $O(N^{2+o(1)})$ time for N variables. For example, if we assume we can operate a Wiedemann solver on a $T \times T$ submatrix of the extended Macaulay matrix of the original system, then the polynomial is $3n(n-1)/2$. When $m = n = 200$, $D_0 = 25$, making the value of T exceeds 2^{102}; even taking into consideration guessing before solving [7, 28], we can still easily conclude that Gröbner-basis methods would not outperform exhaustive search in the practically interesting range of $m = n \leq 200$.

The questions we address are therefore: how far can we go, on both theoretical and practical sides, by pushing exhaustive search further? Is it possible to design more efficient exhaustive search algorithms? Can we get better performance using different hardware such as GPUs? Is it possible to solve *in practice*, with a modest budget, a system of 64 equations in 64 unknowns over $\mathbb{F}_2$? Less than 15 years ago, this was considered so difficult that it even underlied the security of a particular signature scheme [21]. Intuitively, some people may consider an algebraic attack that reduces a cryptosystem to 64 equations of degree 4 in 64 $\mathbb{F}_2$-variables to be a successful practical attack. However, the matter is not that easily settled because the complexity of a naïve exhaustive search algorithm would actually be *much higher* than 2^{64}: simply testing all the solutions in a naïve way results in $2 \cdot \binom{64}{4} \cdot 2^{64} \approx 2^{84}$ logical operations, which would make the attack hardly feasible even on today's best available hardware.

Our Contribution. Our contribution is twofold. On the theoretical side, we present a new type of exhaustive search algorithm which is both asymptotically and practically faster than existing techniques. In particular, we show that finding *all* zeroes of a single degree-d polynomial in n variables requires just $d \cdot 2^n$ bit operations. We then extend this technique and show how to find all the common zeroes of m random quadratic polynomials in $\log_2 n \cdot 2^{n+2}$ bit operations, which is only slightly higher. Surprisingly, this complexity is *independent of the number of equations* m.

On the practical side, we have implemented our new algorithms on x86 CPUs and on NVIDIA GPUs. While our CPU implementation is fairly optimized using vector instructions, our GPU implementation running on one single NVIDIA GTX 295 graphics card runs up to 13 times faster than the CPU implementation using all four cores of an Intel quad-code Core i7 at 3 GHz, one of the fastest CPUs currently available. Today, we can solve 48+ quadratic equations in 48 binary variables using just an NVIDIA GTX 295 graphics card in 21 minutes. This device is available for about $500. It would be 36 minutes for cubic equations and two hours for quartics. The 64-bit signature challenge [21] can thus be broken with 10 such cards in 3 months, using a budget of $5000. Even taking into account Moore's law, this is still quite an achievement.

In contrast, the implementation of F_4 in `MAGMA-2.16`, often cited as the best Gröbner-basis solver *commercially* available today, will completely use up 64 GB of RAM in solving just 25 cubic equations in as many $\mathbb{F}_2$-variables. We have also tested it with overdefined systems, for which Gröbner-basis algorithms are known to work better. While it does not run out of memory, the results are not satisfying: 2.5 hours to solve 20 cubic equations in 20 variables, half an hour for 45 quadratic equations in 30 variables, and 7 minutes for 60 quadratic equations in 30 variables on one 2.2-GHz Opteron core. Some very recent improvements on Gröbner-basis solvers have reported speed-up over MAGMA $\mathbf{F_4}$ of two- to five-fold [11]. However, even with such significant improvements, Gröbner-basis solvers do not seem to be able to compete with exhaustive search algorithms in this range, as each of the above is solved in a split second using negligible amount of memory on the same CPU by the latter.

Table 1. Performance results for $n = 48$ and projected budgets for solving $n = 64$ in one month

Time (minutes)			Testing platform				#cores	est. cost
$d = 2$	$d = 3$	$d = 4$	GHz	Arch.	Name	USD	(#used)	(USD)
1217	2686	3191	2.2	K10	Phenom 9550	120	4(1)	54,000
1157	1992	2685	2.3	K10+	Opteron 2376	184	4(1)	113,316
142	240	336	2.3	K10+	Opteron 2376×2	368	8(8)	
780	1364	1819	2.4	C2	Xeon X3220	210	4(1)	60,720
671	1176	1560	2.83	C2+	Core2 Q9550	225	4(1)	55,575
179	294	390	2.83	C2+	Core2 Q9550	225	4(4)	
761	1279	1856	2.26	Ci7	Xeon E5520	385	4(1)	78,720
95	154	225	2.26	Ci7	Xeon E5520×2	770	8(8)	
41	73	271	1.3	G200	GTX 280	n/a	240	n/a
21	36	126	1.25	G200	GTX 295	500	480	15,500

Implications. The new exhaustive search algorithm can be used as a black box in cryptanalysis that needs to solve quadratic equations. This includes, for instance, several algorithms for the Isomorphism of Polynomials problem [8, 24], as well as attacks that rely on such algorithms, e.g., [9].

We also show with a concrete example that (relatively simple) computations requiring 2^{64} operations can be easily carried out in practice with readily available hardware and a modest budget. Lastly, we highlight the fact that GPUs have been used successfully by the cryptographic community to obtain very efficient implementations of combinatorial algorithms or cryptanalytic attacks, in addition to the more numeric-flavored cryptanalysis algorithm demonstrated by the implementation of the ECM factorization algorithm on GPUs [6].

Organization of the Paper. Section 2 establishes a formal framework of exhaustive search algorithms including useful results on Gray Codes and derivatives of multivariate polynomials. Known exhaustive search algorithms are reviewed in Section 3. Our algorithm to find the zeroes of a single polynomial of any degree is given in Section 4, and it is extended to find the common zeroes of a collection of polynomials in Section 5. Section 6 describes the two platforms on which we implemented the algorithm, and Section 8 describes the implementation and performance evaluation results.

2 Generalities

In this paper, we will mostly be working over the finite vector space $(\mathbb{F}_2)^n$. The canonical basis is denoted by $(e_0, \ldots, e_{n-1})$. We use $\oplus$ to denote addition in $(\mathbb{F}_2)^n$, and $+$ to denote integer addition. We use $i \ll k$ (resp. $i \gg k$) to denote binary left-shift (resp. right shift) of the integer i by k bits.

Gray Code. Gray Codes play a crucial role in this paper. Let us denote by $b_k(i)$ the index of the k-th lowest-significant bit set to 1, or -1 if the hamming weight of i is less than k. For example, $b_k(0) = -1$, $b_1(1) = 0, b_1(2) = 1$ and $b_2(3) = 1$.

Definition 1. $\textsc{GrayCode}(i) = i \oplus (i \gg 1)$.

Lemma 1. *For $i \in \mathbb{N}$:* $\textsc{GrayCode}(i+1) = \textsc{GrayCode}(i) \oplus e_{b_1(i+1)}$.

Lemma 2. *For $j \in \mathbb{N}$:*

$$\textsc{GrayCode}\left(2^k + j \cdot 2^{k+1}\right) = \begin{cases} \textsc{GrayCode}\left(2^k\right) \oplus (\textsc{GrayCode}(j) \ll (k+1)) & \textit{if } j \textit{ is even} \\ \textsc{GrayCode}\left(2^k\right) \oplus (\textsc{GrayCode}(j) \ll (k+1)) \oplus e_k & \textit{if } j \textit{ is odd.} \end{cases}$$

Proof. It should be clear that $2^k + j \cdot 2^{k+1}$ and $2^k \oplus j \cdot 2^{k+1}$ in fact denote the same number. Also, GRAYCODE is a linear function on $(\mathbb{F}_2)^n$. Thus it remains to establish that $\text{GRAYCODE}(j \cdot 2^{k+1}) = \text{GRAYCODE}(j) \ll k+1$ (resp. $e_k \oplus (\text{GRAYCODE}(j) \ll k+1)$) when j is even (resp. odd). Again, $j \cdot 2^{k+1} = j \ll (k+1)$, and by definition we have:

$$\text{GRAYCODE}(j \cdot 2^{k+1}) = \text{GRAYCODE}(j \ll (k+1)) = (j \ll (k+1)) \oplus ((j \ll (k+1)) \gg 1)$$

Now, we have :

$$(j \ll k+1) \gg 1 = \begin{cases} (j \gg 1) \ll k+1 & \text{when } j \text{ is even} \\ ((j \gg 1) \ll k+1) \oplus e_k & \text{when } j \text{ is odd} \end{cases}$$

and the result follows. □

Derivatives. Define the $\mathbb{F}_2$ *derivative* $\frac{\partial f}{\partial i}$ of a polynomial with respect to its i-th variable as $\frac{\partial f}{\partial i} : \mathbf{x} \mapsto f(\mathbf{x} + e_i) + f(\mathbf{x})$. Then for any vector $\mathbf{x}$, we have:

$$f(\mathbf{x} \oplus e_i) = f(\mathbf{x}) \oplus \frac{\partial f}{\partial i}(\mathbf{x}) \tag{1}$$

If f is of total degree d, then $\frac{\partial f}{\partial i}$ is a polynomial of degree $d-1$. In particular, if f is quadratic, then $\frac{\partial f}{\partial i}$ is an affine function. In this case, it is easy to isolate the constant part (which is a constant in $\mathbb{F}_2$) : $c_i = \frac{\partial f}{\partial i}(0) = f(e_i) \oplus f(0)$. Then, the function $\mathbf{x} \mapsto \frac{\partial f}{\partial i}(\mathbf{x}) \oplus c_i$ is by definition a linear form and can be represented by a vector $D_i \in (\mathbb{F}_2)^n$. More precisely, we have $D_i[j] = f(e_i \oplus e_j) \oplus f(e_i) \oplus f(e_j) \oplus f(0)$. Then equation (1) becomes:

$$f(\mathbf{x} \oplus e_i) = f(\mathbf{x}) \oplus D_i \cdot \mathbf{x} \oplus c_i \tag{2}$$

Enumeration Algorithms. We are interested in *enumeration algorithms*, *i.e.*, algorithms that evaluate a polynomial f over all the points of $(\mathbb{F}_2)^n$ to find its zeroes. Such an enumeration algorithm is composed of two functions: INIT and NEXT. $\text{INIT}(f, x_0, k_0)$ returns a *State* containing all the information the enumeration algorithm needs for the remaining operations. The resulting State is configured for the evaluation of f over $x_0 \oplus (\text{GRAYCODE}(i) \ll k_0)$, for increasing values of i. $\text{NEXT}(State)$ advance to the next value and updates $State$. Three values can be directly read from the state: $State.\mathbf{x}$, $State.\mathbf{y}$ and $State.i$. These are linked at all times by the following three invariants:

i) $State.\mathbf{y} = f(State.\mathbf{x})$
ii) $State.\mathbf{x} = x_0 \oplus (\text{GRAYCODE}(State.i) \ll k_0)$.
iii) $\text{NEXT}(State).i = State.i + 1$.

Finding all the zeroes of f is then achieved with the loop shown in fig. 1.

3 Known Techniques for Quadratic Polynomials

We briefly discuss the enumeration techniques known to the authors.

```
1: procedure ZEROES(f)
2:     State ← INIT(f, 0, 0)
3:     for i from 0 to 2^n − 1
4:         if State.y = 0 then State.x is a zero of f
5:         NEXT(State)
6:     end for
7: end procedure
```

Fig. 1: Main loop common to all enumeration algorithms.

Naive Evaluation. The simplest way to implement an enumeration algorithm is to evaluate the polynomial f from scratch at each point of $(\mathbb{F}_2)^n$. If f is of degree d, this requires $(d-1)$ AND per monomial, and nearly one XOR per monomial. Since the evaluation takes place many times for the same f with different values of the variables, we will usually assume that the polynomial can be *hard-coded*, and that multiplication of a monomial by its coefficient come for free. Each call to NEXT would then require at most $d \cdot \binom{n}{d}$ bit operations, $1/d$ of which being XORs and the rest being ANDs (not counting the cost of enumerating $(\mathbb{F}_2)^n$, *i.e.*, incrementing a counter). This can be improved a bit, using what is essentially a multivariate Hörner evaluation technique. If f is quadratic, it can be written:

$$f(\mathbf{x}) = c \oplus \sum_{i=0}^{n-1} \mathbf{x}_i \cdot \left(c_j \oplus \sum_{j=i+1}^{n-1} a_{ij} \cdot \mathbf{x}_j \right) \tag{3}$$

If f is cubic, it can be written:

$$f(\mathbf{x}) = c \oplus \sum_{i=0}^{n-1} \mathbf{x}_i \cdot \left(c_j \oplus \sum_{j=i+1}^{n-1} \mathbf{x}_j \cdot \left(c_{ij} \oplus \sum_{k=j+1}^{n-1} a_{ijk} \cdot \mathbf{x}_k \right) \right)$$

And so on and so forth. The required numbers of operations in this representation is given by:

$$N_{AND} = \sum_{k=1}^{d-1} \binom{n}{k} \qquad N_{XOR} = \sum_{k=1}^{d} \binom{n}{k}$$

This method is not without its advantages, chiefly (a) insensitivity to the order in which the points of $(\mathbb{F}_2)^n$ are enumerated, and (b) we can bit-slice and get a speed up of nearly ω, where ω is the maximum width of the CPU logical instructions.

The Folklore Differential Technique. It was pointed out in Section. 2 that once $f(\mathbf{x})$ is known, computing $f(\mathbf{x} \oplus e_i)$ amounts to compute $\frac{\partial f}{\partial i}(\mathbf{x})$. If f is quadratic, and in this case only, this derivative happens to be a linear function which can be efficiently evaluated by computing a vector-vector product and a few scalar additions. This strongly suggests to evaluate f on $(\mathbb{F}_2)^n$ using a *Gray Code*, *i.e.*, an ordering of the elements of $(\mathbb{F}_2)^n$ such that two consecutive elements differ in only one bit (see lemma 1). This leads to the algorithm shown in fig. 2.

We believe this technique to be folklore, and in any case it appears more or less explicitly in the existing literature. Each call to NEXT requires n ANDs, as well as $n+2$ XORs, which makes a total bit operation count of $2(n+1)$. This is about $n/4$ times less than the naive method applied to a quadratic f. Note that when we describe an enumeration algorithm, the variables that appear inside NEXT are in fact implicit functions of $State$. The dependency has been removed to lighten the notational burden.

```
1: function INIT(f, k_0, x_0)
2:     i ← 0
3:     x ← x_0
4:     y ← f(x_0)
5:     For all 0 ≤ k ≤ n − 1,
           initialize c_k and D_k
6: end function
```

```
1: function NEXT(State)
2:     i ← i + 1
3:     k = b_1(i)
4:     z ← VECTORVECTORPRODUCT(D_k, x) ⊕ c_k
5:     y ← y ⊕ z
6:     x ← x ⊕ e_{k+k_0}
7: end function
```

Fig. 2: The Folklore differential algorithm.

4 A Faster Recursive Algorithm for any Degree

We now describe one of the main contributions of this paper, a new algorithm which is both asymptotically and practically faster than other known exhaustive search techniques in evaluating a polynomial of any degree on all the points of $(\mathbb{F}_2)^n$.

4.1 Intuition

In the folklore differential algorithm of fig. 2, the dominating part is the scalar product computed in line 4 of NEXT. It would be great if it were possible to exploit the fact that $\mathbf{x}$ is only slightly changed between to calls to NEXT. The problem is that k (defined on line 3) is never the same in two consecutive iterations. Now assume we modify the function this way:

```
1: function NEXT(State)
2:     i ← i + 1
3:     k = b_1(i)
4:     z[k] ← VECTORVECTORPRODUCT(D_k, x) ⊕ c_k
5:     x[k] ← x
6:     y ← y ⊕ z[k]
8:     x ← x ⊕ e_k
9: end function
```

Then, on line 4, the *previous* value of $\mathbf{z}[k]$, when it exists, is still available, and this value is the scalar product of D_k with $\mathbf{x}[k]$ (which is the previous value of $\mathbf{x}$ for the same value of k). Thus, the new value of $\mathbf{z}[k]$ is going to be $\mathbf{z}[k] \oplus D_k \cdot (\mathbf{x} \oplus \mathbf{x}[k])$. The key observation is proposition 1 below, as its consequence is that the computation of the scalar product can be done in constant time, with two ANDs and one XOR.

Proposition 1. *At the beginning of the function,* $\mathbf{x}^\top \oplus \mathbf{x}[k]^\top$ *has a hamming weight upper-bounded by two.*

Proof. Indeed, $\mathbf{x}[k_0]^\top$ is only accessed and modified when $b_1\left(i^\top + 1\right) = k_0$, for any given k_0. The integers u such that $b_1(u) = k_0$ are precisely the integers written $u = 2^{k_0} + j \cdot 2^{k_0+1}$, for $j \geq 0$. Then, if we consider the values of the variables at the beginning of the function, by invariant ii, we have for some j:

$$\mathbf{x}^\top = \text{GRAYCODE}\left(2^k + (j+1) \cdot 2^{k+1}\right)$$
$$\mathbf{x}[k]^\top = \text{GRAYCODE}\left(2^k + j \cdot 2^{k+1}\right)$$

Thus, it follows from lemma 2 that just before line 1 is executed, we have:

$$\begin{aligned}\mathbf{x}^\top \oplus \mathbf{x}[k]^\top &= e_k \oplus (\text{GRAYCODE}(j) \ll (k+1)) \oplus (\text{GRAYCODE}(j+1) \ll (k+1)) \\ &= e_k \oplus ((\text{GRAYCODE}(j) \oplus \text{GRAYCODE}(j+1)) \ll (k+1))\end{aligned}$$

and by lemma 1,

$$\mathbf{x}^\top \oplus \mathbf{x}[k]^\top = e_k \oplus e_{k+1+b_1(j+1)} \tag{4}$$

□

By looking closely at the proof of proposition 1, we can write an *optimized differential algorithm*. However, before that, a few details still need to be addressed.

- The first time that $b_1(i) = k$, then $\mathbf{z}[k]$ is not defined. In this case, we in fact know that $i = 2^k$. Therefore, special care must be taken to initialize $\mathbf{z}[k]$ when $b_2(i) = -1$, which is equivalent to saying that the hamming weight of i is less than two. In that case, by invariant ii, we have:

$$\mathbf{x} = \begin{cases} e_0 & \text{if } i = 1 \\ e_k \oplus e_{k-1} & \text{if } i = 2^k \text{ and } k > 0 \end{cases}$$

- Also note that with the notation $k_1 = b_1(i)$ and $k_2 = b_2(i)$, then if $b_2(i) \neq -1$, equation (4) becomes:

$$\mathbf{x} \oplus \mathbf{x}[k] = e_{k_1} \oplus e_{k_2}$$

And thus,

$$\text{VECTORVECTORPRODUCT}(D_{k_1}, \mathbf{x} \oplus \mathbf{x}[k_1]) = D_{k_1}[k_1] \oplus D_{k_1}[k_2]$$

This last formula can be further simplified by observing that $D_{k_1}[k_1] = 0$.

All these considerations lead to the algorithm shown in fig. 3. Note that the conditional statement could be removed by unrolling the loop carefully. The critical part of the algorithm is therefore an extremely reduced section of the code, that performs two XORs, increment a counter, and evaluate b_1 as well as b_2. The cost of maintaining i, k_1 and k_2 can again be reduced greatly by unrolling the loop.

```
1:  function INIT(f, k_0, x_0)
2:      i ← 0
3:      x ← x_0
4:      y ← f(x_0)
5:      For all 0 ≤ k ≤ n − 1,
6:          initialize c_k and D_k
7:      End for
8:      z[0] ← c_0
9:      For all 1 ≤ k ≤ n − 1,
10:         z[k] ← D_k[k − 1]
11:     End for
12: end function
```

```
1:  function NEXT(State)
2:      i ← i + 1
3:      k_1 = b_1(i)
4:      k_2 = b_2(i)
5:      if k_2 ≠ −1 then
6:          z[k_1] ← z[k_1] ⊕ D_{k_1}[k_2]
7:      end if
8:      y ← y ⊕ z[k_1]
9:      x ← x ⊕ e_{k_0+k_1}
10: end function
```

Fig. 3: An optimized differential enumeration algorithm for quadratic forms.

4.2 Recursive Generalization to Any Degree.

It is in fact possible to generalize the improvement of the folklore differential algorithm that lead to the optimized differential algorithm in the quadratic case. The core idea is that in this algorithm, a given derivative is evaluated on the consecutive points of something that looks very much like a Gray code. This suggest using the technique recursively.

To make this thing explicit, we introduce a new State for each of the derivatives of f used in the enumeration of f. Instead of storing $\mathbf{x}[k]$ and $\mathbf{z}[k]$, we will access $Derivative[k].\mathbf{y}$ and $Derivative[k].\mathbf{y}$. Also, $Derivative[k].i$ will count the number of times $b_1(k)$ happened. We now reformulate our optimized algorithm in this framework. However, know, the x_0 and k_0 parameters appearing in invariant ii will play a more important role.

Terminal case when f is of degree 0

```
1: function INIT(f, k_0, x_0)
2:     i ← 0
3:     x ← x_0
4:     y ← f(x_0)
5: end function
```

```
1: function NEXT(State)
2:     i ← i + 1
3:     k = b_1(i)
4:     x ← x ⊕ e_{k+k_0}
5: end function
```

Recursive case when $\deg f > 0$.

```
1: function INIT(f, k_0, x_0)
2:     i ← 0
3:     x ← x_0
4:     y ← f(x_0)
7:     Derivative[0] ← INIT(∂f/∂k_0, k_0 + 1, x_0)
6:     for k from 1 to n − k_0 − 1
7:         Derivative[k] ← INIT(∂f/∂k + k_0, k + k_0 + 1, x_0 ⊕ e_{k_0+k−1})
8:     end for
9: end function
```

```
1: function NEXT(State)
2:     i ← i + 1
3:     k = b_1(i)
4:     x ← x ⊕ e_{k+k_0}
5:     y ← y ⊕ Derivative[k].y
6:     NEXT(Derivative[k])
7: end function
```

Fig. 4: The recursive differential for all degrees.

A correctness proof is given in the full version.

4.3 Time and Space Complexity Considerations

It should be clear from the description of NEXT that it has complexity $\mathcal{O}(d)$. Therefore, the complexity of enumerating all the values of f on $(\mathbb{F}_2)^n$ can be done with complexity $\mathcal{O}(d \cdot 2^n)$. What is the space requirement of the algorithm? The answer to this question is twofold: there is an *internal state* that gets modified by the algorithm, and that correspond to the $\mathbf{y}$ field of all the non-constant derivatives. There is also an array of *constants*, which is only read from the memory, and that correspond to the $\mathbf{y}$ field of degree-d derivatives.

INIT stores one bit per degree-d derivative $\partial f/\partial i_1 \partial i_2 \dots \partial i_d$, with $1 \leq i_1 < i_2 < \cdots < i_d \leq n$. The number of such tuples $(i_1, i_2, \dots, i_d)$ is known to be $\binom{n}{d-1}$. This yields the following result:

Proposition 2. *The algorithm allocates* $\sum_{i=0}^{d-1} \binom{n}{i}$ *bits of internal state and* $\binom{n}{d}$ *bits of constants*

4.4 An iterative Version

In section 4.2, we gave a recursive algorithm that works for all degree, which is a generalized version of the iterative algorithm described only in the quadratic case in section 4.1. Indeed, one could check

that unrolling the algorithm of fig. 4 with a quadratic f gives back the algorithm of fig. 3. We now move on to write an iterative version of the general recursive algorithm. This iterative version allows more optimization, such as the removal of extra useless work, and a more careful parallel scheduling.

But first, the function NEXT_2 shown in fig. 5 does exactly the same thing as NEXT, but in a slightly different way. Instead of calling NEXT at the end, it calls it at the beginning, except the first time a given value of k is reached, to avoid calling it an extra time at the begining.

```
1: function NEXT2(State)
2:     i ← i + 1
3:     k = b1(i)
4:     if i ≠ 2^k then
5:         NEXT2 (Derivative [k])
6:     end if
7:     x ← x ⊕ e_{k+k0}
8:     y ← y ⊕ Derivative[k].y
9: end function
```

Fig. 5: An equivalent version of NEXT.

We can therefore work on NEXT_2. A first remark is that maintaining $\mathbf{x}$ is required by the invariants, but is otherwise useless for the actual computation. A first step is to completely remove $\mathbf{x}$ from the algorithm. Less obviously, we can also avoid maintaining i. To see that, we first need to state an equivalent of lemma **??** adapted to NEXT_2, the proof of which is left to the reader.

Lemma 3. *After k is updated on line 3 of NEXT_2, we have:*

$$i^\top + 1 = 2^k + (Derivative[k].i + 1) \times 2^{k+1}.$$

It is an easy consequence of lemma 3 that in NEXT_2, after k is updated on line 3, we have for any j:

$$b_j(Derivative[k].i + 1) = b_{j+1}(i^\top + 1).$$

Thus, it is possible to avoid storing the i values, except in the main loop, and to re-generate online by evaluating b_j on the index of the main loop. These computations, although taking amortized constant time, can be made negligible by unrolling. To ease notation, we introduce the following shorthand:

$$D[k_1, k_2, \ldots, k_\ell] \stackrel{\Delta}{=} State.Derivative[k_1].Derivative[k_2] \ldots .Derivative[k_\ell].\mathbf{y}$$

With this notation, the algorithm of fig. 6 is just an unrolled version of the recursive algorithm of fig. 4 in which all the useless operations have been removed.

5 Enumeration of Several Multivariate Polynomials Simultaneously

In the previous section, we discussed how to enumerate one single polynomial. We now move on to the enumeration of several polynomial simultaneously.

We will use several time the following simple idea: all the techniques we discussed above perform a sequence of operations that is independent of the coefficients of the polynomials. Therefore, m instances of (say) the algorithm of fig. 6 could be run in parallel on $f_1, \ldots, f_m$. All the parallel runs would execute

```
1:  procedure ZEROES(f)
2:    State ← INIT(f, 0, 0)
3:    for i from 0 to 2^n − 1
4:      if State.y = 0 then GRAYCODE(i) is a zero of f
5:      k_1 = b_1(i + 1)
6:      k_2 = b_2(i + 1)
7:      ...
8:      k_d = b_d(i + 1)
9:      if k_d > −1 then D[k_1, ..., k_{d−1}] ← D[k_1, ..., k_{d−1}] ⊕ D[k_1, ..., k_{d−1}, k_d]
10:     ...
11:     if k_3 > −1 then D[k_1, k_2] ← D[k_1, k_2] ⊕ D[k_1, k_2, k_3]
12:     if k_2 > −1 then D[k_1] ← D[k_1] ⊕ D[k_1, k_2]
13:     y ← y ⊕ D[k_1]
14:   end for
15: end procedure
```

Fig. 6: Iterative algorithm for all degrees.

the same instruction on different data, making it efficient to implement on vector or SIMD architectures. In each iteration of the main loop, it is easy to check if *all* the polynomials vanished on the current point of $(\mathbb{F}_2)^n$. Evaluating all the m polynomials in parallel using the algorithm of fig. 6 would require roughly $m \cdot d \cdot 2^n$ bit operations. The point of this section is that it is possible to do much better than this.

Let us first introduce a useful notation. Given an ordered set U, we denote the common zeroes of $f_1, \ldots, f_m$ belonging to U by $Z([f_1, \ldots, f_m], U)$. Let us also denote $Z_0 = (\mathbb{F}_2)^n$, and $Z_i = Z([f_i], Z_{i-1})$. It should be clear that $Z = Z_m$ is the set of common zeroes of the polynomials, and therefore the object we wish to obtain.

5.1 General Technique: Splitting the Problem

A possible strategy is to compute the Z_i recursively: first Z_1, then Z_2, etc. However, while the algorithms of section 4 can be used to compute Z_1, they cannot be used to compute Z_2 from Z_1, because they intrinsically enumerate all $(\mathbb{F}_2)^n$. In practice, the best results are in fact obtained by computing Z_k, for some well-chosen value of k, using k parallel runs of the algorithm of fig. 6, and then computing Z_m using a *secondary algorithm*. Computing Z_k requires $d \cdot k \cdot 2^n$ bit operations. It then remains to compute Z_m from Z_k, and to find the best possible value of k.

Note that if $m > n$, we can focus on the first n equations, since a system of n randomly chosen multivariate polynomial equations in n variables of constant degree d is expected to have a constant number of solutions, which can in turn be checked against the remaining equations efficiently. If $m < n$, then we can specialize $m - n$ variables, and solve the m equations in m variables for any possible values of the specialized variables. All-in-all, the interesting case is when $m = n$.

Also note that it makes sense to choose k according to the targeted hardware platform (*e.g.*, $k = 32$ if only 32-bit registers are available), it is an interesting theoretical problem choose k in order to minimize the global number of bit operations.

We now move on to discuss several secondary algorithms to compute Z_m from Z_k, and discuss their relative merits.

5.2 Naive Secondary Evaluation

We compute Z_{i+1} from Z_i using naive evaluation, for $k \leq i \leq n - 1$. It is clear that the expected cardinality of Z_i for random polynomial equations is 2^{n-i}. We will assume for the sake of simplicity that

evaluating a degree-d polynomial requires $\binom{n}{d}$, following the reasoning in section 3. Computing Z_{i+1} then takes about $\binom{n}{d} \cdot 2^{n-i}$ bit ops. The expected cost of computing Z is then approximately:

$$\sum_{i=k}^{n} \binom{n}{d} \cdot 2^{n-i} \approx \binom{n}{d} \cdot 2^{n-k+1} \text{ bit operations.}$$

Minimizing the global cost means solving the equation:

$$k \cdot d \cdot 2^n = \binom{n}{d} \cdot 2^{n-k+1}.$$

which is easily seen to be equivalent to:

$$(k \cdot \ln 2) \cdot \exp(k \cdot \ln 2) = 2 \cdot \binom{n}{d} \cdot \frac{\ln 2}{d}$$

Now, the Lambert W function is such that $W(x) \cdot \exp(W(x)) = x$. Thus, the solution of our equation is:

$$k = W\left(\binom{n}{d} \cdot \frac{2 \cdot \ln 2}{d}\right) / \ln 2$$

Using the known fact [15] that when x goes to infinity:

$$W(x) = \ln x - \ln \ln x + o(\ln \ln x)$$

we find that when $n \to \infty$:

$$k = 1 + \log_2\left(\binom{n}{d} \cdot \frac{1}{d}\right) + \mathcal{O}\left(\ln \ln n\right)$$

The full cost of the algorithm is then approximately $d^2 \cdot \log_2 n \cdot 2^{n+1}$ bit operations..

5.3 Differential Secondary Evaluation

We only describe the quadratic case, but this could be extended to higher degrees. We can efficiently evaluate Z_{i+1} from Z_i using an easy consequence of equation (1): given $f(\mathbf{x})$, computing $f(\mathbf{x} + \Delta)$ takes $2|\Delta| \cdot n$ bit operations, where $|\Delta|$ denote the hamming weight of Δ, by computing $|\Delta|$ vector-vector products with the derivatives. Let us order the elements of Z_i by writing: $Z_i = \left\{\mathbf{x}_1^i, \ldots \mathbf{x}_{q_i}^i\right\}$ (the elements are ordered using the usual lexicographic order), and $\Delta_j^i = \mathbf{x}_{j+1}^i \oplus \mathbf{x}_j^i$.

Computing Z_{i+1} therefore requires approximately:

$$2n \cdot \sum_{j=1}^{q_i - 1} |\Delta_j^i| \text{ bit operations.}$$

Now, let us consider the Δ_j^i as integer number between 0 and $2^n - 1$. The x_{j+1}^i are the zeroes of a set of i random polynomials, and under the assumption that each point of $(\mathbb{F}_2)^n$ has one chance over 2^i to be such a zero, then the difference Δ_j^i between two such consecutive zeros follows a geometric distribution of parameter 2^{-i}, and thus has expectation 2^i. The hamming weight $|\Delta_j^i|$ is upper-bounded by $\lceil \log_2 \Delta_j^i \rceil$ (considered as an integer), and therefore $|\Delta_j^i|$ has expectation less than i.

Computing Z_{i+1} therefore requires in average $2n \cdot i \cdot 2^{n-i}$ bit op. Finally, computing Z from Z_k requires on average:

$$2n \cdot \sum_{i=k}^{n-1} i \cdot 2^{n-i} \leq 4n \cdot (k+1) \cdot 2^{n-k} \text{ bit operations}$$

An approximately optimal value of k would then satisfy

$$2k \cdot 2^n = 4n \cdot (k+1) \cdot 2^{n-k}$$

which is approximately $k = 1 + \log_2 n$. The complexity of the whole procedure is then $4 \log_2 n \cdot 2^n$. However, implementing this technique efficiently looks like a lot of work for at best a $2\times$ gain.

6 A Brief Description of the Hardware Platforms

6.1 Vector Units on x86-64

The most prevalent SIMD (single instruction, multiple data) instruction set today is SSE2, available on all current Intel-compatible CPUs. SSE2 instructions operate on 16 architectural `xmm` registers, each of which is 128-bit wide. We use integer operations, which treat `xmm` registers as vectors of 8-, 16-, 32- or 64-bit operands.

The highly non-orthogonal SSE instruction set includes Loads and Stores (to/from `xmm` registers, memory — both aligned and unaligned, and traditional registers), Packing/Unpacking/Shuffling, Logical Operations (`AND, OR, NOT, XOR`, Shifts Left, Right Logical and Arithmetic — bit-wise on units and byte-wise on the entire `xmm` register), and Arithmetic (add, substract, multiply, max-min) with some or all of the arithmetic widths. The interested reader is referred to Intel and AMD's manuals for details on these instructions, and to references such as [19] for throughput and latencies.

6.2 G2xx-series Graphics Processing Units from NVIDIA

We choose NVIDIA's G2xx GPUs as they have the least hostile GPU parallel programming environment called CUDA (Compute Unified Device Architecture). In CUDA, we program in the familiar C/C++ programming language plus a small set of GPU extensions.

An NVIDIA GPU contains anywhere from 2–30 streaming multiprocessors (MPs). There are 8 ALUs (streaming processors or SPs in market-speak) and two super function units (SFUs) on each MP. A top-end "GTX 295" card has two GPUs, each with 30 MPs, hence the claimed "480 cores". The theoretical throughput of each SP per cycle is one 32-bit integer or floating-point instruction (including add/subtract, multiply, bitwise AND/OR/XOR, and fused multiply-add), and that of an SFU 2 floating-point multiplications or 1 special operation. The arithmetic units have 20+-stage pipelines.

Main memory is slow and forms a major bottleneck in many applications. The read bandwidth from memory on the card to the GPU is only one 32-bit read per cycle per MP and has a latency of > 200 cycles. To ease this problem, the MP has a register file of 64 KB (16,384 registers, max. of 128 per thread), a 16-bank shared memory of 16 KB, and an 8-KB cache for read-only access to a declared "constant region" of at most 64 KB. Every cycle, each MP can achieve one read from the constant cache, *which can broadcast to many thread at once*.

Each MP contains a scheduling and dispatching unit that can handle a large number of lightweight threads. However, the decoding unit can only decode once every 4 cycles. *This is typically 1 instruction, but certain common instructions are "half-sized", so two such instructions can be issued together if independent.* Since there are 8 SPs in an MP, CUDA programming is always on a Single Program Multiple Data basis, where a "warp" of threads (32) should be executing the same instruction. If there is a branch which is taken by some thread in a warp but not others, we are said to have a "divergent" warp; from then on only part of the threads will execute until all threads in that warp are executing the same instruction again. Further, as the latency of a typical instruction is about 24 cycles, NVIDIA recommends a minimum of 6 warps on each MP, although it is sometimes possible to get acceptable performance with 4 warps.

7 Parallelization and Memory Bandwith Issues

The critical loop of the algorithm is very short, since it performs only d logical operations. However, it accesses the memory $d+1$ times, which suggests that the memory bandwith will be the actual performance bottleneck. We address this issue in two complementary ways. First we argue that the algorithm is *cache-oblivious* [20], *i.e.*, that is uses the cache efficiently regardless of its size. Then we argue that on massively concurrent architectures such as GPUs, then any word read from the memory can be broadcast to all the concurrently running threads almost systematically.

7.1 Spatial and Temporal Proximity on Iterative Architectures

We will study the behavior of the algorithm in the *Ideal Cache Model*. This model consists of a computer with a two-level memory hierarchy consisting of an ideal (data) cache of Z words and an arbitrarily large main memory. The cache is partitioned into cache lines, each consisting of L consecutive words that are always moved together between cache and main memory. The processor can only reference words that reside in the cache. If the referenced word belongs to a line already in cache, a *cache hit* occurs, and the word is delivered to the processor. Otherwise, a *cache miss* occurs, and the line is fetched into the cache. The ideal cache is *fully associative*: cache lines can be stored anywhere in the cache. If the cache is full, a cache line must be evicted. The ideal cache uses the optimal off-line strategy of replacing the cache line whose next access is farthest in the future, and thus it exploits temporal locality perfectly. An algorithm with an input of size n is measured in the ideal-cache model in terms of the usual number of operation performed by the processor, but also in terms of its *Cache Complexity* $Q(n, Z, L)$ – the number of cache misses it incurs as a function of Z and L. We now move on to evaluate the cache complexity of the enumeration algorithm, as show in fig. 6. We will assume that the "memory cells" accessed by the algorithm have the same size as the a word in the cache (if this were not the case, it would only incur a constant multiplicative loss, and we are mostly interested in an asymptotic result).

The memory words accessed in the algorithm belong to arrays of various dimension, and are accessed with indices of variable length. It should be clear from the description of the algorithm that for all $k \leq d$, the memory location of index $[i_1, \ldots, i_k]$ is accessed at step s if $b_j(s) = i_j$, for all $j \leq k$. This memory access pattern is in fact very regular. We say that a memory word is accessed with period T if, when it is accessed at iteration i, it is also accessed at iteration $i + T$, but not in-between.

Lemma 4. *For all $k \leq d$, the memory location of index $[i_1, \ldots, i_k]$ is accessed with period 2^{i_k+1}.*

Proof. We associate with an index $[i_1, \ldots, i_k]$ the set $\Omega_{i_1,\ldots,i_k}$ of integers n such that $b_1(n) = i_1, \ldots, b_k(n) = i_k$. The problem amounts to show that the difference between two consecutive elements of this set is 2^{i_k+1}. But it is easily seen that if $n \in \Omega_{i_1,\ldots,i_k}$, then $n + j \cdot 2^{i_k+1} \in \Omega_{i_1,\ldots,i_k}$ for any positive integer j. This follows from the fact that $b_j(n) = b_j(n + 2^\ell)$ if $\ell > j$, and establishes the result. □

It should be clear that all the memory location accessed with period exactly T are accessed in the first T iteration of the main loop. Moreover, they are accessed in a certain order. For instance, memory words with period 8 are accessed in this order in the first 8 iterations: $[2], [0, 2], [1, 2]$. By definition of the period, this access pattern is reproduced without modifications in the next T iterations. Thus, memory words with period T are accessed in a *cyclic* fashion.

The algorithm easily defines a total order relation on the memory locations it accesses: $x \leq y$ if and only if the first access to x takes places *before* the first access to y. Let us assume that the actual memory addresses are compatible with this order relation. Then, more frequently accessed words are stored with the lowest addresses, and words with the same access frequency are stored contiguously in memory. There are $\sum_{i=0}^{\min(d-1,k)} \binom{k}{i}$ memory locations that are accessed with period 2^{k+1}.

This being said, we will focus on the case where all the $\sum_{i=0}^{d} \binom{n}{i}$ memory words accessed by the algorithm do not fit into the cache, to avoid studying the trivial case. Let us define the *critical period* 2^{T_c+1} to be the biggest integer such that all the memory words accessed with period 2^{T_c+1} fit in the cache:

$$\sum_{k=0}^{T_c} \sum_{i=0}^{\min(d-1,k)} \binom{k}{i} \leq Z - 1$$

Under the (mild) assumption that the cache contains $Z \geq 2^d$ words, and thus that T_c is greater than d, this condition becomes:

$$2^d - 1 + \sum_{k=d}^{T_c} \sum_{i=0}^{d-1} \binom{k}{i} \leq Z - 1$$

This this is the summation in a rectangle inside Pascal's triangle, then by applying Pascal's rule recursively, we may simplify this expression, and find that it is equivalent to:

$$\sum_{i=0}^{d} \binom{T_c + 1}{i} \leq Z$$

The important point is that all memory words with period 2^{T_c+1} fit in the cache and *do not leave it*. This fact is almost true by definition of T_c: the optimal off-line cache strategy will not evict a cache line that will be accessed in T steps if it can evict a cache line that will only be accessed in $2T$ steps. And there will always be a cache line not containing a word accessed with period 2^{T_c+1} or less. This being said, we can state our result:

Proposition 3. *Under the assumption that $T_c \geq 2d$, the following two inequalities hold:*

i) $Q(n, d, Z, L) \leq 2^{n-2-T_c} \cdot (d+1) \cdot \binom{T_c + 1}{d - 1}$

ii) $Q(n, d, Z, L) \leq 2^{n-2-T_c} \cdot \dfrac{d \cdot (d+1)}{T_c + 2 - d} \cdot Z$

The proof is in the full version of the paper.

Let us consider a polynomial in 64 variables. If we assume an incredibly small cache of $Z = 2^{10}$ bits and that our polynomial is of degree 2, then $T_c = 44$ and the enumeration will make about 2^{25} cache misses, for a running time of at least 2^{65}. If we assume that our polynomial is of degree 4, and that the cache is 2^{14}-bit large, then $T_c = 24$, and there will be 2^{52} cache misses, for more than 2^{66} memory accesses.

7.2 Constrained Small Memory Chips on Concurrent Architectures

The problem is formulated in very different terms on some parallel architectures, such as GPUs, or the Cell, in which the available "fast" memory is fairly restricted, and main memory is relatively slow.

Parallelizing the process is very easy, as it simply comes down to partition the search space into the number of available cores. An interesting side effect is that when done properly, this partition reduces the amount of data that needs to be transfered from the main memory. We will now assume that we have 32-bit registers, and we will use 32 parallel copies of the algorithm of fig. 6, to enumerate 32 polynomials simultaneously.

For instance, the loop of fig. 4 can be split in independent chunks, as illustrated by fig. 7. An additional benefit is that processing one such chunk only require access to a fraction of the memory used by the full enumeration. In fact, $b_1(i+1)$ is greater or equal than L if the L rightmost bits of $(i+1)$ are zero, or,

in other terms, if $(i+1)$ is a multiple of 2^L. This suggests to split the iteration in chunks of size 2^L. Enumerating a chunk of size 2^L amounts to enumerate a polynomial in L variables (it requires the same amount of internal state, and it makes the same number of calls to NEXT). Let us now consider the k-th chunk:

$$C_k = \left\{ i \in \mathbb{N} \mid k \cdot 2^L < i \leq (k+1) \cdot 2^L \right\}$$

```
1: procedure ParallelZeroes(f, L, T)
2:   for b from 0 to 2^{n-L-T} - 1 do
2:     parallel-for t from 0 to 2^T - 1 do
3:       State[t] ← Init(f, 0, GrayCode((t + b · 2^T) · 2^L))
2:       for i from 0 to 2^L - 1 do
4:         if State[t].y = 0 then State[t].x is a zero of f
5:         Next(State[t])
6:     end for
7:   end-parallel for
6:     end for
8: end procedure
```

Fig. 7: Parallel enumeration, assuming one processing unit capable of running 2^T threads. It should be possible to improve it using the enumeration algorithm itself for initialization.

Let $\psi_L(i)$ denote the integer $(i \bmod 2^L)$. We will call $\psi_L(i)$ the *local part* of i when $i \in C_k$, and we will denote it by $\psi(i)$, when not ambiguous. So, what can we say about $b_j(i)$, when $i \in C_k$? We define the subset $\Omega_{k,j}$ of C_k to be such that if $i \in \Omega_{k,j}$, then $b_j(i)$ only depends on the local part $\psi(i)$ of i. Very clearly, we in fact have:

$$\Omega_{k,j} = \left\{ i \in C_k \mid \text{HammingWeight}(\psi(i)) \geq j \right\}$$

And the three following points are immediate to estalbish.

Lemma 5. *For any k and j, we have the following properties:*

i) If $j_1 < j_2$ then $\Omega_{k,j_2} \subseteq \Omega_{k,j_1}$

ii) $|\Omega_{k,j}| = 2^L - \sum_{\ell=0}^{j-1} \binom{L}{\ell}$

iii) If $i \in \Omega_{k,j}$, then $b_j(i) < L$.

Intuitively, lemma 5 tells us that on a chunk of size 2^L, the b_j that we will compute will be smaller than L except on $\mathcal{O}\left(L^{d-1}\right)$ points, and will only depend on the local part of the index. This has two interesting consequences:

1. Instead of having to store and maintain an internal state of $\sum_{i=0}^{d-1} \binom{n}{i} = \mathcal{O}\left(n^{d-1}\right)$ words, it is sufficient to deal with an internal state of $\sum_{i=0}^{d-1} \binom{L}{i} = \mathcal{O}\left(L^{d-1}\right)$ words.
2. If we were capable of processing all the chunks synchronously, the constant fetched from memory in line 9 of fig. 6 could be used by *all* the chunks *at the same time*, except on $\mathcal{O}\left(L^{d-1}\right)$ points. This means that *most of the time*, we can broadcast a single value to as many threads as possible, and we can amortize the lattency of the memory over the huge number of chunks processed in parallel.

Now that we controlled what happened *inside* $\Omega_{k,d}$, we may take a look at what happens *outside*. Generally speaking, if $\psi(i)$ has hamming weight h, and $j > h$, then $b_j(i) = L + b_{j-h}(k)$. Thus, if only a subset of all the chunks can be processed concurrently, it would make sense to treat simultaneously chunks for which k has similar least-significant bits. If a processing unit can handle at most 2^T threads simultaneously, then the maximum sharing of values fetched from main memory is achieved by scheduling the 2^T Chunks sharing the T most significant bit of k on it.

What level of broadcast should we expect in this situation, namely when 2^t threads process chunks of size 2^L synchronously? To fully understand what is going on, let us define $H_{k,h} = \Omega_{k,h} - \Omega_{k,h+1}$. It is easily seen that $H_{k,h}$ describes the subset of C_k formed of words of hamming weight exactly h, and thus $|H_{h,k}| = \binom{L}{h}$. Now, in all the chunks processed in parallel, the $c = n - L - t$ least significant bits of k remain fixed to $\psi_c(k)$, therefore we will call this value the "fixed part of k". We already argued that if $i \in H_{h,k}$, then $b_j(i) = L + b_{j-h}(k)$, and the 2^T threads will fetch the same memory location if and only if $b_{j-h}(k)$ only depends on the fixed part of k, or, in other terms, if $\psi_c(k)$ has hamming weight at least $j - h$.

An easy consequence of the previous considerations is that if $\psi_c(k)$ has hamming weight at least d, then *all* the memory fetches issued on the 2^L steps can be broadcast to *all* the 2^T threads. If $\psi_c(k)$ has hamming weight $d - 1$, then *all but one* memory fetches can be broadcast.

This raises the following question: assume we enumerate the 2^n points on a processing unit handling 2^T concurrent threads, each thread processing a chunk of size 2^L (we of course assume $n \geq L + T$). How many times we will witness non-broadcast memory accesses? Let us denote this number by $N_{NB}(d, n, T, L)$.

Proposition 4.

$$N_{NB}(d, n, T, L) = \sum_{i=0}^{d-1} \binom{n-T}{i}.$$

The proof is in the full version.

Fig. 8 shows how the algorithm can be run with 4 threads and obtain the number of non-broadcasts advertised by the proposition.

8 Implementations

We describe the structure of our code, the approximate cost structure, our design choices and justify what we did. Our implementation code always consists of three stages:

Partial Evaluation: We substitute all possible values for s variables $(x_{n-s}, \ldots, x_{n-1})$ out of n, thus splitting the original system into 2^s smaller systems, of w equations each in the remaining $(n - s)$ variables $(x_0, \ldots, x_{n-s-1})$, and output them in a form that is suitable for ...

Enumeration Kernel: Run the algorithm of Sec. 4 to find all candidate vectors $\mathbf{x}$ satisfying w equations out of m ($\approx 2^{n-w}$ of them), which are handed over to ...

Candidate Checking: Checking possible solutions $\mathbf{x}$ in remaining $m - w$ equations.

8.1 CPU Enumeration Kernel

Typical code fragments from the unrolled inner loops can be seen below:

thread 0				thread 1				thread 2				thread 3				
k	$\psi_L(i)$	$b_1(i)$	$b_2(i)$	k	$\psi_L(i)$	$b_1(i)$	$b_2(i)$	k	$\psi_L(i)$	$b_1(i)$	$b_2(i)$	k	$\psi_L(i)$	$b_1(i)$	$b_2(i)$	non-broadcast ?
0	1	0	-1	100	1	0	5	1000	1	0	6	1100	1	0	5	✓
0	10	1	-1	100	10	1	5	1000	10	1	6	1100	10	1	5	✓
0	11	0	1	100	11	0	1	1000	11	0	1	1100	11	0	1	
0	100	2	-1	100	100	2	5	1000	100	2	6	1100	100	2	5	✓
0	101	0	2	100	101	0	2	1000	101	0	2	1100	101	0	2	
0	110	1	2	100	110	1	2	1000	110	1	2	1100	110	1	2	
0	111	0	1	100	111	0	1	1000	111	0	1	1100	111	0	1	
0	0	3	-1	100	0	3	5	1000	0	3	6	1100	0	3	5	✓
1	1	0	3	101	1	0	3	1001	1	0	3	1101	1	0	3	
1	10	1	3	101	10	1	3	1001	10	1	3	1101	10	1	3	
1	11	0	1	101	11	0	1	1001	11	0	1	1101	11	0	1	
1	100	2	3	101	100	2	3	1001	100	2	3	1101	100	2	3	
1	101	0	2	101	101	0	2	1001	101	0	2	1101	101	0	2	
1	110	1	2	101	110	1	2	1001	110	1	2	1101	110	1	2	
1	111	0	1	101	111	0	1	1001	111	0	1	1101	111	0	1	
1	0	4	-1	101	0	4	5	1001	0	4	6	1101	0	4	5	✓
10	1	0	4	110	1	0	4	1010	1	0	4	1110	1	0	4	
10	10	1	4	110	10	1	4	1010	10	1	4	1110	10	1	4	
10	11	0	1	110	11	0	1	1010	11	0	1	1110	11	0	1	
10	100	2	4	110	100	2	4	1010	100	2	4	1110	100	2	4	
10	101	0	2	110	101	0	2	1010	101	0	2	1110	101	0	2	
10	110	1	2	110	110	1	2	1010	110	1	2	1110	110	1	2	
10	111	0	1	110	111	0	1	1010	111	0	1	1110	111	0	1	
10	0	3	4	110	0	3	4	1010	0	3	4	1110	0	3	4	
11	1	0	3	111	1	0	3	1011	1	0	3	1111	1	0	3	
11	10	1	3	111	10	1	3	1011	10	1	3	1111	10	1	3	
11	11	0	1	111	11	0	1	1011	11	0	1	1111	11	0	1	
11	100	2	3	111	100	2	3	1011	100	2	3	1111	100	2	3	
11	101	0	2	111	101	0	2	1011	101	0	2	1111	101	0	2	
11	110	1	2	111	110	1	2	1011	110	1	2	1111	110	1	2	
11	111	0	1	111	111	0	1	1011	111	0	1	1111	111	0	1	
11	0	5	-1	111	0	6	-1	1011	0	5	6	1111	0	7	-1	✓

Fig. 8: Enumeration with $n = 7$, in chunks of 2^3 elements with 4 batches of 4 concurrent threads. "Non-local" means that a constant of index greater than 3 is accessed, while "Non-broadcast" means that the 4 threads do not access the same memory location. In conformance with lemma 4, there are $1 + 7 - 2 = 6$ non-broadcast memory accesses.

(a) quadratics, C++ x86 instrinsics

```
...
diff0 ^= deg2_block[ 1 ];
res ^= diff0;
Mask = _mm_cmpeq_epi16(res, zero);
mask = _mm_movemask_epi8(Mask);
if(mask) check(mask, idx, x^155);
...
```

(b) quadratics, x86 assembly

```
.L746:
  movq      976(%rsp), %rax  //
  pxor      (%rax), %xmm2    // d_y ^= C_yz
  pxor      %xmm2, %xmm1     // res ^= d_y
  pxor      %xmm0, %xmm0     //
  pcmpeqw   %xmm1, %xmm0     // cmp words for eq
  pmovmskb  %xmm0, %eax      // movemask
  testw     %ax, %ax         // set flag for branch
  jne       .L1266           // if needed, check and
.L747:                       // comes back here
```

```
.L1624:
  movq     2616(%rsp), %rax   // load C_yza
  movdqa   2976(%rsp), %xmm0  // load d_yz
  pxor     (%rax), %xmm0      // d_yz ^= C_yza
  movdqa   %xmm0, 2976(%rsp)  // save d_yz
  pxor     8176(%rsp), %xmm0  // d_y ^= d_yz
  pxor     %xmm0, %xmm1       // res ^= d_y
  movdqa   %xmm0, 8176(%rsp)  // save d_y
  pxor     %xmm0, %xmm0       //
  pcmpeqw  %xmm1, %xmm0       // cmp words for eq
  pmovmskb %xmm0, %eax
  testw    %ax, %ax           // ...
  jne      .L2246             // branch to check
.L1625:                       // and comes back
```

(c) cubics, x86 assembly

```
...
diff[0] ^= deg3_ptr1[0];
diff[325] ^= diff[0];
res ^= diff[325];
Mask = _mm_cmpeq_epi16(res, zero);
mask = _mm_movemask_epi8(Mask);
if(mask) check(mask, idx, x^2);
...
```

(d) cubics, C++ x86 instrinsics

testing All zeroes in one byte, word, or dword in a XMM register can be tested cheaply on x86-64. We hence wrote code to test 16 or 32 equations at a time. Strangely enough, even though the code above is for 16 bits, the code for checking 32/8 bits at the same time is nearly identical, the only difference being that we would subtitute the intrinsics `_mm_cmpeq_epi32/8` for `_mm_cmpeq_epi16` (leading to the SSE2 instruction `pcmpeqd/b` instead of `pcmpeqw`). Whenever one of the words (or double words or bytes, if using another testing width) is non-zero, the program branches away and queues the candidate solution for checking.

unrolling One common aspect of our CPU and GPU code is deep unrolling by upwards of $1024\times$ to avoid the expensive bit-position indexing. To illustrate with quartics as an example, instead of having to compute the positions of the four rightmost non-zero bits in every integer, we only need to compute the first four rightmost non-zero bits in bit 10 or above, then fill in a few blanks. This avoids most of the indexing calculations and all the calculations involving the most commonly used differentials.

We wrote similar Python scripts to generate unrolled loops in C and CUDA code. Unrolling is even more critical with GPU, since divergent branching and memory accesses are prohibitively expensive.

8.2 GPU Enumeration Kernel

register usage Fast memory is precious on GPU and register usage critical for CUDA programmers. Our algorithms' memory complexity grows exponentially with the degree d, which is a serious problem when implementing the algorithm for cubic and quartic systems, compounded by the immaturity of NVIDIA's `nvcc` compiler which tends to allocate more registers than we expected.

Take quartic systems as an example. Recall that each thread needs to maintain third derivatives, which we may call d_{ijk} for $0 \le i < j < k < K$, where K is the number of variables in each small system. For $K = 10$, there are 120 d_{ijk}'s and we cannot waste all our registers on them, especially as all differentials are not equal — d_{ijk} is accessed with probability $2^{-(k+1)}$.

Our strategy for register use is simple: Pick a suitable bound u, and among third differentials d_{ijk} (and first and second differentials d_i and d_{ij}), put the most frequently used — i.e., all indices less than u — in registers, and the rest in device memory (which can be read every 8 instructions without choking). We can then control the number of registers used and find the best u empirically.

fast conditional move We discovered during implementation an undocumented feature of CUDA for G2xx series GPUs, namely that `nvcc` reliably generates conditional (predicated) move instructions, dispatched with exceptional adeptness.

```
...
xor.b32 $r19, $r19, c0[0x000c]       // d_y^=d_yz
xor.b32 $p1|$r20, $r17, $r20
mov.b32 $r3, $r1
mov.b32 $r1, s[$ofs1+0x0038]
xor.b32 $r4, $r4, c0[0x0010]
xor.b32 $p0|$r20, $r19, $r20         // res^=d_y
@$p1.eq mov.b32 $r3, $r1
@$p1.eq mov.b32 $r1, s[$ofs1+0x003c]
xor.b32 $r19, $r19, c0[0x0000]
xor.b32 $p1|$r20, $r4, $r20
@$p0.eq mov.b32 $r3, $r1             // cmov
@$p0.eq mov.b32 $r1, s[$ofs1+0x0040] // cmov
...
```

(a) `decuda` result from cubin

```
 ...
diff0 ^= deg2_block[ 3 ];   // d_y^=d_yz
res ^= diff0;               // res^=d_y
if( res == 0 ) y = z;       // cmov
if( res == 0 ) z = code233; // cmov
diff1 ^= deg2_block[ 4 ];
res ^= diff1;
if( res == 0 ) y = z;
if( res == 0 ) z = code234;
diff0 ^= deg2_block[ 0 ];
res ^= diff0;
if( res == 0 ) y = z;
if( res == 0 ) z = code235;
 ...
```

(b) CUDA code for a inner loop fragment

Consider, for example, the code displayed above right. According to our experimental results, the repetitive 4-line code segments average less than three SP (stream-processor) cycles. However, `decuda` output of our program shows that each such code segment corresponds to at least 4 instructions including 2 XORs and 2 conditional moves [as marked in above left]. The only explanation is that conditional moves can be dispatched by the SFUs (Special Function Units) so that the total throughput can exceed 1 instruction per SP cycle. Further note that the annotated segment on the right corresponds to actual instructions far apart because *an NVIDIA GPU does opportunistic dispatching but is nevertheless a purely in-order architecture,* so proper scheduling must interleave instructions from different parts of the code.

testing The inner loop for GPUs differs from CPUs due to the fast conditional moves.

Here we evaluate 32 equations at a time using Gray code. The result is used to set a flag if it happens to be all zeroes. We can now conditional move of the index based on the flag to a register variable `z`, and at the end of the loop write `z` out to global memory.

However, how can we tell if there are too many (here, *two*) candidate solutions in one small subsystem? Our answer to that is to use a buffer register variable `y` and a second conditional move using the same flag. At the end of the thread, (`y`, `z`) is written out to a specific location in device memory and sent back to the CPU.

Now subsystems which have all zero constant terms are automatically satisfied by the vector of zeroes. Hence we note them down during the partial evaluation phase include the zeros with the list of candidate solutions to be checked, and never have to worry about for all-zero candidate solution. The CPU reads the two doublewords corresponding to `y` and `z` for each thread, and:

1. `z==0` means no candidate solutions,
2. `z!=0` but `y==0` means exactly one candidate solution, and
3. `y!=0` means 2+ candidate solutions (necessitating a re-check).

8.3 Checking Candidates

Checking candidate solutions is always done on CPU because the programming involves branching and hence is difficult on a GPU even with that available. However, the checking code for CPU enumeration and GPU enumeration is different.

CPU With the CPU, the check code receives a list of candidate solutions. Today the maximum machine operation is 128-bit wide. Therefore we should collect solutions into groups of 128 possible solutions. We would rearrange 128 inputs of n bits such that they appear as n `__int128`'s, then evaluate one polynomial for 128 results in parallel using 128-bit wide ANDs and XORs. After we finish all candidates

for one equation, go through the results and discard candidates that are no longer possible. Repeat the result for the second and any further equations (cf. Sec. 3).

We need to transpose a bit-matrix to achieve the effect of a block of w inputs n-bit long each, to n machine-words of w-bit long. This looks costly, however, there is an SSE2 instruction `PMOVMSKB` (packed-move-mask-bytes) that packs the top bit of each byte in an XMM register into a 16-bit general-purpose register *with 1 cycle throughput*. We combine this with simultaneous shifts of bytes in an XMM and can, for example, on a K10+ transpose a 128-batch of 32-bit vectors (0.5kB total) into 32 `__int128`'s in about 800 cycles, or an overhead of 6.25 cycles per 32-bit vector. In general the transposition cost is at most a few cycles per byte of data, negligible for large systems.

GPU As explained above, for the GPU we receive a list with 3 kinds of entries:

1. The knowledge that there are two or more candidate solutions within that same small system, with only the position of the last one in the Gray code order recorded.
2. A candidate solution (and no other within the same small system).
3. Marks to subsystems that have all zero constant terms.

For Case 1, we take the same small system that was passed into the GPU and run the Enumerative Kernel subroutine in the CPU code and find all possible small systems. Since most of the time, there are exactly two candidate solutions, we expected the Gray code enumeration to go two-thirds of the way through the subsystem. Merge remaining candidate solutions with those of Case 2+3, collate for checking in a larger subsystem if needed, and pass off to the same routine as used in the CPU above. Not unexpectedly, the runtime is dominated by the thread-check case, since those does millions of cycles for two candidate solutions (most of the time).

8.4 Partial Evaluation

The algorithm for Partial Evaluation is for the most part the same Gray Code algorithm as used in the Enumeration Kernel. Also the highest degree coefficients remain constant, need no evaluation and and can be shared across the entire Enumeration Kernel stage. As has been mentioned in the GPU description, these will be stored in the *constant memory*, which is reasonably cached on NVIDIA CUDA cards. The other coefficients can be computed by Gray code enumeration, so for example for quadratics we have $(n-s)+2$ XOR per w bit-operations and per substitution. In all, the cost of the Partial Evaluation stage for w' equations is $\sim 2^s \frac{w'}{8} \left(\binom{n-s}{d-1} + (\text{smaller terms}) \right)$ byte memory writes. The only difference in the code to the Enumerative Kernel is we write out the result (smaller systems) to a buffer, and *check for a zero constant term only* (to find all-zero candidate solutions).

Peculiarities of GPUS Many warps of threads are required for GPUs to run at full speed, hence we must split a kernel into many threads, the initial state of each small system being provided by Partial Evaluation. In fact, for larger systems on GPUs, we do two stages of partial evaluation because

1. there is a limit to how many threads can be spawned, and how many small systems the device memory can hold, which bounds how small we can split; *but*
2. increasing s decreases the fast memory pressure; and
3. a small systems reporting two or more candidate solutions is costly, yet we can't run a batch check on a small system with only one candidate solution — hence, an intermediate partial evaluation so we can batch check with fewer variables.

8.5 More Test Data and Discussion

Some minor points which the reader might find useful in understanding the test data, a full set of which will appear in the extended version.

Candidate Checking. **The check code is now 6–10% of the runtime**. In theory (cf. Sec. 3) evaluation should start with a script which hard-wires a system of equations into C and compiling to an excutable, eliminating half of the terms, and leading to $\binom{n-s}{d}$ SSE2 (half XORs and half ANDs) operations to check one equation for $w = 128$ inputs. The check code can potentially become more than an order of magnitude faster. We do not (yet) do so presently, because compiling may take more time than the checking code. However, we may want to go this route for even larger systems, as the overhead from testing for zero bits, re-collating the results, and wasting due to the number of candidate solutions is not divisible by w would all go down proportionally.

Without hard-wiring, the running time of the candidate check is dominated by loading coefficients. E.g., for quartics with 44 variables, 14 pre-evaluated, K10+ and Ci7 averages 4300 and 3300 cycles respectively per candidate. With each candidate averaging 2 equations of $\binom{44-14}{4}$ terms each, the 128-wide inner loop averages about 10 and 7.7 cycles respectively per term to accomplish 1 `PXOR` and 1 `PAND`.

Partial Evaluation. We point out that Partial Evaluation also reduces the complexity of the Checking phase. The simplified description in Sec. 5 implies the cost of checking each candidate solution to be $\approx \frac{1}{w}\binom{n}{d}$ instructions. But we can get down to $\approx \frac{1}{w}\binom{n-s}{d}$ instructions by partially evaluating $w' > w$ equations and storing the result for checking. For example, when solving a quartic system with $n = 48$, $m = 64$, the best CPU results are $s = 18$, and we cut the complexity of the checking phase by factor of at least $4\times$ even if it was not the theoretical $7\times$ (i.e., $\binom{n}{d}/\binom{n-s}{d}$) due to overheads.

The Probability of Thread-Checking for GPUs. If we have n variables, pre-evaluate s, and check w equations via Gray Code, then the probability of a subsystem with 2^{n-s} vectors including at least two candidates $\approx \binom{2^{n-s}}{2}(1-2^{-w})^{2^{n-s}}(2^{-w})^2 \approx 1/2^{2(s+w-n)+1}$, provided that $n < s + w$. As an example, for $n = 48$, $s = 22$, $w = 32$, the thread-recheck probability is about 1 in 2^{13}, and we must re-check about 2^9 threads using Gray Code. This pushes up the optimal s for GPUs.

Architecture and Differences. All our tests with a huge variety of machines and video cards show that the kernel time in cycles per attempt is almost a constant of the architecture, and the speed-up in multi-cores is almost completely linear for almost all modern hardware. So we can compute the time complexity given the architecture, the frequency, the number of cores, and n. The marked cycle count difference between Intel and AMD cores is explained by Intel dispatching *three* XMM (SSE2) logical instructions to AMD's *two* per cycle and handling branch prediction and caching better.

As the Degree d increases. We plot how many cycles is taken by the inner loop (which is 8 vectors per core for CPUs and 1 vector per SP for GPUs) on different architectures in Fig. 9. As we can see, all except two architectures have inner loop cycle counts that are increasing roughly linearly with the degree. The exceptions are the AMD K10 and NVIDIA G200 architectures, which is in line with fast memory pressure on the NVIDIA GPUs and fact that K10 has the least cache among the CPU architectures.

More Tuning. We can conduct a Gaussian elimination among the m equations and such that $m/2$ selected terms in $m/2$ of the equations are all zero. We can of course make this the most commonly used coefficients (i.e., $c_{01}, c_{02}, c_{12}, \ldots$ for the quadratic case). The corresponding XOR instructions can be removed from the code by our code generator. This is not yet automated and we have to test everything by hand. However, this clearly leads to significant savings. On GPUs, we have a speed up of 21% on quadratic cases, 18% for cubics, and 4% for quadratics. [The last is again due to the memory problems.]

Notes and Acknowledgements

C. Bouillaguet thanks Jean Vuillemin for helpful discussions. The Taiwanese authors thank Ming-Shing Chen for assistance with programming and fruitful discussion, Taiwan's National Science Council for

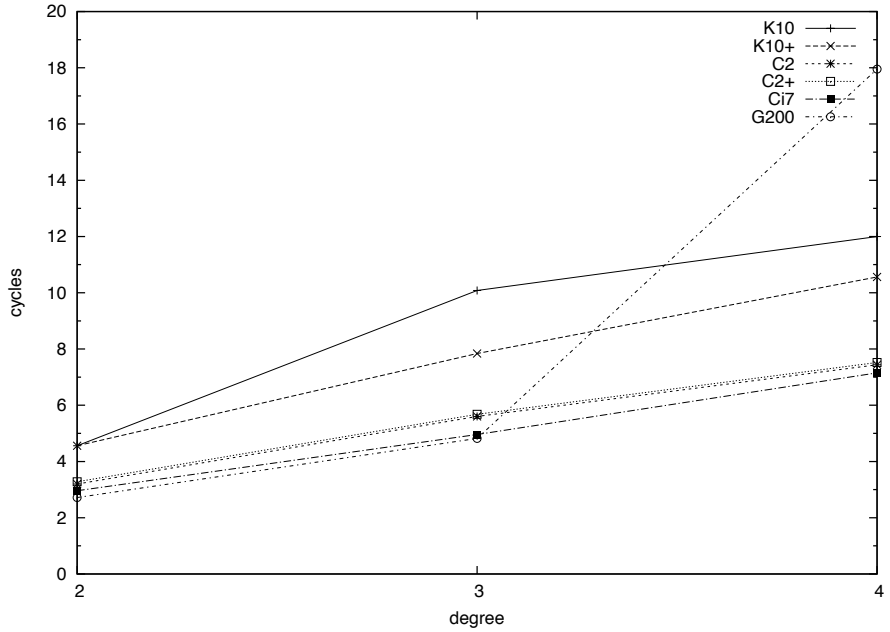


Fig. 9: Cycles per candidate tested for degree 2,3 and 4 polynomials.

Table 2. Efficiency comparison: cycles per candidate tested on one core

$n = 32$			$n = 40$			$n = 48$			Testing platform			
$d = 2$	$d = 3$	$d = 4$	$d = 2$	$d = 3$	$d = 4$	$d = 2$	$d = 3$	$d = 4$	GHz	Arch.	Name	USD
0.58	1.21	1.41	0.57	1.27	1.43	0.57	1.26	1.50	2.2	K10	Phenom9550	120
0.57	0.91	1.32	0.57	0.98	1.31	0.57	0.98	1.32	2.3	K10+	Opteron2376	184
0.40	0.65	0.95	0.40	0.70	0.94	0.40	0.70	0.93	2.4	C2	Xeon X3220	210
0.40	0.66	0.96	0.41	0.71	0.94	0.41	0.71	0.94	2.83	C2+	Core2 Q9550	225
0.50	0.66	1.00	0.38	0.65	0.91	0.37	0.62	0.89	2.26	Ci7	Xeon E5520	385
2.87	4.66	15.01	2.69	4.62	17.94	2.72	4.82	17.95	1.296	G200	GTX280	n/a
2.93	4.90	14.76	2.70	4.62	15.54	2.69	4.57	15.97	1.242	G200	GTX295	500

partial sponsorship under grants NSC96-2221-E-001-031-MY3, 98-2915-I-001-041, and 98-2219-E-011-001 (Taiwan Information Security Center), and Academia Sinica for the Career Development Award. Questions and esp. corrections about the extended version should be addressed to by@crypto.tw.

References

1. G. V. Bard, N. T. Courtois, and C. Jefferson. 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/.
2. M. Bardet, J.-C. Faugère, and B. Salvy. On the complexity of Gröbner basis computation of semi-regular overdetermined algebraic equations. In *Proceedings of the International Conference on Polynomial System Solving*, pages 71–74, 2004. Previously INRIA report RR-5049.
3. M. Bardet, J.-C. Faugère, B. Salvy, and B.-Y. Yang. Asymptotic expansion of the degree of regularity for semi-regular systems of equations. In P. Gianni, editor, *MEGA 2005 Sardinia (Italy)*, 2005.
4. A. T. Benjamin and J. Quinn. *Proofs that Really Count: The Art of Combinatorial Proof (Dolciani Mathematical Expositions)*. The Mathematical Association of America, August 2003.
5. C. Berbain, H. Gilbert, and J. Patarin. QUAD: A practical stream cipher with provable security. In S. Vaudenay, editor, *EUROCRYPT*, volume 4004 of *Lecture Notes in Computer Science*, pages 109–128. Springer, 2006.
6. D. J. Bernstein, T.-R. Chen, C.-M. Cheng, T. Lange, and B.-Y. Yang. ECM on graphics cards. In A. Joux, editor, *EUROCRYPT*, volume 5479 of *Lecture Notes in Computer Science*, pages 483–501. Springer, 2009. http://eprint.iacr.org/2008/480/.
7. L. Bettale, J.-C. Faugère, and L. Perret. Hybrid approach for solving multivariate systems over finite fields. *Journal of Mathematical Cryptology*, 3:177–197, 2009.
8. C. Bouillaguet, J.-C. Faugère, P.-A. Fouque, and L. Perret. Differential-algebraic algorithms for the isomorphism of polynomials problem. Cryptology ePrint Archive, Report 2009/583, 2009. http://eprint.iacr.org/.
9. C. Bouillaguet, P.-A. Fouque, A. Joux, and J. Treger. A family of weak keys in hfe (and the corresponding practical key-recovery). Cryptology ePrint Archive, Report 2009/619, 2009. http://eprint.iacr.org/.
10. B. Buchberger. *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal*. PhD thesis, Innsbruck, 1965.
11. J. Buchmann, D. Cabarcas, J. Ding, and M. S. E. Mohamed. Flexible partial enlargement to accelerate gröbner basis computation over $_2$. In D. J. Bernstein and T. Lange, editors, *AFRICACRYPT*, volume 6055 of *Lecture Notes in Computer Science*, pages 69–81. Springer, 2010.
12. N. Courtois, G. V. Bard, and D. Wagner. Algebraic and slide attacks on keeloq. In K. Nyberg, editor, *FSE*, volume 5086 of *Lecture Notes in Computer Science*, pages 97–115. Springer, 2008.
13. N. Courtois, L. Goubin, and J. Patarin. *SFLASH: Primitive specification (second revised version)*, 2002. https://www.cosic.esat.kuleuven.be/nessie, Submissions, Sflash, 11 pages.
14. N. T. Courtois, A. Klimov, J. Patarin, and A. Shamir. Efficient algorithms for solving overdefined systems of multivariate polynomial equations. In *Advances in Cryptology — EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 392–407. Bart Preneel, ed., Springer, 2000. Extended Version: http://www.minrank.org/xlfull.pdf.
15. N. de Bruijn. *Asymptotic methods in analysis. 2nd edition.* Bibliotheca Mathematica. Vol. 4. Groningen: P. Noordhoff Ltd. XII, 200 p. , 1961.
16. J.-C. Faugère. A new efficient algorithm for computing Gröbner bases (F_4). *Journal of Pure and Applied Algebra*, 139:61–88, June 1999.
17. J.-C. Faugère. A new efficient algorithm for computing Gröbner bases without reduction to zero (F_5). In *International Symposium on Symbolic and Algebraic Computation — ISSAC 2002*, pages 75–83. ACM Press, July 2002.
18. J.-C. Faugère and A. Joux. Algebraic cryptanalysis of Hidden Field Equations (HFE) using Gröbner bases. In *Advances in Cryptology — CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 44–60. Dan Boneh, ed., Springer, 2003.
19. A. Fog. *Instruction Tables*. Copenhagen University, College of Engineering, Feb 2010. Lists of Instruction Latencies, Throughputs and micro-operation breakdowns for Intel, AMD, and VIA CPUs, http://www.agner.org/optimize/instruction_tables.pdf.

20. M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *FOCS*, pages 285–298, 1999.
21. J. Patarin. Asymmetric cryptography with a hidden monomial. In *Advances in Cryptology — CRYPTO 1996*, volume 1109 of *Lecture Notes in Computer Science*, pages 45–60. Neal Koblitz, ed., Springer, 1996.
22. J. Patarin. Hidden Field Equations (HFE) and Isomorphisms of Polynomials (IP): two new families of asymmetric algorithms. In *Advances in Cryptology — EUROCRYPT 1996*, volume 1070 of *Lecture Notes in Computer Science*, pages 33–48. Ueli Maurer, ed., Springer, 1996. Extended Version: `http://www.minrank.org/hfe.pdf`.
23. J. Patarin, N. Courtois, and L. Goubin. QUARTZ, 128-Bit Long Digital Signatures. In D. Naccache, editor, *CT-RSA*, volume 2020 of *Lecture Notes in Computer Science*, pages 282–297. Springer, 2001.
24. J. Patarin, L. Goubin, and N. Courtois. Improved algorithms for Isomorphisms of Polynomials. In *Advances in Cryptology — EUROCRYPT 1998*, volume 1403 of *Lecture Notes in Computer Science*, pages 184–200. Kaisa Nyberg, ed., Springer, 1998. Extended Version: `http://www.minrank.org/ip6long.ps`.
25. H. Raddum. Mrhs equation systems. In C. M. Adams, A. Miri, and M. J. Wiener, editors, *Selected Areas in Cryptography*, volume 4876 of *Lecture Notes in Computer Science*, pages 232–245. Springer, 2007.
26. M. Sugita, M. Kawazoe, L. Perret, and H. Imai. Algebraic cryptanalysis of 58-round sha-1. In A. Biryukov, editor, *FSE*, volume 4593 of *Lecture Notes in Computer Science*, pages 349–365. Springer, 2007.
27. B.-Y. Yang and J.-M. Chen. Theoretical analysis of XL over small fields. In *ACISP 2004*, volume 3108 of *Lecture Notes in Computer Science*, pages 277–288. Springer, 2004.
28. B.-Y. Yang, J.-M. Chen, and N. Courtois. On asymptotic security estimates in XL and Gröbner bases-related algebraic cryptanalysis. In *ICICS 2004*, volume 3269 of *Lecture Notes in Computer Science*, pages 401–413. Springer, Oct. 2004.

Links Between Theoretical and Effective Differential Probabilities: Experiments on PRESENT

C. Blondeau and B. Gérard

INRIA project-team SECRET, France
{celine.blondeau, benoit.gerard}@inria.fr

Abstract. Recent iterated ciphers have been designed to be resistant to differential cryptanalysis. This implies that cryptanalysts have to deal with differentials having so small probabilities that, for a fixed key, the whole codebook may not be sufficient to detect it. The question is then, do these theoretically computed small probabilities have any sense? We propose here a deep study of differential and differential trail probabilities supported by experimental results obtained on a reduced version of PRESENT.
Keywords : differential cryptanalysis, differential probability, iterated block cipher, PRESENT.

1 Introduction

Differential cryptanalysis has first been applied to the *Data Encryption Standard* (DES) in the early 90's by E. Biham and A. Shamir [BS91,BS92]. Since then, many ciphers have been cryptanalyzed using differential cryptanalysis or one of the large family of variants (truncated differential [Knu94], higher order differential [Knu94], impossible differential [BBS99], ...).

The basic differential cryptanalysis is based on a differential over r rounds of the cipher.

Definition 1. *A r-rounds differential*
A r-rounds differential *is a couple* $(\delta_0, \delta_r) \in \mathbb{F}_2^m \times \mathbb{F}_2^m$. *The* probability of a differential (δ_0, δ_r) *is*

$$p_* \stackrel{\text{def}}{=} \Pr_{\mathbf{X},\mathbf{K}}\left[F_K^r(X) \oplus F_K^r(X \oplus \delta_0) = \delta_r\right],$$

where m is the input/output size of the cipher and F the round function.

Then, r rounds of the cipher can be distinguished from a random permutation using this differential. To break $r+1$ rounds of the block cipher, we look for differentials on r rounds. Then, for all the possible subkeys for the last round, the attacker does a partial decryption of the ciphertext pairs and count the number of time δ_r appears. For a wrong candidate, the probability that δ_r appears is around 2^{-m} and for the correct subkey, this probability is around $p_* + 2^{-m}$. It is widely accepted that the number of pairs needed to distinguish those two probabilities is of order p_*^{-1} if the so called *signal-to-noise ratio* is large enough $\left(S_N = \frac{p_*}{2^{-m}}\right)$.

Recent ciphers, the *Advanded Encryption Standard* (AES) for instance, have been designed to be resistant to the basic differential cryptanalysis. Nevertheless, when a new cipher is proposed, cryptanalysts try to mount the best possible linear and differential attacks. In the case of PRESENT[BKL+07], the cipher we used for the experiments, the actual best published attack is the one of Wang [Wan08]. But actually, there is still lacks in the data complexity estimates of those differential attacks.

The first one is the use of Gaussian or Poisson distributions to estimate what actually is a binomial distribution. Since we are interested in differential cryptanalysis, Gaussian distribution is known to be worse than Poisson [Sel08] but such an approximation is used to estimate the success probability of most of the recent differential cryptanalyses. Nevertheless, Poisson distribution might not be tight if the differential probability is close to the uniform probability 2^{-m}. Work has been done to give good estimates of the data complexity and the success probability of a statistical cryptanalysis for any setting [BGT10]. That is the reason why we chose to directly deal with binomial distributions without making any approximation.

The second point is the estimation of a differential probability. It is well known that a differential is composed of many trails and that the probability of one trail may not be a good estimate of the whole differential probability [NK92]. Again, in most of the recent differential cryptanalysis papers, the differential probability is estimated computing the probability of the main trail. We recall in Subsection 3.4 how to efficiently find many trails.

Last but not least, a widely used assumption is made in statistical cryptanalysis that is the assumption of *fixed key equivalence* or *stochastic equivalence* that is assuming that the probability of a differential that is computed over all the possible keys is roughly the same that the probability of a differential for some fixed key [LMM91].

2

Contributions of this work.
A deep study of this hypothesis has been done in [DR05] and this paper aims at providing evidences to confirm this theory by the way of practical experiments on a toy version of PRESENT.

We first present the cipher we used for experiments Section 2. Then, in Section 3, we focus on differential trails that is sets of intermediate differences taken by a pair that matches a differential. The classical way of estimating a trail probability relies on some hypotheses that are not true. Nevertheless, experiments show that this theoretical probability makes sense as an average of the probability over the keys. In Section 4 we recall that a differential probability is the sum of the corresponding trail probabilities. Then, we present some experiments about the key dependency of this differential probability that corroborate the results in [DR05]. Finally, we conclude in Section 5 and sum-up the results as well as the problematics left as open questions.

2 PRESENT

Experiments are made on a lightweight cipher presented in 2007 at *CHES* conference: PRESENT [BKL$^+$07]. This cipher is a *Substitution Permutation Network* and thus is easy to describe.

2.1 Reduced version of PRESENT: SMALLPRESENT-[s]

For the experiments to be meaningful, we need to be able to exhaustively compute the ciphertexts corresponding to all possible plaintexts for all possible keys. That is the reason why we chose to work on a reduced version of PRESENT named SMALLPRESENT-[s] [Lea10]. The family of SMALLPRESENT-[s] has been designed for such experiments. The value of s indicate the number of Sboxes of the cipher. These Sboxes are all the same which is defined on $\mathbb{F}_2^4$. This substitution is described in Table 1. The size of the message is then $4\,s$. In this paper, we make experiments on SMALLPRESENT-[4] that is the version with 4 Sboxes (the full version of PRESENT has 16 Sboxes). One round of SMALLPRESENT-[4] and PRESENT are respectively depicted in Figure 8, Figure 9 (Appendix B).

2.2 Different key schedules for SMALLPRESENT-[4]

The problem with the reduced cipher presented in [Lea10] is the key schedule. Actually, in the whole PRESENT, most of the bits of a subkey are directly used in the subkey of the next round. Since, for SMALLPRESENT-[s], the number of key bits is always 80 but the state size is only $4\,s$, this

is not true anymore for a small s. We decided to introduce two additional key schedules for our experiments.

1. *Same key*: The cipher has a master key that has the same size as the state and each subkey is equal to this master key. Therefore SMALLPRESENT-[4] is parameterized by a 16 bits master key.
2. *80-bits*: This key schedule is the one used in the full version of PRESENT and proposed in [Lea10].
3. *20-bits*: a homemade key schedule used with SMALLPRESENT-[4] similar to the one of the full version.
   The master key is represented as $K = k_{19}k_{18}\dots k_0$. At round i the 16-bits round key $K_i = k_{19}k_{18}\dots k_4$ consists in the 16 left-most bits of the current content of register K. After extracting the round key K_i, the key register is updated as follows:
   (a) $[k_{19}k_{18}\dots k_1k_0] = [k_6k_5\dots k_8k_7]$
   (b) $[k_{19}k_{18}k_{17}k_{16}] = S[k_{19}k_{18}k_{17}k_{16}]$
   (c) $[k_7k_6k_5k_4k_3] = [k_7k_6k_5k_4k_3] \oplus \textit{roundcounter}$
   The key is rotated by 13 bit positions to the left, the left most four bits are passed through the PRESENT Sbox, and the *roundcounter* value is exclusive-ored with bits $k_8k_7k_6k_5k_4$. We keep the 5-bits counter version. But we only study less than 7 rounds of SMALLPRESENT-[4] so the counter can be represented in 3 bits.

3 Differential trail probability

3.1 Notation

Let us denote by K the master key. The round subkeys derived from K are denoted by $K_1, K_2, \dots, K_r$. Let $F_{K_i} : \mathbb{F}_2^m \mapsto \mathbb{F}_2^m$ be a round function of a block cipher. We will denote by F_K^r the application of r rounds of the block cipher.

$$F_K^r = F_{K_r} \circ F_{K_{r-1}} \circ \cdots \circ F_{K_1}.$$

Generally, there is not one but many ways to go from the input difference to the output difference of a differential. Since the term if *differential characteristic* seems to be ambiguous, we use the linear cryptanalysis notation and call such a way a *differential trail*.

Definition 2. *Differential trail*
A differential trail *of a cipher is a* $(r+1)$*-tuple* $(\beta_0, \beta_1, \cdots, \beta_r) \in (\mathbb{F}_2^m)^{r+1}$ *of intermediate differences at each round. The* probability of a differential trail $\beta = (\beta_0, \beta_1, \cdots, \beta_r)$ *is*

$$\Pr_{\mathbf{X},\mathbf{K}}\left[\forall i \;\; F_K^i(X) \oplus F_K^i(X \oplus \beta_0) = \beta_i\right].$$

Computing the exact value of a trail probability is not possible for real ciphers since it needs to encipher the whole codebook for all possible keys. The classical way of estimating a trail probability is to chain trails on 1 round. This approach is based on a formalism introduced by Lai, Massey and Murphy [LMM91].

Markov cipher
A Markov cipher is a cipher for which, the probability

$$\Pr_{\mathbf{X},\mathbf{K}}\left[F_K^r(X) \oplus F_K^r(X \oplus \delta_0) = \delta_r | X = x\right]$$

does not depend on x if the subkeys $\mathbf{K}_i$ are uniformly distributed.

In the case of Markov ciphers where the subkeys are xored to the state, the theoretical probability of a trail $\beta = (\beta_0, \beta_1, \cdots, \beta_r)$ is computed as follow

$$p_\beta^t = \prod_{i=1}^{r} \Pr_{\mathbf{X}}\left[F(X) \oplus F(X \oplus \beta_{i-1}) = \beta_i\right].$$

Notice that we did not used the notation F_{K_i} because when the subkeys are xored to the state, the probability $\Pr_{\mathbf{X}}\left[F(X) \oplus F(X \oplus \beta_{i-1}) = \beta_i\right]$ does not depend on the value of the subkey.

3.2 Key dependency of a trail

The probability of a differential trail can be influenced by the choice of the master key used to encipher samples. This remark is the main motivation of the work in [DR05]. In order to take into account this fact, let us introduce some notation. For a r-round differential trail $\beta = (\beta_0, \beta_1, \cdots, \beta_r)$, let us define

$$\begin{aligned} T_K &\stackrel{\text{def}}{=} \frac{1}{2}\#\{X \in \mathbb{F}_2^m | F_K^i(X) \oplus F_K^i(X + \beta_0) = \beta_i \quad \forall\ 1 \le i \le r\}, \\ T[j] &\stackrel{\text{def}}{=} \#\{K | T_K = j\}. \end{aligned} \tag{1}$$

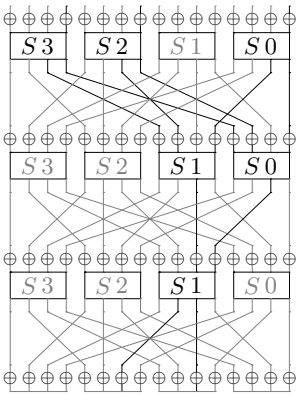
Let n_k be the number of bits of the master key. The *real* or *effective value* of the trail probability is

$$p_\beta = 2^{-m-1} \sum_{K \in \mathbb{F}_2^{n_k}} T_K = 2^{-m-1-n_k} \sum_j T[j] \cdot j.$$

To motivate these new notation, we give an example where the key dependency is obvious.

Example of a trails with experimental probability not equal to the theoretical one

We illustrate this phenomena by a differential trail over 3 rounds on SMALLPRESENT - [4]: $\beta = (\texttt{0x1101}, \texttt{0xdd}, \texttt{0x30}, \texttt{0x220})$ (see Figure 1).



j	0	8	16
$T[j]$	131072	524288	393216

Fig. 1. Trail $\beta = (\texttt{0x1101}, \texttt{0xdd}, \texttt{0x30}, \texttt{0x220})$ and the corresponding $T[j]$'s

First, we are going to compute the theoretical probability of this trail. We suppose that SMALLPRESENT-[4] is a Markov cipher and that uses independent subkeys.

- *Round 1* 3 S-boxes with input difference `0x1` and output difference `0x3`.
- *Round 2* 2 S-boxes with input difference `0xd` and output difference `0x2`.
- *Round 3* 1 S-box with input difference `0x3` and output difference `0x6`.

We have 6-Sboxes with transition probability 2^{-2} therefore $p_\beta^t = 2^{-12}$. This means that the number of plaintext such that $(X, X \oplus \texttt{0x1101})$ follows this trail for a fixed key should be $2^{16-1} \cdot 2^{-12} = 2^3$. We made experiments to check this assumption. For a fixed key we computed the number of plaintexts that follows this trail. In Figure 1 are given the values taken by $T[j]$ for all keys in $\mathbb{F}_2^{20}$ using the 20-bit key schedule. We can see that there are three kinds of key leading to three different values of $T[j]$.

Experiments on this trail show that for a fixed key, the theoretical probability of a differential trail do not always match with the value of

this trail probability for a fixed key. Experiments also show that for some keys the trail can be impossible. This can be of real significance because such phenomenon is also existing on 3 rounds of PRESENT. That means that, maybe, some differential trails used in a differential cryptanalysis may not have the expected probability for most of the keys.

Averaging the probabilities over the keys, we see that the effective probability of this differential trail is $2^{-11.6}$ (the theoretical one is 2^{-12}). This difference between real and theoretical probabilities may weaken (or strengthen) some attacks. Does a lot of trails have such a difference between their theoretical value and the average probability? In the next subsection we will show that, most of the times, the theoretical value of a differential trail is close to the effective one (averaged over the keys).

3.3 Theoretical probability and average probability of a trail over the keys

We observed that the theoretical probability is likely to be the average of the trail probabilities over all the possible keys. We made experiments on SMALLPRESENT-[4] with different key schedules. Let us recall that the theoretical probability of the differential trail β is denoted by p_β^t and the effective one (averaged over the keys) is denoted by p_β. In Figure 4, Figure 3 and Figure 2, we have computed the difference between $\log(p_\beta^t)$ and $\log(p_\beta)$ for 500 random trails.

- In *Figure 4* we assume that the round subkeys are derived from the 20-bits key schedule. We average the probabilities over the whole set of 2^{20} keys to obtain the value p_β.
- In *Figure 3* we assume that all the round subkeys are the same. We average the probabilities over the whole set of 2^{16} keys to obtain the value p_β.
- In *Figure 2* we assume that the round subkeys are derived from the 80-bits key schedule. Since we cannot average probabilities over the 2^{80} possible master keys, the computed value is obtained averaging over 2^{20} keys.

Remark:
We can see that the phenomenon is not the same depending on the key schedule. Indeed, in Figure 3 when the same key is taken over the 5 rounds the dependency of the round key is more important than in Figure 2 where the 80-bits key schedule is used implying that all the key

Number of trails as a function of $\log(\mathbf{p}_\beta) - \log(\mathbf{p}^\mathbf{t}_\beta)$ for a sample of 500 trails on 5 rounds using different key schedules

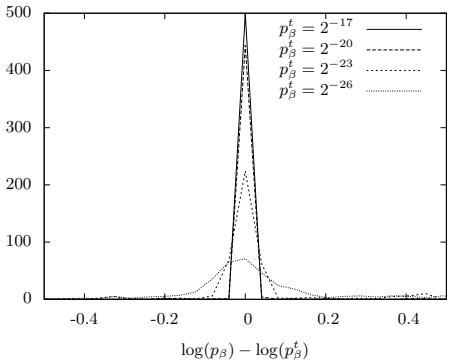


Fig. 2. 80-bits key schedule.

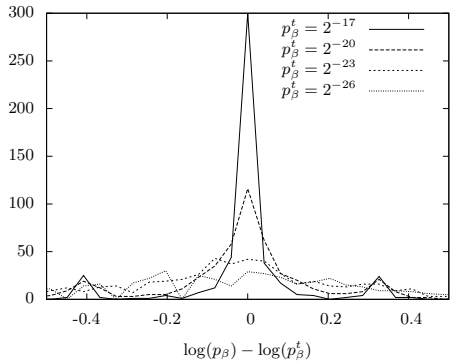


Fig. 3. same subkey for all rounds.

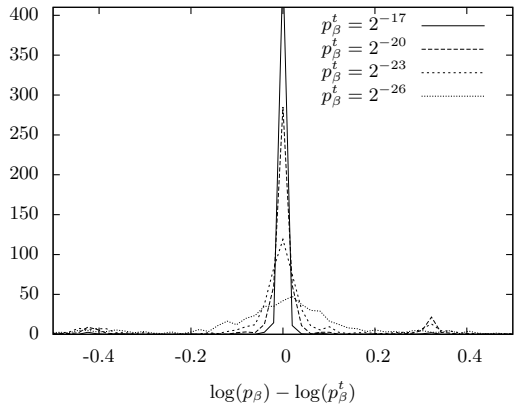


Fig. 4. 20-bits key schedule.

bits are used only once (on average). This remark has motivate the 20-bits key schedule we use in the following experiments that seems to be the most appropriate.

Experiments show that the average proportion of pairs satisfying a differential trail is close to the theoretical probability. We can observe that this behavior is getting worse as the probability is decreasing.
Nevertheless, it seems to be some symmetry what leads to the idea that taking enough trails into account will correct this and give better results.

3.4 Automatic search of differential trails

In order to find the best trails, we use a Branch and Bound algorithm (the one used in linear cryptanalysis). This one is explained in Appendix A.

4 Differential probability

4.1 Differential probability and trail probabilities

The first thing to say here is that the probability of a differential is the sum of the probabilities of the corresponding differential trails.

Lemma 1. *Let (δ_0, δ_r) be a r-round differential. Then the probability p_* of this differential is*

$$p_* = \sum_{\beta=(\delta_0,\beta_1,\ldots,\beta_{r-1},\delta_r)} p_\beta.$$

Proof. A pair that matches a trail cannot match any other (they are disjoint events) and thus $\Pr[\cup_i A_i] = \sum_i \Pr[A_i]$.

For a large number of rounds, it is impossible to compute the probability of a differential (δ_0, δ_r) because there is too much differential trails that go from δ_0 to δ_r. Actually, in differential cryptanalysis, one uses a lower bound on the probability of the differential (δ_0, δ_r) by considering the sum of the likeliest trail probabilities.

In Section 3, we saw that the effective trail probability may not match with the theoretical one. Nevertheless, it seems to be some symmetry what leads to the idea that the sum of theoretical trail probabilities may give a good estimate of a effective differential probability.

We made some experiments on 5 rounds of SMALLPRESENT-[4] with the 20 bits key schedule to see how many trails are required to get a good estimate of a differential probability. We computed the sum of the theoretical probabilities of many trails corresponding to the same differential. Since the cipher is small we also have computed the effective value of the differential by averaging over all plaintexts and all keys. In Figure 5 we have plotted the difference between both values for 20 differentials. We can see that taking many trails give a better estimation of the differential probability.

Looking at the results in Figure 5 we can wonder whether it is possible to determine the number of trails to consider for estimating a differential probability. In this example we see that taking 2^7 trails seems to be sufficient but when we look at the whole cipher it is obviously not enough (see the following paragraph).

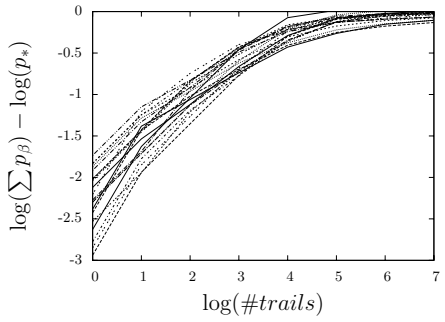


Fig. 5. Convergence of the sum of trails probabilities to the real value of the differential probability.

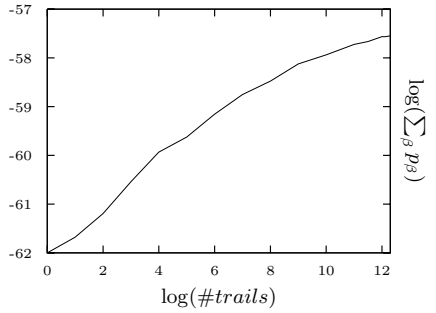


Fig. 6. Probability of the differential presented in [Wan08] as a function of the logarithm of the number of trails.

Remark on Wang's paper [Wan08]
In [Wan08], the target is the whole PRESENT (64 bits). One of the differentials used is

$$(d_0, d_{14}) = (\texttt{0x0700000000000700}, \texttt{0x0000000900000009})$$

on 14 rounds obtained by iterating 3 times a differential trail on 4 rounds and by adding one more round at the beginning and at the end.

- This trail on 4 rounds is not one of the best since it has a probability equal to 2^{-18} and we found a lot of differential trails with probability 2^{-12}. Nevertheless, it is the best iterative differential on 4 rounds.
- There exists lots of differential trails on 14 rounds with probability 2^{-62} and using algorithm given in Subsection 3.4, 2^{-62} seems to be the best trail probability over 14 rounds.
- We have theoretically computed all differential trails with input difference d_0, output difference d_{14} and probability greater than 2^{-73}. Summing the probabilities of the 2^{12} best trails, we observe that the probability of the differential (d_0, d_{14}) is greater than $2^{-57.53}$ and that it does not seem to converge yet (see Figure 6).

4.2 Key dependency of a differential probability

We now consider a differential (δ_0, δ_r) that is to be used in a differential cryptanalysis. The attacker will get some samples enciphered with a fixed master key. Depending on this key, the real probability of the differential will be smaller/equal/larger than the theoretically computed value.

For a fixed key K, let us denote by D_K the number of pairs of plaintexts with input difference δ_0 that lead to an output difference δ_r. Since we do not want to count a pair twice, we introduce a $\frac{1}{2}$ coefficient.

$$D_K \stackrel{\text{def}}{=} \frac{1}{2}\#\{X|F_K^r(X) + F_K^r(X+\delta_0) = \delta_r\}.$$

We are going to study the distribution of D_K's.

$$D[j] \stackrel{\text{def}}{=} \#\{K|D_K = j\}.$$

It is proven in [DR05] that D_K follows a hypergeometric distribution that, in cryptography setting, is tightly approximated by a binomial distribution of parameters $(2^{m-1}, p_*)$.

We made some experiments on 5 rounds of SMALLPRESENT-[4] to check this. Using the 20-bits key schedule we computed the repartition of the D_K's. In Figure 7, we see that the D_K's seems to follow a binomial distribution.

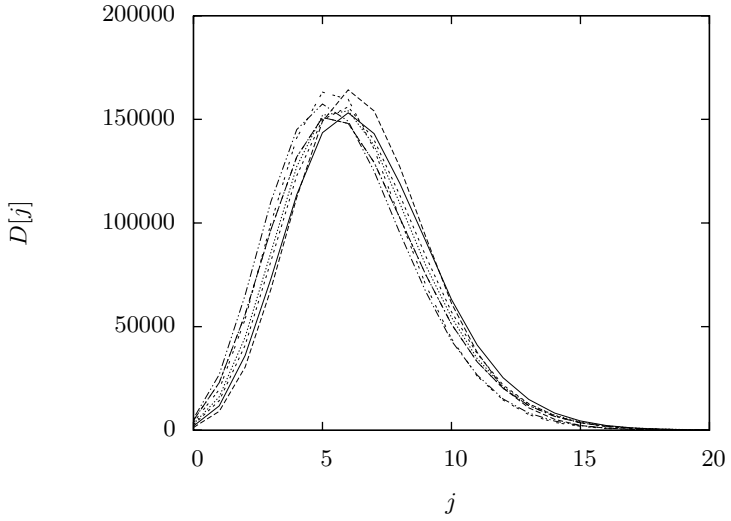


Fig. 7. Distribution of $D[j]$'s for 8 differentials over 5 rounds of SMALLPRESENT-[4].

This observation should be taken into account when computing the success probability of an attack. Let us denote by p_*^t the theoretical probability of the differential used in a cryptanalysis. We recall that n_k is the number of key bits. For $K \in \mathbb{F}_2^{n_k}$, the effective probability of the differential is $\frac{D_K}{2^{m-1}}$ where D_K is a random variable that follows a binomial distribution of parameters $(2^{m-1}, p_*^t)$. If we denote by $P_S(p_*)$ the success probability of a differential cryptanalysis using a differential with **effective** probability p_* (see [Sel08,BGT10]). Then the success probability of a differential cryptanalysis using a differential with **theoretical** probability p_*^t is

$$P_{\text{success}} = \sum_{i=0}^{2^{m-1}} P_S\left(\frac{i}{2^{m-1}}\right) \cdot \left[(p_*^t)^i (1-p_*^t)^{2^{m-1}-i} \binom{2^{m-1}}{i}\right].$$

5 Conclusion

We have presented lots of experiments on differential cryptanalysis. The main teaching of this work is that claimed complexities of differential cryptanalyses on recent ciphers may be under/over-estimated.

The first point is the fact that estimating a differential probability with the probability of its main trail is really not suitable. To illustrate the first point, we estimated the probability of a differential used in [Wan08] to $2^{-57.53}$ while the author only takes into account the best trail and provides an estimate of 2^{-62}.

The second point is the key dependency of a differential probability. Experiments confirmed the theory exposed in [DR05] and thus we propose a formula for the success probability that takes this phenomenon into account.

This work give some elements for understanding differential cryptanalysis but it still remains some open questions. The two main problematics that seems to be of great interest are the following.

- The theoretical probability of a trail seems to be less meaningful as this probability decreases. How far does this theoretical value make sense?
- How can we get a good estimate of a differential probability without finding all the corresponding differential trails?

References

[BBS99] E. Biham, A. Biryukov, and A. Shamir. Cryptanalysis of Skipjack Reduced to 31 Rounds Using Impossible Differentials. In *EUROCRYPT '99*, volume 1592 of *LNCS*, pages 12–23, 1999.

[BGT10] C. Blondeau, B. Gérard, and J.-P. Tillich. Accurate Estimates of the Data Complexity and Success Probability for Various Cryptanalyses . *DCC special issue on Coding and Cryptography*, 2010. To appear.

[BKL+07] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin, and C. Vikkelsoe. PRESENT: An Ultra-Lightweight Block Cipher. In *CHES '07*, volume 4727 of *LNCS*, pages 450–466. SV, 2007.

[BS91] E. Biham and A. Shamir. Differential Cryptanalysis of DES-like Cryptosystems. *Journal of Cryptology*, 4(1):3–72, 1991.

[BS92] E. Biham and A. Shamir. Differential Cryptanalysis of the Full 16-round DES. In *CRYPTO'92*, volume 740 of *LNCS*, pages 487–496. Springer–Verlag, 1992.

[DR05] J. Daemen and V. Rijmen. Probability distributions of Correlation and Differentials in Block Ciphers. Cryptology ePrint Archive, Report 2005/212, 2005. http://eprint.iacr.org/.

[Knu94] L. R. Knudsen. Truncated and Higher Order Differentials. In *FSE '94*, volume 1008 of *LNCS*, pages 196–211. Springer–Verlag, 1994.

[Lea10] G. Leander. Small Scale Variants Of The Block Cipher PRESENT. Cryptology ePrint Archive, Report 2010/143, 2010. http://eprint.iacr.org/.

[LMM91] X. Lai, J. L. Massey, and S. Murphy. Markov Ciphers and Differential Cryptanalysis. In *EUROCRYPT '91*, volume 547, pages 17–38, 1991.

[NK92] K. Nyberg and L.R. Knudsen. Provable Security Against Differential Cryptanalysis. In *CRYPTO'92*, volume 740 of *LNCS*, pages 566–574. Springer–Verlag, 1992.

[Sel08] A. A. Selçuk. On Probability of Success in Linear and Differential Cryptanalysis. *Journal of Cryptology*, 21(1):131–147, 2008.

[Wan08] M. Wang. Differential Cryptanalysis of Reduced-Round PRESENT. In *AFRICACRYPT '08*, volume 5023 of *LNCS*, pages 40–49. SV, 2008.

A Algorithm for finding differential trails

Let **B** be a lower bound on the probability of the trails we are looking for. We suppose that we are interested in differential trails over r rounds and that we already know the best trail probabilities for a smaller number of rounds.

We are going to traverse the tree defined as follow.

- Each node contains a difference.
- The root contains the input difference.
- The sons of a node correspond to all differences that are reachable after one round of the cipher from the input difference contained in the node.

– An edge has a weight that corresponds to the probability of transition from the father's difference to the son's one.

Then, we do a depth-first traversal of this tree and only consider leaves of the tree. The path from the root to the leave is a differential trail. The probability of this trail is computed multiplying the weights of the path edges.

There is a simple criterion to avoid some useless branches. When going from a father to a son, we compute the path probability from the root to the son and multiply it to the best trail probability for the remaining rounds (that is the depth between the son and the leaves). If it is smaller than **B**, then no leaves under the son will leads to a trail with probability greater than **B**. Then we look at another son and so on... Notice that this criterion can be quickly checked because the probability of a trail is computed as one advances through the tree and thus when looking at a node, the probability of the path from the root to that node is already known (the cost is one multiplication when going one step deeper and one division when returning to the father node).

B Characteristics of PRESENT

Here are the Sbox and the round function of both PRESENT and SMALL-PRESENT - [4].

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
S(x)	C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2

Table 1. The S-box of PRESENT

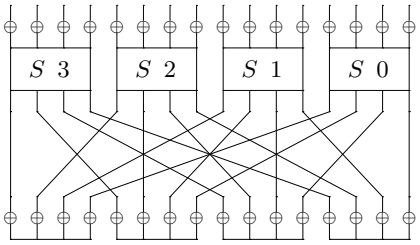


Fig. 8. 1 round of SMALLPRESENT-[4]

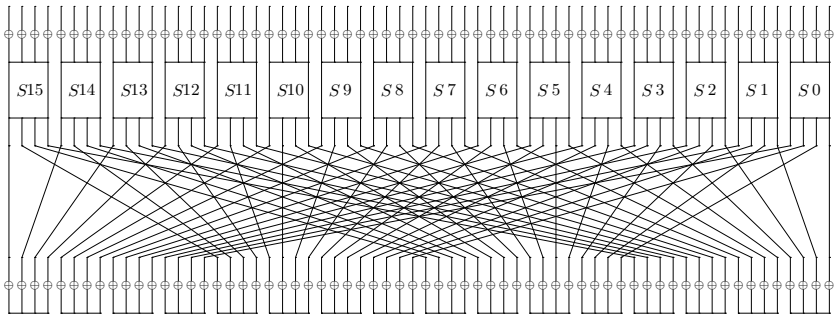


Fig. 9. 1 round of PRESENT

Toolkit for the Differential Cryptanalysis of ARX-based Cryptographic Constructions*

Nicky Mouha[1,2,**], Vesselin Velichkov[1,2,***], Christophe De Cannière[1,2,†], and Bart Preneel[1,2]

[1] Department of Electrical Engineering ESAT/SCD-COSIC, Katholieke Universiteit Leuven. Kasteelpark Arenberg 10, B-3001 Heverlee, Belgium.
[2] Interdisciplinary Institute for BroadBand Technology (IBBT), Belgium.
{Nicky.Mouha,Vesselin.Velichkov,Christophe.DeCanniere}@esat.kuleuven.be

Abstract

We propose a software toolkit, intended to automate the differential cryptanalysis of cryptographic constructions based on the operations addition, rotation and xor (ARX). The toolkit consists of several programs, each of which evaluates the probability that xor or additive differences propagate through a certain type of operation. Types of operations that are supported are xor, modular addition and multiplication by a constant.

A subset of the problems to which the proposed toolkit can be applied, have been studied in literature before. In [1], matrix multiplications are used to calculate the differential probability xdp^+ of addition modulo 2^n, when differences are expressed using xor, and the differential probability $\mathrm{adp}^\oplus$ of xor when differences are expressed using addition modulo 2^n. The time complexity of these computations is linear in the word size n.

In our toolkit, we use the same concept of matrix multiplications. The generated matrices are correct by construction, and their size is automatically minimized. The main advantage of our technique, is that it is more general, and can therefore easily be extended to a larger number of cases. The proposed tools can be used to compute xdp^+ and $\mathrm{adp}^\oplus$, as well as $\mathrm{xdp}^+(\alpha, \beta, \ldots \to \gamma)$ – the calculation of xdp^+ for more than two inputs, and the differential probability $\mathrm{xdp}^{\times C}$ of multiplication by a constant C where differences are expressed by xor.

The tool is also capable of efficiently counting the number of output differences for each of the mentioned operations. An instance where this problem occurs, is in the cryptanalysis of Threefish-512 [2], where an exponential-in-n time algorithm is proposed. Using the toolkit, this can be solved in linear time in n.

* This work was supported in part by the IAP Program P6/26 BCRYPT of the Belgian State (Belgian Science Policy), and in part by the European Commission through the ICT program under contract ICT-2007-216676 ECRYPT II.
** This author is funded by a research grant of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen).
*** DBOF Doctoral Fellow, K.U.Leuven, Belgium.
† Postdoctoral Fellow of the Research Foundation – Flanders (FWO).

The tool also provides a general algorithm to efficiently list the output differences with the highest probability, for a given type of difference and operation.

The cases handled by the toolkit, are encountered in many ARX-based cryptographic algorithms. Examples are the XTEA block cipher [3], the Salsa20 stream cipher family [4], and the hash functions MD5 and SHA-1. Other examples are 6 out of the 14 second-round candidates of NIST's SHA-3 hash function competition [5]: BLAKE [6], Blue Midnight Wish [7], CubeHash [8], Shabal [9], SIMD [10] and Skein [11]. Our tools can assist in the cryptanalysis of each of these algorithms.

References

1. Lipmaa, H., Wallén, J., Dumas, P.: On the Additive Differential Probability of Exclusive-Or. In Roy, B.K., Meier, W., eds.: FSE. Volume 3017 of Lecture Notes in Computer Science., Springer (2004) 317–331
2. Aumasson, J.P., Çagdas Çalik, Meier, W., Özen, O., Phan, R.C.W., Varıcı, K.: Improved Cryptanalysis of Skein. In Matsui, M., ed.: ASIACRYPT. Volume 5912 of Lecture Notes in Computer Science., Springer (2009) 542–559
3. Needham, R.M., Wheeler, D.J.: Tea extensions. Computer Laboratory, Cambridge University, England (1997) `http://www.movable-type.co.uk/scripts/xtea.pdf`.
4. Bernstein, D.J.: The Salsa20 Family of Stream Ciphers. In Robshaw, M.J.B., Billet, O., eds.: The eSTREAM Finalists. Volume 4986 of Lecture Notes in Computer Science. Springer (2008) 84–97
5. National Institute of Standards and Technology: Announcing Request for Candidate Algorithm Nominations for a New Cryptographic Hash Algorithm (SHA-3) Family. Federal Register **27**(212) (November 2007) 62212–62220 Available: `http://csrc.nist.gov/groups/ST/hash/documents/FR_Notice_Nov07.pdf` (2008/10/17).
6. Aumasson, J.P., Henzen, L., Meier, W., Phan, R.C.W.: SHA-3 proposal BLAKE. Submission to the NIST SHA-3 Competition (Round 2) (2008)
7. Gligoroski, D., Klima, V., Knapskog, S.J., El-Hadedy, M., Amundsen, J., Mjølsnes, S.F.: Cryptographic Hash Function BLUE MIDNIGHT WISH. Submission to the NIST SHA-3 Competition (Round 2) (2009)
8. Bernstein, D.J.: CubeHash specification (2.B.1). Submission to the NIST SHA-3 Competition (Round 2) (2009)
9. Bresson, E., Canteaut, A., Chevallier-Mames, B., Clavier, C., Fuhr, T., Gouget, A., Icart, T., Misarsky, J.F., Naya-Plasencia, M., Paillier, P., Pornin, T., Reinhard, J.R., Thuillet, C., Videau, M.: Shabal, a Submission to NIST's Cryptographic Hash Algorithm Competition. Submission to the NIST SHA-3 Competition (Round 2) (2008)
10. Leurent, G., Bouillaguet, C., Fouque, P.A.: SIMD Is a Message Digest. Submission to the NIST SHA-3 Competition (Round 2) (2009)
11. Ferguson, N., Lucks, S., Schneier, B., Whiting, D., Bellare, M., Kohno, T., Callas, J., Walker, J.: The Skein Hash Function Family. Submission to the NIST SHA-3 Competition (Round 2) (2009)

KeccakTools (abstract)*

Guido Bertoni[1], Joan Daemen[1], Michaël Peeters[2], and Gilles Van Assche[1]

[1] STMicroelectronics
[2] NXP Semiconductors

Keywords: Keccak, software tools

KeccakTools is a set of C++ classes aimed at helping analyze the sponge function family Keccak [1,2,3]. The first version of KeccakTools was released in April 2009 and provided the following features:

- the parameterized implementation of the seven Keccak-f permutations, from Keccak-$f[25]$ to Keccak-$f[1600]$, possibly with a specific number of rounds;
- the implementation of the inverses of the Keccak-f permutations;
- the generation of look-up tables for Keccak-$f[25]$;
- the generation of GF(2) equations of the round functions and step mappings in the Keccak-f permutations and their inverses;
- the generation of optimized C code for the Keccak-f round functions, including lane complementing and bit interleaving techniques;
- the implementation of the sponge construction using any transformation or permutation, and of the Keccak sponge function family.

Note that the equations can be generated in a format compatible with SAGE [5,4].

In June 2010, we released version 2.1 of KeccakTools, which adds several important classes aimed at the linear and differential cryptanalysis of Keccak-f. Essentially, these classes provide ways to represent and process linear and differential *trails*. As much as possible, linear and differential trails are considered on an equal footing, and most routines can be applied to both kinds of trails. In more details, the new classes provide the following features:

- the representation and serialization of linear and differential trails;
- for χ, the affine representation of
  - the output differences compatible with a given input difference, and
  - the input masks compatible with a given output mask;
- for the round function, the iteration through all
  - the output differences compatible with a given input difference,
  - the input differences compatible with a given output difference (possibly up to a specified weight),
  - the input masks compatible with a given output mask,
  - the output masks compatible with a given input mask (possibly up to a specified weight);
- the generation of the conditions, expressed as equations in GF(2), for a pair to follow a given differential trail.

The package includes examples of trails and an example routine that takes a trail and extends it forwards and backwards to show how to use the various classes.

The code is documented with comments in Doxygen format. The documentation can also be browsed online. Finally, the code is released in the public domain, allowing anyone to freely extend it or to adapt it to its own needs.

References

1. G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, Keccak *specifications, version 2*, NIST SHA-3 Submission, September 2009, `http://keccak.noekeon.org/`.
2. _____, Keccak *sponge function family main document*, NIST SHA-3 Submission (updated), June 2010, `http://keccak.noekeon.org/`.
3. _____, KeccakTools *software*, June 2010, `http://keccak.noekeon.org/`.
4. M. Brickenstein and A. Dreyer, *PolyBoRi: A framework for Gröbner-basis computations with Boolean polynomials*, Journal of Symbolic Computation **44** (2009), no. 9, 1326–1345, Effective Methods in Algebraic Geometry.
5. W. A. Stein et al., *Sage Mathematics Software*, The Sage Development Team, 2009, `http://www.sagemath.org/`.

* Presented at the *Workshop on Tools for Cryptanalysis*, June 22-23, 2010, Egham, UK

The CodingTool Library

Tomislav Nad

Institute for Applied Information Processing and Communications (IAIK)
Graz University of Technology, Austria
Tomislav.Nad@iaik.tugraz.at

Introduction

It has been shown in the past that tools from coding theory are very powerful in the cryptanalysis of cryptographic primitives. For instance they can be used to find differential characteristics with low Hamming weight like in the cryptanalysis of SHA-0 [3], SHA-1 [7], EnRUPT [4], CubeHash [1] or SIMD [6]. As observed by Rijmen and Oswald [8], all differential characteristics for a linearized hash function can be seen as the code words of a linear code. Algorithms for finding low Hamming weight code words in a linear code work well for finding linear differential characteristics with low Hamming weight. Our contribution is a library which should make the cryptanalysis of cryptographic primitives a little easier. The library implements a search algorithm, interfaces, data structures and several other useful functionalities. The abstraction level is high, so that a cryptanalyst does not have to care about complicated implementation details.

The Library

The CodingTool library is a new collection of tools to use techniques from coding theory in cryptanalysis. It is completely independent from other libraries and can be used on Unix and Windows platforms. It benefits from the 64-bit architecture in terms of speed. The core part is an implementation of the probabilistic algorithm from Canteaut and Chabaud [2] to search for code words with low Hamming weight. Additional functionalities like shortening and puncturing of a linear code or adding a weight to each bit of a code word are implemented. Furthermore, the library provides data structures to assist the user in creating a linear code for a specific problem. An easy to use interface to the provided algorithms, powerful data structures and a command line parser reduces the implementation work of a cryptanalyst to a minimum. Beside the existing functionality, the library can be extended very easily. A possible improvement is the implementation of faster search algorithms or the improvement of the existing one.

The complete library is under the GPL 3.0 license. The provided archive consists of the source code, documentation, examples and precompiled binaries.

Example

To demonstrate some of the functionalities we picked the SHA-1 message expansion. Jutla and Patthak showed [5] that the minimum Hamming weight for the last 60 words of the SHA-1 message expansion is 25. We show how one can use the library to build the linear code, search for low Hamming weights and force specific bits to zero for this kind of problem.

References

1. Eric Brier, Shahram Khazaei, Willi Meier, and Thomas Peyrin. Linearization framework for collision attacks: Application to cubehash and md6. Cryptology ePrint Archive, Report 2009/382, 2009. urlhttp://eprint.iacr.org/.
2. Anne Canteaut and Florent Chabaud. A New Algorithm for Finding Minimum-Weight Words in a Linear Code: Application to McEliece's Cryptosystem and to Narrow-Sense BCH Codes of Length 511. *IEEE Transactions on Information Theory*, 44(1):367–378, 1998.

3. Florent Chabaud and Antoine Joux. Differential Collisions in SHA-0. In Hugo Krawczyk, editor, *CRYPTO*, volume 1462 of *Lecture Notes in Computer Science*, pages 56–71. Springer, 1998.
4. Sebastiaan Indesteege and Bart Preneel. Practical Collisions for EnRUPT. In Orr Dunkelman, editor, *FSE*, volume 5665 of *Lecture Notes in Computer Science*, pages 246–259. Springer, 2009.
5. Charanjit S. Jutla and Anindya C. Patthak. A matching lower bound on the minimum weight of sha-1 expansion code. Cryptology ePrint Archive, Report 2005/266, 2005. `http://eprint.iacr.org/`.
6. Florian Mendel and Tomislav Nad. A distinguisher for the compression function of simd-512. In Bimal K. Roy and Nicolas Sendrier, editors, *INDOCRYPT*, volume 5922 of *Lecture Notes in Computer Science*, pages 219–232. Springer, 2009.
7. Norbert Pramstaller, Christian Rechberger, and Vincent Rijmen. Exploiting Coding Theory for Collision Attacks on SHA-1. In Nigel P. Smart, editor, *IMA Int. Conf.*, volume 3796 of *Lecture Notes in Computer Science*, pages 78–95. Springer, 2005.
8. Vincent Rijmen and Elisabeth Oswald. Update on SHA-1. In Alfred Menezes, editor, *CT-RSA*, volume 3376 of *Lecture Notes in Computer Science*, pages 58–71. Springer, 2005.

Grain of Salt — An Automated Way to Test Stream Ciphers through SAT Solvers

Mate Soos

Paris LIP6, INRIA Rocquencourt

Abstract. In this paper we describe Grain of Salt, a tool developed to automatically test stream ciphers against standard SAT solver-based attacks. The tool takes as input a set of configuration options and the definition of each filter and feedback function of the stream cipher. It outputs a problem in the language of SAT solvers describing the cipher. The tool can automatically generate SAT problem instances for Crypto-1, HiTag2, Grain, Bivium-B and Trivium. In addition, through a simple text-based interface it can be extended to generate problems for any stream cipher that employs shift registers, feedback and filter functions to carry out its work.

1 Introduction

SAT solvers have recently been enjoying a boom in the area of cryptanalysis. It has been shown in multiple papers [1,2,3] that SAT solvers are indeed a viable technique in algebraic cryptanalysis to both analyse and potentially break stream or block ciphers. SAT solvers work with problems described in Conjunctive Normal Form (CNF), but obtaining such a problem description is non-trivial. Essentially all works that aimed to analyse a cipher through SAT solvers have developed a way to convert descriptions of ciphers to their CNF form.

In this paper we present Grain of Salt (GoS), a tool that generates optimised CNFs given the description of a stream cipher. It is aimed to be flexible and easy to use, helping the cryptanalyst obtain the best results within the least amount of time. The tool comes loaded with the descriptions of ciphers Crypto-1 [4], HiTag2 [5], Trivium [6], Bivium-B [7], and Grain [8], but can be easily extended with any stream cipher that uses shift registers, feedback functions and filter functions to carry out its work. The tool is designed to be intuitive to use and general enough to cover a large set of different ciphers while remaining specific enough to address the optimisations possible for the SAT-based cryptanalysis of many stream ciphers.

The rest of this paper is structured as follows. In Sect. 2 we give some background on SAT solvers and SAT-based cryptanalysis. Then, in Sect. 3 we present the input format that GoS uses to describe ciphers. In Sect. 4 we present the various features that GoS offers, and in Sect. 5 we shortly describe the timing results possible with the use of the GoS tool. Finally, in Sect. 6 we conclude this paper.

2 Background

In this section we give a short description of SAT solvers and their use in cryptanalysis.

2.1 SAT solvers

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

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

Most DPLL-based SAT solvers understand and deal with problems described in CNF. Usually, a non-trivial part of using SAT solvers is to convert the problem at hand to CNF format. The CNF can then be given to many different SAT solvers, and an appropriate one (e.g. fastest, distributed, etc.) can be selected.

2.2 SAT Solver-based cryptanalysis

SAT solver-based algebraic cryptanalysis have successfully been applied to break a number of ciphers secure against other forms of cryptanalysis. The first SAT solver-based algebraic cryptanalysis was by Massacci et al. [9], experimenting with the Data Encryption Standard (DES) using DPLL-based SAT solvers. More recent work by Courtois and Bard has produced attacks against KeeLoq [10] and investigated DES [1]. SAT solver-based algebraic cryptanalysis has also been effectively used on modern stream ciphers, such as the reduced version of Trivium, Bivium-B [3] by Soos et al.

In parallel to the above mentioned papers, there have been multiple tools developed that convert cryptographic functions to CNF. Among them is the python module developed by Martin Albrecht for the `sage` mathematics platform [11], `Logic2CNF` developed by Edd Barrett [12], and `STP` (Simple Theorem Prover)

by Ganesh et al. [13]. These tools offer widely different features and can be used at different levels of abstraction. For instance, `Logic2CNF` only converts a description of the cipher in Algebraic Normal Form (ANF) to CNF, but cannot generate the ANF given a cipher description. `STP` can parse a complete cipher description but does not retain or deal with the ANF form of the description, thus omitting optimisations possible at that level of abstraction. Finally, the `sage` module only converts to CNF a description that has already been described in `sage`, but can use the tools provided by `sage` to process (and simplify) the problem at the ANF level.

Most of the above mentioned papers and tools implement their own way of describing the cipher in CNF, inventing or re-inventing methods on the way. A well-known reference is the paper by Bard et al. [14] which describes some starting points for the conversion, but individual conversion methods vary widely. Grain of Salt tries to merge the ideas from these papers and tools into one, easy-to-use package.

3 The input to GoS

The input to GoS describes a stream cipher in terms of shift registers, feedback, filter and output functions. There are two phases for each attack, the initialisation phase, and the standard running phase. Accordingly, there are two feedback functions associated with each shift register: one that operates during initialisation, and one that operates during normal operation. These feedback functions are often different, as is the case with Crypto-1, Grain and Trivium. Filter functions are always calculated, and their outputs can be used at any point in time by any of the functions, including other filter functions, thus forming a chain. This is important for ciphers such as Crypto-1, where there are multiple micro-filter functions that make up the final output (when in normal mode) and the feedback (when in initialisation mode). The output function is simply a specially designated filter function that produces the output, active only during the normal phase.

Let us now take Grain as an example cipher, and describe it in GoS. Grain has two shift registers, both 80 bits long and its initialisation phase has 160 steps. The main configuration file for this cipher is `grain/config`, and looks as follows:

```
sr_size = 80,80                          (1)
linearizable_sr_during_init =            (2)
linearizable_sr_during_norm = 1          (3)
filters = 1                              (4)
init_clock = 160                         (5)
tweakable = sr1-0...63                   (6)
one_out = sr1-64...79                    (7)
```

Line `(1)` tells that there are two shift registers, numbered `sr0` and `sr1`. Line `(2)` means that none of the shift registers' feedback functions are linearizeable during the initialisation phase — i.e. their feedback functions are non-linear. Line

`(3)` says that during normal operation, shift register `sr1`'s state is linearizeable, as according to the Grain specification [8], the second shift register is an LFSR. Line `(4)` says that the number of filter functions used is one, called `f0`. The function `f0` models the complex filter that is used during both the initialisation and the normal phase of the cipher. Line `(5)` says that the initialisation takes 160 cycles. Line `(6)` says that the first 64 bits of the second shift register is the IV, i.e. these bits are tweakable (can be freely chosen). Finally, line `(7)` says that the last 16 bits of the second shift register must be filled with binary ones.

3.1 Initialisation phase

The Grain cipher has two phases: the initialisation phase and the normal running phase. Each shift register has to have a feedback function associated with it for each phase. The files describing these functions for Grain must be under the directory `grain/functions/srX/`, where `X` is the number of of the shift register (`0` or `1` in case of Grain). The feedback of `sr0` during initialisation is described in the file `grain/functions/sr0/feedback_init.txt` shown in Fig. 1(a), which corresponds to the line in the Grain specification file

$$\begin{aligned} b_{i+80} =& s_i + b_{i+62} + b_{i+60} + b_{i+52} + b_{i+45} + b_{i+37} + b_{i+33} + b_{i+28} + b_{i+21} + \\ &+ b_{i+14} + b_{i+9} + b_i + b_{i+63}b_{i+60} + b_{i+37}b_{i+33} + b_{i+15}b_{i+9} + \\ &+ b_{i+60}b_{i+52}b_{i+45} + b_{i+33}b_{i+28}b_{i+21} + b_{i+63}b_{i+45}b_{i+28}b_{i+9} + \\ &+ b_{i+60}b_{i+52}b_{i+37}b_{i+33} + b_{i+63}b_{i+60}b_{i+21}b_{i+15} + \\ &+ b_{i+63}b_{i+60}b_{i+52}b_{i+45}b_{i+37} + b_{i+33}b_{i+28}b_{i+21}b_{i+15}b_{i+9} + \\ &+ b_{i+52}b_{i+45}b_{i+37}b_{i+33}b_{i+28}b_{i+21} \end{aligned}$$

The last line of the file (containing "`f0`") in Fig. 1(a) cannot be found in this equation since the filter (modelled with `f0` in our case) must be XOR-ed into the feedback during the initialisation phase. This filter function is defined in `grain/functions/f0.txt`, present in Fig. 2, which corresponds to the following set of definitions in the Grain specification:

$$\begin{aligned} z_i =& \sum_{k \in A} b_{i+k} + h(s_{i+3}, s_{i+25}, s_{i+46}, s_{i+64}, b_{i+63}) \\ A =& \{1, 2, 4, 10, 31, 43, 56\} \\ h(x) =& x_1 + x_4 + x_0x_3 + x_2x_3 + x_3x_4 + x_0x_1x_2 + x_0x_2x_3 + x_0x_2x_4 + x_1x_2x_4 + x_2x_3x_4 \end{aligned}$$

The feedback function of the second filter function during initialisation is present in file `grain/functions/sr1/feedback_init.txt`, present in Fig. 1(b), which corresponds to the line in the Grain specification

$$s_{i+80} = s_{i+62} + s_{i+51} + s_{i+38} + s_{i+23} + s_{i+13} + s_i$$

which again is missing the `f0`, since the authors of the paper only specified the initialisation phase later, in Sect. 2.1. Here, they make it clear that the filter function, described by `f0` in our case, needs to be XOR-ed to this feedback function during the initialisation phase.

```
sr1-0
sr0-62
sr0-60
sr0-52
sr0-45
sr0-37
sr0-33
sr0-28
sr0-21
sr0-14
sr0-9
sr0-0
sr0-63 sr0-60
sr0-37 sr0-33
sr0-15 sr0-9
sr0-60 sr0-52 sr0-45
sr0-33 sr0-28 sr0-21
sr0-63 sr0-45 sr0-28 sr0-9
sr0-60 sr0-52 sr0-37 sr0-33
sr0-63 sr0-60 sr0-21 sr0-15
sr0-63 sr0-60 sr0-52 sr0-45 sr0-37
sr0-33 sr0-28 sr0-21 sr0-15 sr0-9
sr0-52 sr0-45 sr0-37 sr0-33 sr0-28 sr0-21
f0
```

(a) Feedback of the NLFSR

```
sr1-62
sr1-51
sr1-38
sr1-23
sr1-13
sr1-0
f0
```

(b) Feedback of the LFSR

Fig. 1. Feedback functions of Grain used during the initialisation phase. The left-hand figure is stored in the file `grain/functions/sr0/feedback_init.txt` while the right-hand figure is stored in the file `grain/functions/sr1/feedback_init.txt`

3.2 Normal phase

The feedbacks during normal running look exactly like the feedbacks for initialisation, except for the last lines: the filter function is not XOR-ed in, instead it is simply output as the keystream. Therefore the file describing the feedback of the NLFSR during normal operation, `grain/functions/sr0/feedback.txt`, is exactly the same as that present in Fig. 1(a), with the exception of the last line, `f0`. Similarly, the file describing the feedback of the LFSR during normal operation is missing the `f0`. Finally, the file that specifies the keystream, `grain/functions/output0.txt`, contains just one line with `f0`, signifying that it is equal to the filter function `f0`.

3.3 Composition of filters

Filter functions, such as `f0` for Grain can be used extensively in the function descriptions of the cipher. They can also be combined to create some interesting

```
sr0-1
sr0-2
sr0-4
sr0-10
sr0-31
sr0-43
sr0-56
sr1-25
sr0-63
sr1-3 sr1-64
sr1-46 sr1-64
sr1-64 sr0-63
sr1-3 sr1-25 sr1-46
sr1-3 sr1-46 sr1-64
sr1-3 sr1-46 sr0-63
sr1-25 sr1-46 sr0-63
sr1-46 sr1-64 sr0-63
```

Fig. 2. File that describes the filter function for Grain, stored in the file `grain/functions/f0.txt`

effects. For instance, the output of the Crypto-1 cipher is generated using a set of mini-filter functions as present in Fig. 3(b). In the case of Crypto-1, it is best not to describe the final feedback function as one big function, but to preserve the structure of the mini-filter functions. To achieve this, we can define `f0`...`f4`, similarly to how we defined filters in Grain, and define the output in `crypto1/functions/output0.txt`, present in Fig. 3(a), as a combination of the internal filter functions.

4 Features offered

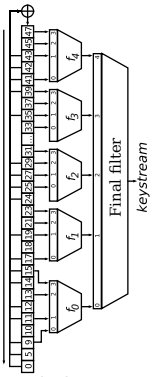
The GoS tool offers multiple features to help analyse the stream cipher. We list the most important features here.

4.1 Variable number of generated output bits

The number of output bits generated and given to the solver as the base of solving can be chosen at will. The command line switch for this option is `--outputs NUM`, where `NUM` is a number that should be sufficient to fully determine the searched-for data. For example, if the initialisation is used for Grain, the number of output bits needed should be at least 80. However, if the initialisation is not used, then at least 160 bits are needed, since the solver has to solve for 160 bits of unknowns (the full state of both shift registers) in that case.

```
f0
f2 f0
f3 f0
f3 f1 f0
f3 f2 f1
f4
f4 f0
f4 f1 f0
f4 f2 f1 f0
f4 f3
f4 f3 f0
f4 f3 f1
f4 f3 f2 f1
```

(a) The output of Crypto-1, described as a function of mini-filter functions `f0`...`f4`.



(b) The functional diagram of the Crypto-1 cipher

Fig. 3. The Crypto-1 cipher (on the right), and the description of its final filter function (on the left), made up of multiple micro-functions. The network of micro-functions is clearly visible in the functional diagram, and is replicated in Grain-of-Salt with the use of multiple filter functions.

4.2 ANF generation with fact propagation

An Algebraic Normal Form of the described cipher with the given number of parameters (described below) can be generated. Various statistical data on the ANF can also be obtained, such as the size (number of monomials) of the each function, the sum size of all functions, etc.

We call fact propagation the effect of evaluating all equations with respect to the given information. The given information can be the output of the cipher or other form of helping information. The evaluations might, for instance, cause a variable to be set instantly, for example, if $a = bc \oplus d$ and $b = \texttt{false}, d = \texttt{true}$ then $a = \texttt{true}$, and by substituting this fact into other equations, further facts could be found. GoS automatically handles this, and recursively propagates all such facts.

Under the aegis of fact propagation GoS also propagates variable equivalences. For example, if $a = bc$ and $c = \texttt{true}$ then $a = b$, which might lead to further facts. For example, the equation $d = b \oplus ab$ would be changed to $d = b \oplus bb = b \oplus b = \texttt{false}$ allowing the propagation of a further fact. Fact and variable equivalence propagation considerably shortens problems, which help when they need to be solved using the SAT solver.

4.3 CNF Generation

GoS automatically generates CNF from the fact-propagated ANF using a variety of mechanisms to optimise the conversion. There are mainly two ways of converting an equation in ANF to CNF:

1. Through cutting long XOR-s, and introducing internal variables for each monomial of degree > 1.
2. Through the use of a Karnaugh map generator [15]. Karnaugh maps essentially directly generate the CNF from a truth table, needing no conversion. This method was first used in converting cryptographic ANFs by Soos et al. [3].

Deciding which method to use is non-trivial, and GoS can be given a heuristic cut-off that decides which method to use. The cut-off is given with the command-line parameter `--karnaugh NUM` where if more than `NUM` monomials are present in an ANF, the first method is used, while if less or equal, the second method is used to convert to CNF. Essentially, the first method is relatively straight-forward, but can generate very non-optimal representation if the number of monomials is small, their average degree is high and they make use of a small number of variables. For example, the Crypto-1 and HiTag2 ciphers' mini filter functions all fall into this category, and they are best represented as such. However, if for example the degree is low, then the Karnaugh map representation is uniquely non-optimal, as Karnaugh maps behave the worst (exponentially) for XOR functions, and they also behave very badly with near-XOR functions.

The straight ANF-to-CNF method of simply converting the XOR to CNF and then introducing internal variables for monomials of degree > 1 is done as follows. Long XOR-s must be cut due to the exponential nature of their conversion: an n-long XOR can only be represented (without introduction of internal variables) as 2^{n-1} clauses. To overcome this, XOR-s are cut such as:

$$\begin{aligned} a \oplus b \oplus c \oplus d \oplus e \oplus f &= \texttt{true} \leftrightarrow \\ a \oplus b \oplus c \oplus i &= \texttt{false} \\ d \oplus e \oplus f \oplus i &= \texttt{true} \end{aligned}$$

but the best limit at which XOR-s must be cut, which is usually called the *cutting number* is not easy to determine. The default is 7 in GoS, but can be changed with `--xorcut NUM`. Monomials are expressed in the CNF language through the introduction of internal variables. For example, the monomial ab is expressed as $i_2 = ab$, leading to the clause-set:

$$\begin{aligned} \neg i_2 \vee b \\ \neg i_2 \vee a \\ i_2 \vee \neg a \vee \neg b \end{aligned}$$

An optimisation for the straight ANF-to-CNF conversion is that monomials in the CNF world can contain negations, i.e. it is no longer necessary to write $a \oplus ab$, since that can be simply written as $a(1 + b) = a\neg b$. This optimisation

can be applied recursively. For example:

$$\begin{aligned} a \oplus b \oplus ab \oplus c \oplus cd &= \texttt{true} \leftrightarrow \\ b \oplus a(1 \oplus b) \oplus c(1 \oplus d) &= \texttt{true} \leftrightarrow \\ 1 \oplus (1 \oplus a)(1 \oplus b) \oplus c(1 \oplus d) &= \texttt{true} \leftrightarrow \\ \neg a \neg b \oplus c \neg d &= \texttt{false} \end{aligned}$$

reducing the original 5 monomials into a mere two. Representing these monomials that are not free of negations takes exactly the same amount of resources in CNF as representing those that are free of negations, leading to a potential overall reduction in the final CNF. The reduction is only potential, as the internal variables that represent monomials that are used in multiple places need only be described once, and the extended monomials could possibly make it more difficult for the same monomials to appear, limiting their benefits. Therefore, this optimisation can be turned off with the command-line switch `--noextmonomials`. The default is for this optimisation to be turned on, as we have experienced speedups using it.

4.4 Dependency tree generation

It is assumed that only the output of the stream cipher is known to the attacker. Therefore, functions that are not connected in some way to the output of the function can be discarded: they are internal variables that need not be calculated, since they cannot help solving the internal state. To remove these functions, a dependency tree is generated that takes as root all the output bits of the cipher, and generates a tree that reaches the original internal state bits. All functions that are not connected to this tree can be discarded.

Dependency tree generation is very important, as it lets the designer describe as many filter functions as he or she wishes: the functions that are not used will not hamper the solving. For example, some ciphers use filter functions that are specific to the initialisation phase. Without dependency tree generation, these filter functions would be calculated (but not used) during the normal running phase, slowing down the solver.

4.5 Solving with and without initialisation phase

The GoS tool has two running modes. It can either try to solve for the non-tweakable and non-oned-out parts of the cipher when initialisation is turned on, or it can solve for the entire state when initialisation is turned off. In other words, the two typical scenarios are covered: either the IV is known, the key is unknown, and the initialisation is carried out, or the entire state of the cipher is unknown, but the initialisation is not carried out. This behaviour can be simply switched using a command line switch `--init yes` or `--init no`.

Typically, with the initialisation turned on, the number of bits to be solved is much less. For example, in the case of Grain, the IV is 64, and the one-ed out part

is 16 bits, so only 80 bits of the first shift register (i.e. the key) is the unknown. However, the initialisation takes 160 cycles, which greatly increases the difficulty of the resulting set of equations. On the other hand, without initialisation, the number of unknown bits increases to 2*80 = 160, but since initialisation is not carried out, most equations are very short.

4.6 Base shifting

Base shifting can be activated when solving without initialisation. Base shifting is the name we use for the technique first presented in [3, Sect. 4.3]. There, the authors show that the base unknown of the cipher can be *any* moment in time. So, for example, if the number of output bits generated is 200, and no initialisation is used, then the cipher is clocked for 100 bits, with a total length of 100 + 80 = 180 (since 80 is the original size of each). Any consecutive 80 bit frame of this can be taken as the unknown, as the feedback functions can be re-arranged to clock backwards for these ciphers. This is very advantageous, as typically, the complexity increases exponentially starting from a point T, and if we take T to be near the middle of the time-frame (e.g. at $T = 90$ in our example), then the total complexity of the generated functions are much less than if we had taken the typical approach, i.e. to take $T = 0$.

The command line parameter for base shifting is `--base-shift NUM` where `NUM` must be smaller or equal to the number of output bits. For this to work with `NUM` > 0, the feedback functions of the cipher must be reversible. This is true for Crypto-1, HiTag2, Bivium-B, Trivium, and Grain. Stream ciphers can be created where this is not the case — these stream cipher are, however, usually constrained in that it is hard to make them work faster through parallel implementation of the feedback and filter functions in hardware.

As a concrete example, let us take the Grain cipher, without the initialisation phase switched off. If the number of output bits generated is 200, the base shifting can be any number between 0 and 200. For a shifting number x, the unknowns are the states of the shift registers at time x. In other words, if we take each shift register as a memory line that does not forget its old contents, then the unknowns are the state variables $x \dots x + 80$ of both shift registers. We call these variables the *reference state variables*. When initialisation is turned on, the reference state variables are simply the variables that are neither tweakable nor one-ed out, i.e. they are the state variables where (typically) the key is loaded.

4.7 Help bit calculation

Help bits are data pieces that are given such that it is easier to solve for the state of a cipher. These are important, as it is infeasible to wait immense amounts of time to check whether, for example, the state of Grain can be solved. In order to circumvent this problem, we give some reference state variables as *help bits* to the solver, such that it can solve faster. Once the solving has finished, one can estimate the time it would take to solve for the whole state of the cipher, without the help bits. The GoS tool offers two types of help bit calculations. One

is a probabilistic calculator, and the other is a deterministic calculator, and both employ a Monte-Carlo method to achieve their goals. For the following sections, let us assume that V the possible set of variables that can be help bits (i.e. V contains exactly the reference state variables).

The Monte-Carlo method, first introduced by Metropolis and Ulam [16] is used in many areas of research such as integration and computer security (e.g. the Rabin primality test [17]). It is essentially a randomised algorithm that samples a tiny part of the possibly immense space and processes the results to approximate an unknown value for the whole space. In case of the Rabin primality test, the Monte-Carlo algorithm uses a randomised test to decide if a positive integer is a prime or not. The algorithm has a certain chance ($< 1/4$) to give a false negative result, but running the algorithm many times essentially eliminates the chance that a number is composite.

Deterministic method Since using different reference state variables as help bits could give different timings, it is non-trivial which ones to use. To achieve maximum performance, we use an approach that we have found to be adequate to find a good set.

We first generate the ANF that describes the cipher given all settings. Then, we set a variable $v \in V$, $v \leftarrow$ `true`, and *propagate* all changes. We count the number of monomials in the resulting ANF. Then, we set $v \leftarrow$ `false`, and again count the size of the resulting ANF. The sum of these two values is the "score" for this help bit. We perform these steps for each variable in L, and the one that has the smallest score wins. We now put this winning variable into the ordered set H, and continue the search as follows.

Let us call L the possible set of variables that can be help bits (i.e. L contains exactly the reference state variables). We take a variable $v \in L$
H and randomly set all variables in H, plus we set $v \leftarrow$ `true`, and count the score. We do ten such measures, each time setting the variables in H randomly, and sum the scores. Then, we do the same, but with $v \leftarrow$ `false`, and sum the scores. The sum of these 20 measurements will be the score for this v. We do this for all $v \in L$
H: the variable with the smallest score wins and enters H. At the end of the algorithm, we reach a point where L
$H = \emptyset$, and all variables have been ordered in H.

The presented algorithm is a randomised greedy algorithm that tries to find a local minima at each point. Since even a local minima is very difficult to find, the algorithm probabilistically tries to find this local minima, through 20 random tests. For better local minima finding, the number 20 can be increased to any even number, ameliorating the algorithm.

We have found this algorithm to be very powerful in reducing the time to solve a given cipher. Without such ordering of bits, the speed to solve a certain problem can be hundreds of times more difficult. The output of this algorithm is simply put a file called "best-bits", and the variable numbers are simply listed

one after the other. If the cryptographer knows a better ordering, this file can simply be overwritten.

Once the "best bits" file has been generated, it can be used from the program by giving the option `--deterBits NUM`, where `NUM` is the number of best bits the program should set randomly when generating the problem instances. Averaging the time it takes to solve these problems and multiplying the average by $2^{\texttt{NUM}}$ one gets the amount of expected time to solve the cipher.

The program specifically does not include a method to break a cipher, though given a specific cipher output, it could generate all $2^{\texttt{NUM}}$ possible problems. Naturally, one of the generated problems would actually break the given output stream, revealing the key or the state of the cipher (depending on whether initialisation was enabled or not).

Probabilistic method The probabilistic method is activated with the command line switch `--probBits NUM` and it simply randomly sets a random set of `NUM` variables from L and does many runs of these random configurations. The time it takes to solve these randomly picked instances is then averaged.

To approximate the time it takes to attack the cipher without giving any variables we use the following technique. We run many instances of the above algorithm with $\texttt{NUM} = n, n-1, \ldots n-k$ number of reference state variables, where n is small enough such that the algorithm is not trivial to solve, and $n-k$ is as small as possible such that the resulting system is still solved within a reasonable amount of time. The average time is then plotted against the number of reference state variables given, and the plot is extrapolated to the point where there are no reference state variables given.

Although there is no proof that at any point during $\texttt{NUM} = n-k-1 \ldots 0$ the graph does not suddenly change, we believe this to be extremely unlikely. For the explication of the reasons, let us first define some notions. Let us define two problems for a given cipher: problem A is when $\texttt{NUM} = x$, and problem B is where $\texttt{NUM} = x-1$, where $n \geq x > 0$ but otherwise x is irrelevant. Let us assume, without loss of generality, that V' is the set of reference variables selected to be assigned in B. Let the set of reference variables assigned in A be $V' \bigcup v$. We can now list the reasons why we believe the graph does not deviate from a straight line if the time is plotted in a logarithmic scale:

- Every problem in B can be directly mapped to $2|V \setminus V'| = 2(n-x+1)$ problems in A. Since the underlying algorithm of DPLL-based SAT solvers is essentially an intelligent brute-force, we can safely assume it does not behave worse than a brute-force, and solves the problem B in at most twice the time than solving any problem in A. This is further underlined by our observation that SAT solvers branch on the reference state variables — thus the first branching of the solver when solving B will indeed be a variable from $|V \setminus V'|$
- The more choice of variables a SAT solver has to branch on, the better the dynamic variable branch ordering will work. This means that it is expected of the solver to solve in less than twice the time problem B with a choice of

$n - x$ branch variables, than two problems in A with a choice of $n - x - 1$ branch variables
- Clauses learnt during the solving of A that are independent of the setting of variable v cannot be reused between the solving instances. Therefore, it is expected that problem B can be solved faster than two problems in A, as the solver in the former case does not need to re-learn these same clauses
- The underlying problem structure does not change between problem A and problem B
- It is the same underlying randomised solving algorithm that is used to solve both problem A and problem B

The extrapolation is usually straightforward if a large enough number of randomisation steps are involved: the plotted graph is straight if plotted against a logarithmic time. We note that the possibility of extrapolation is an advancement over previous attempts. Previous attempts failed, as they did not introduce sufficient randomness into the system. This lack of suitable randomisation meant that their results were not extrapolateable [18,2].

5 Results

GoS in conjunction with an appropriate SAT solver such as CryptoMiniSat [19] can be used to break Crypto-1 in 40 s, HiTag2 in $2^{14.5}$ s, and Bivium-B in an approximated $2^{36.5}$ s using a a Xeon E5345@2.33GHz computer. All these figures are faster than exhaustive search, leading to the breaking of these algorihtms. In the literature we have not found any results that indicated a faster solving time for these ciphers using a SAT-based cryptanalsysis, and so we believe these figures to be the current state-of-the-art.

6 Conclusions

We have presented Grain of Salt, an integrated package to test stream ciphers against SAT solver-based attacks. The tool can flexibly generate with a minimum of user intervention a CNF representation of any shift-register based stream cipher, helping the researcher evaluate the cipher against SAT solver-based algebraic attacks. The input language and the command line options of the tool are easy to use and user-friendly, helping the novice as well as the advanced users to profit from the tool. We envision that Grain of Salt will be further extended by researchers to carter for their specific needs, making the tool more diverse and more useful for the whole of the research community.

Acknowledgements

The author was supported by the RFID-AP ANR Project, project number `ANR-07-SESU-009`. I would like to thank Karsten Nohl for some initial ideas, notably base-shifting.

References

1. Courtois, N.T., Bard, G.V.: Algebraic cryptanalysis of the Data Encryption Standard. In Galbraith, S.D., ed.: IMA Int. Conf. Volume 4887 of Lecture Notes in Computer Science., Springer (2007) 152–169
2. Eibach, T., Pilz, E., Völkel, G.: Attacking Bivium using SAT solvers. In Büning, H.K., Zhao, X., eds.: SAT. Volume 4996 of LNCS., Springer (2008) 63–76
3. Soos, M., Nohl, K., Castelluccia, C.: Extending SAT solvers to cryptographic problems. In Kullmann, O., ed.: SAT. Volume 5584 of Lecture Notes in Computer Science., Springer (2009) 244–257
4. Garcia, F.D., de Koning Gans, G., Muijrers, R., van Rossum, P., Verdult, R., Schreur, R.W., Jacobs, B.: Dismantling MIFARE Classic. In Jajodia, S., López, J., eds.: ESORICS. Volume 5283 of Lecture Notes in Computer Science., Springer (2008) 97–114
5. Courtois, N., O'Neil, S., Quisquater, J.J.: Practical algebraic attacks on the HiTag2 stream cipher. In Samarati, P., Yung, M., Martinelli, F., Ardagna, C.A., eds.: ISC. Volume 5735 of Lecture Notes in Computer Science., Springer (2009) 167–176
6. Cannière, C.D.: Trivium: A stream cipher construction inspired by block cipher design principles. In Katsikas, S.K., et al, eds.: ISC. Volume 4176 of LNCS., Springer (2006) 171–186
7. Raddum, H.: Cryptanalytic results on Trivium. Technical Report 2006/039, ECRYPT Stream Cipher Project (2006)
8. Hell, M., Johansson, T., Meier, W.: Grain: a stream cipher for constrained environments. IJWMC **2**(1) (2007) 86–93
9. Massacci, F., Marraro, L.: Logical cryptanalysis as a SAT-problem: Encoding and analysis. Journal of Automated Reasoning **24** (2000) 165–203
10. Courtois, N., Bard, G.V., Wagner, D.: Algebraic and slide attacks on KeeLoq. In Nyberg, K., ed.: FSE. Volume 5086 of Lecture Notes in Computer Science., Springer (2008) 97–115
11. The SAGE Group: SAGE mathematics software (2008) http://www.sagemath.org.
12. Barrett, E.: Logic2CNF logic solver and converter (March 2010) http://projects.cs.kent.ac.uk/projects/logic2cnf/.
13. Ganesh, V., Dill, D.L.: A decision procedure for bit-vectors and arrays. In Damm, W., Hermanns, H., eds.: CAV. Volume 4590 of Lecture Notes in Computer Science., Springer (2007) 519–531
14. Bard, G.V., Courtois, N.T., 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)
15. Karnaugh, M.: The map method for synthesis of combinational logic circuits. Transactions of American Institute of Electrical Engineers part I **72**(9) (November 1953) 593–599
16. Metropolis, N., Ulam, S.: The Monte Carlo method. Journal of the American Statistical Association **44**(247) (1949) 335–341
17. Rabin, M.O.: Probabilistic algorithm for testing primality. J. Number Theory **12**(1) (1980) 128–138
18. McDonald, C., Charnes, C., Pieprzyk, J.: Attacking Bivium with MiniSat. Technical Report 2007/040, ECRYPT Stream Cipher Project (2007)
19. Soos, M.: CryptoMiniSat — a SAT solver for cryptographic problems (2009) http://planete.inrialpes.fr/~soos/CryptoMiniSat2/index.php.

Analysis of Trivium by a Simulated Annealing Variant

Julia Borghoff[1], Lars R. Knudsen[1], and Krystian Matusiewicz[2]

[1] Department of Mathematics, Technical University of Denmark
{J.Borghoff, Lars.R.Knudsen}@mat.dtu.dk
[2] Institute of Mathematics and Computer Science, Wroclaw University of Technology
Krystian.Matusiewicz@pwr.wroc.pl

Abstract. This paper proposes a new method of solving certain classes of systems of multivariate equations over the binary field and its cryptanalytical applications. We show how heuristic optimization methods such as hill climbing algorithms can be relevant to solving systems of multivariate equations. A characteristic of equation systems that may be efficiently solvable by the means of such algorithms is provided.
As an example, we investigate equation systems induced by the problem of recovering the internal state of the stream cipher Trivium. We propose an improved variant of the simulated annealing method that seems to be well-suited for this type of system and provide some experimental results.

Keywords. simulated annealing, cryptanalysis, Trivium

1 Introduction

Cryptanalysis focuses on efficient ways of exploiting, perhaps unexpected, structure of cryptographic problems. It could be a difference which propagates with a high probability through the cipher as used in differential cryptanalysis [6, 2] or a linear approximation of the non-linear parts of a cipher that holds for many of the possible inputs as it is the case with linear cryptanalysis [20].

More recently, the so-called algebraic attacks have received much attention. They exploit the fact that many cryptographic primitives can be described by sparse multivariate non-linear equations over the binary field in such a way that solving these equations recovers the secret key or the initial state in the case of stream ciphers. In general, solving random systems of multivariate non-linear Boolean equations is an NP-hard problem [12]. However, when the system has a specific structure, we can hope that more efficient methods may exist.

One technique to tackle such equation systems is linearisation, where each non-linear term is replaced by an independent linear variable. It works only if there are enough linear independent equations in the resulting system. Courtois et al [7] proposed the XL algorithm which increases the number of equations by multiplying them with all monomials of a certain degree. It has been refined to the XSL algorithm [9], which, when applied to the AES, exploits the special structure of the equation system. Neither the XL nor the XSL algorithm have been able to break AES but algebraic attacks were successful in breaking a number of stream cipher designs [8, 1].

In this paper we also investigate systems of sparse multivariate equations. The important additional requirement we make is that each variable appears only in a very limited number of equations. The equation system generated by the key stream generation algorithm of the stream cipher Trivium [10] satisfies those properties and

will be examined in this paper as our main example. The fully determined Trivium systems consists of 954 equations in 954 variables. Solving this system allows us to recover the 288-bit initial state.

Our approach considers the problem of finding a solution for the system as an optimization problem and then applies an improved variant of simulated annealing to it. As opposed to the XL and XSL algorithms, the simulated annealing algorithm does not increase the size of the problem, it does not generate more nor change the existing equations. The only additional requirement is an objective function, called the cost function, that should be minimized.

Simulated annealing has been studied in the context of cryptography before. Knudsen and Meier [19] presented an attack on an identification scheme based on the permuted perceptron problem (PPP). They found an appropriate cost function which enabled them to solve the simpler perceptron problem as well as the PPP using a simulated annealing search. The attack showed that the recommended smallest parameters for the identification scheme are not secure. The same identification scheme was later a subject to an improved attack by Clark and Jacob [5]. They used simulated annealing to solve a related problem that had solutions highly correlated with the solution of the actual problem. They also made use of timing analysis where the search process is monitored and one can observe that some variables are stuck at correct values at an early state and never change again.

With our current experiments, we are not able to break Trivium in the cryptographic sense which means with a complexity equivalent to less than 2^{80} key setups and the true complexity of our method against Trivium is unknown. However, if we consider the Trivium system purely as a multivariate quadratic Boolean system in 954 variables then we are able to solve the system significantly faster than brute force, namely in around 2^{210} bit flips which is roughly equivalent to 2^{203} evaluations of the system. This shows that our variant of simulated annealing seems to be a promising tool for solving non-linear Boolean equation systems with certain properties.

2 Hill climbing algorithms

Hill climbing algorithms are a general class of heuristic optimization algorithms that deal with the following optimization problem. We have a finite set X of possible configurations. Each configuration is assigned a non-negative, real number called cost, or, in other words, we have a cost function defined as $f : X \to \mathbb{R}$. For each configuration $x \in X$ a set of neighbours $\eta(x) \subset X$ is defined. The aim of the search is to find $x_{min} \in X$ minimizing the cost function $f(x)$, $f(x_{min}) = \min\{f(x) : x \in X\}$, by moving from neighbour to neighbour depending on the cost difference between the neighbouring configurations.

Johnson and Jacobsen [15] presented a unified view of many hill climbing algorithms by describing conditions on accepting a move from one configuration to another. The transition probability $p_k(x, y)$ of accepting a move from x to $y \in \eta(x)$ is defined as the product of a configuration generation probability $g_i(x, y)$ and a configuration acceptance probability $\Pr[R_k(x, y) \geq f(y) - f(x)]$, where $R_k(x, y)$ is a random variable and k is an iteration index that is increased by one after a fixed number of moves. Algorithm 1 presents a general form of a hill climbing algorithm.

Note that when $R_k(x, y) = 0$, we obtain a local search algorithm as only moves that decrease the cost are accepted.

Algorithm 1 General formulation of hill climbing algorithms

```
x_best ← x
while stopping criterion not met do
    k ← 0                                              ▷ set the outer loop counter
    while k < K do
        for m = 0, ..., M − 1 do
            generate a neighbour y ∈ η(x) with probability g_k(x, y)
            compute the cost function f(y) of the candidate
            if R_k(x, y) ≥ f(y) − f(x) then
                x ← y                                  ▷ accept the move
                if f(x) < f(x_best) then
                    x_best ← x                         ▷ store the best configuration
                end if
            end if
        end for
        k ← k + 1
    end while
end while
```

Simulated annealing Classical simulated annealing algorithm [18] is a special case of the general hill climbing algorithm presented above with a particular definition of the transition probability. The inspiration and the name comes from the process used in metallurgy to improve the durability of steel and alloys. When a metal is heated above its recrystallization temperature, the atoms break from their initial positions in the crystals and are able to relocate to other places. When slowly cooling down, the atoms are most likely to stay in new positions guaranteeing a lower total energy of the system, improving its regular structure and thus also the mechanical properties.

The simulated annealing algorithm uses a key parameter called the temperature t. The configuration generation probability is taken to be uniform, i.e. each neighbour is equally likely to be picked from each state. The acceptance probability depends on the difference $f(y)-f(x)$ in cost function between the current state x and the selected neighbour y and the current temperature t_k. The move is always accepted when it decreases the cost and with probability $e^{-(f(y)-f(x))/t_k}$ when the cost increases. In terms of the general formulation presented above, we get this behaviour when we define $R_k(x, y) = -t_k \ln(U)$, where U is a uniform random variable on $[0, 1]$.

Note that when the temperature t_k is high, many cost-increasing moves are accepted. When the temperature is lower, worsening moves are less and less likely to be accepted.

The way the "temperature" t_k of the system decreases over time (k) is called the cooling schedule. The condition necessary for the global convergence of the method is that $t_k \geq 0$ and $\lim_{k\to\infty} t_k = 0$. In practice, two most commonly used cooling schedules are the exponential cooling schedule $t_k = \alpha \cdot \beta^k$ for some parameter $0 < \beta < 1$ and the logarithmic cooling schedule $t_k = \alpha / \log_2(k+1)$ proposed in [13], where α is a constant corresponding to the starting temperature.

3 Trivium system as an optimization problem

Trivium [10] is an extremely simple and elegant stream cipher that was submitted to the ECRYPT eStream project. It successfully withstood significant cryptanalytical attention [21–23, 3] and became part of the portfolio of the eStream finalists.

To our knowledge, there is no attack on Trivium faster than the exhaustive key search so far. However, several attacks have been proposed which are faster than the naive guess-and-determine attack with complexity 2^{195} which was considered by the designers [10]. A more intelligent guess-and-determine attack with complexity 2^{135} using a reformulation of Trivium has been sketched in [17]. Furthermore, Maximov and Biryukov [21] described an attack with complexity $2^{85.5}$ and Raddum proposed a new algorithm for solving non-linear Boolean equations and applied it to Trivium in [22]. The attack complexity was 2^{164}. There have been further attacks on the small scale variant called Bivium as well as fault attacks on Trivium but we do not go into the details here.

Trivium has an 80-bit key, an 80-bit IV and 288 bits of the internal state $(s_1, \ldots, s_{288})$. At each clock cycle it updates only three bits of the state and produces one bit of the key stream using the following procedure.

for $i = 1, 2, \ldots$ **do**

$z_i \leftarrow s_{66} + s_{93} + s_{162} + s_{177} + s_{243} + s_{288}$ ▷ Generate output bit z_i

$t_{i,1} \leftarrow s_{66} + s_{93} + s_{91} \cdot s_{92} + s_{171}$

$t_{i,2} \leftarrow s_{162} + s_{177} + s_{175} \cdot s_{176} + s_{264}$

$t_{i,3} \leftarrow s_{243} + s_{288} + s_{286} \cdot s_{287} + s_{69}$

$(s_1, s_2, \ldots, s_{93}) \leftarrow (t_{i,3}, s_1, \ldots, s_{92})$

$(s_{94}, s_{95}, \ldots, s_{177}) \leftarrow (t_{i,1}, s_{94}, \ldots, s_{176})$

$(s_{178}, s_{179}, \ldots, s_{288}) \leftarrow (t_{i,2}, s_{178}, \ldots, s_{287})$

end for

During the key setup phase, the key is loaded into the first 80 bits of the state, followed by 13 zero bits, then the IV is loaded into the next 80 bits of the state and the remaining bits are filled with constant values. Then $4 \cdot 288$ clockings are computed without producing any keystream bits. Our results do not depend on this procedure.

The initial state which is the state of the registers at the time when the key generation starts can be expressed as system of sparse linear and quadratic Boolean equations [22]. We consider the initial state bits as variables and label them with $s_1 \ldots, s_{288}$. In each clocking of the Trivium algorithm three state bits are updated. The update function is a quadratic Boolean function of the state bits. In order to keep the degree low and the equations sparse we introduce new variables for each updated state bit $t_{i,1}$, $t_{i,2}$, $t_{i,3}$. We get the following equations from the first clocking

$$\begin{aligned} s_{66} \oplus s_{93} \oplus s_{91} \cdot s_{92} \oplus s_{171} &= s_{289} \\ s_{162} \oplus s_{177} \oplus s_{175} \cdot s_{176} \oplus s_{264} &= s_{290} \\ s_{243} \oplus s_{288} \oplus s_{286} \cdot s_{287} \oplus s_{69} &= s_{291} \\ s_{66} \oplus s_{93} \oplus s_{162} \oplus s_{177} \oplus s_{243} \oplus s_{288} &= z \end{aligned} \tag{1}$$

where the last equation is the key stream equation with z being the known key stream bit.

After observing 288 key stream bits we can set up a fully determined system of 954 Boolean equations in 954 unknowns [22]. We only need to consider 954 equations and unknowns instead of 1152 since we do not care about the last 66 state updates for each register. These variables will not be used in the key stream equation because the new bits are not used for the key stream generation before 66 further clockings of the cipher. By clocking the algorithm more than 288 times we can easily obtain an overdetermined system. We know that the initial state together with the corresponding updated state bits fulfills all the generated equations (1). On the other hand, for a random point each equation is satisfied with probability $\frac{1}{2}$. If we consider the

problem of solving the Trivium equation system as an optimization problem which is suitable for hill climbing algorithms (cf. Section 2) $X = \{0,1\}^{954}$ is the set of possible configurations. As a cost function $f : X \to \mathbb{R}$ we count the number of not satisfied equations in the system. We know that the minimum of the cost function is 0 and that the initial state of the Trivium system is a configuration for which the cost function is minimal. Of course there might be other optimal solutions. However, it is easy to check if the solution we found is the desired one. That a configuration is an optimal solution for the discrete optimization problem means that it generates the same first 288 bits of keystream than the initial state we are looking for. But it is unlikely that the keystream will be the same for the following keystream bits. Therefore we can check if a solution is the desired one by observing a few more keystream bit and comparing them to the keystream generated by the solution. In our experiments it is unlikely that multiple solutionss occur because we set some of the variables to there correct values and consider therefore a highly overdetermined equation system.

4 Properties of Trivium landscapes

Hill climbing algorithms are sensitive to the way in which the cost function changes when moving between configurations. Best results are obtained when a move from a configuration $x \in X$ to one of the neighbours $\eta(x)$ does not change the value of the cost function too much.

In our case we move from one configuration to another by flipping the value of a single variable. But each variable appears in at most 8 equations and in 6 equations on average, so when moving to a neighbour of the current configuration the cost function will change by at most 8. Furthermore, changing the value of a single variable will change the value of the equation with probability 1 if the variable appears in a linear term and with probability $\frac{1}{2}$ if the variable appears in a quadratic term. In the latter case flipping the value of a variable will just change the outcome of the equation if the other variable in the quadratic term is assigned to '1'. If a variable appears in the maximum of eight equations it appears in two equations in the quadratic term only. (Here it is important to note that each variable appears only once in an equation.) The expected number of equations which change their outcome is 7. Additionally it is unlikely that flipping the value of a variable changes the outcome of all equations which contain this variable in the same direction or respectively it is unlikely that all equations which contain the variable have the same outcome for the configuration before the flip. (Of course the case that a lot or even all equations have the same outcome will appear with higher probability the closer we are to the minimum.)

From these observations we infer that even if we move from a configuration x to one of its neighbours by flipping the value of a variable which appears in 8 equations we do not expect that the value of the cost function changes by 8 in almost all of the cases.

We confirmed this by the following experiment. We generated a Trivium system for a random key and calculated the cost function for a random starting point. Then we chose a neighbour configuration of our starting point and recorded the absolute value of the change in the cost function. To simulate being close to the minimum we set a number of bits to the correct solution but we allowed those bits to be flipped to move to a neighbouring configuration. The results are summarized in Table 1.

These properties of Trivium cost landscapes can be captured more formally using the notion of NK-landscapes and landscape auto-correlation as follows.

Table 1. Change of the cost function when moving to a neighbour configuration: The first row denotes the number of preassigned bits we use to simulate different distances from the minimum. We count how often out of 10000 trials the cost function changes by 0 to 8 units. The last row gives us the average change of the cost function.

i	0	100	200	300	400	500	600	700	800	900	954
0	1714	1702	1685	1560	1309	1052	944	767	601	264	0
1	3253	3246	3297	3158	2641	2143	1856	1550	1120	389	34
2	2248	2235	2240	2241	1930	1720	1385	1172	937	1001	1062
3	1557	1571	1550	1659	1821	1757	1488	1278	1258	1515	1537
4	675	665	668	754	1024	1020	911	810	741	596	648
5	386	400	380	409	691	940	1088	1078	1024	1068	1160
6	127	128	130	164	409	916	1372	1630	1866	2002	2049
7	32	44	41	46	165	439	837	1352	1854	2297	2534
8	8	9	9	9	10	13	119	363	599	868	976
average change	1.81	1.824	1.814	1.9	2.32	2.83	3.32	3.85	4.38	4.97	5.3

4.1 Trivium systems and NK-landscapes

NK-landscapes were introduced by Kaufmann [16] to model fitness landscapes with tunable "ruggedness". An NK-landscape is a set of configurations $X = \{0,1\}^n$ together with the cost function defined as

$$f(x) = \sum_{i=1}^{n} f_i(x_i; x_{\pi_{i,1}}, \ldots, x_{\pi_{i,k}}) \ ,$$

where each π_i is a tuple of k distinct elements from the set $\{1, \ldots, n\} \setminus \{i\}$. In other words, the cost function of an NK-landscape is a sum of n local cost functions f_i, each one of them depending on the main variable x_i and a set of k other variables. In a random neighbourhood model, the k indices are selected randomly and uniformly for each f_i. Depending on the value of k, we get either smooth landscapes with relatively few local minima when k is small and rugged landscapes for large values of k.

The Trivium optimization problem can be seen as such combinatorial landscape. Consider the basic system of equations. We define each f_i as the contribution of i-th equation (either 0 or 1 depending on whether it is satisfied). Each equation depends on six distinct variables, we verified by a computer program that indeed we can always pick one of them as the main variable leaving exactly five other ones for each equation. Trivium optimization problem can thus be seen as an instance of NK-landscape with $n = 954$ and $k = 5$, a rather small value hinting at a certain smoothness of this landscape.

4.2 Landscape auto-correlation

Another measure of landscape ruggedness is the notion of landscape correlation introduced by Weinberger [24]. We will follow the exposition by Hordijk [14]. The main idea is to perform a random walk on the landscape via neighbouring points. At each step, the cost function y_t is recorded. That way a sequence $(y_t)_{t=1\ldots T}$ is obtained and we compute its auto-correlation coefficients.

The auto-correlation of a sequence (y_t) for the time lag i is defined as

$$\rho_i = Corr(y_t, y_{t+i}) = \frac{E[y_t \cdot y_{t+i}] - E[y_t]E[y_{t+i}]}{Var[y_t]}$$

where E means the expected value and Var variance of a random variable. Estimates r_i of these auto-correlations ρ_i are

$$r_i = \frac{\sum_{t=1}^{T-i}(y_t - \bar{y})(y_{t+i} - \bar{y})}{\sum_{t=1}^{T}(y_t - \bar{y})^2}$$

where $\bar{y}$ means the mean value of y_t. Here a large auto-correlation coefficient corresponds to a smooth landscape. An important assumption that has to be made for such analysis to be meaningful is that the landscape is statistically isotropic. This means that the statistics of the time series generated by a random walk are the same, regardless of the starting point. Only then a random walk is "representative" of the entire landscape. By computing correlation coefficients for many random walks starting at different points we experimentally verified that the Trivium landscape can be seen as isotropic.

Selected correlation coefficients computed for a basic version of Trivium system and overdefined versions are presented in Table 2. Clearly, generating the overdefined system makes the landscape smoother.

Table 2. Correlation coefficients for landscapes generated by Trivium systems of different sizes. n denotes the number of variables in the system.

keystream length	n	$r(1)$ $r(10)$ $r(20)$ $r(30)$ $r(40)$ $r(50)$
288	954	0.989 0.896 0.803 0.720 0.646 0.580
576	1818	0.994
1152	3546	0.997

5 Solving Trivium system with modified simulated annealing

The properties of landscapes generated by the Trivium system of equations suggest that it might be possible to employ stochastic search methods such as simulated annealing to try to find a global optimum and thus recover the secret state of the cipher. In this section we report the results of our experiments in this direction.

Initial experiments with standard simulated annealing were not very encouraging. To be able to solve the Trivium system in reasonable time, we needed to simplify the initial system by setting around 600 out of 954 variables to their correct values throughout the search.

We experimented with the algorithm and its various modifications and found one that yielded a significant improvements over the standard algorithm. The algorithm works as follows. As with standard simulated annealing, we randomly generate a neighbour. If the cost decreases, we accept this move. If not, instead of accepting with probability related to the current temperature, we pick another neighbour and test that one. If after testing a certain number of neighbours we cannot find any cost decreasing move, we accept the increasing move with some probability, just as in the plain simulated annealing. The parameter of this procedure is the number of additional candidates to test before accepting cost increase.

If the parameter is zero, we get plain simulated annealing. On the other end of the spectrum, if we test all possible neighbours, it is easy to see that we get an algorithm that is equivalent to local search, we look for any possible decreasing move and we follow it. When we are in a local minimum, we enter a loop, we finally accept one

of the cost increasing candidates but in the next move we always go back to the local optimum we found. Setting the parameter between those extremes yields an intermediate algorithm.

In practice, we used a probabilistic variant of this approach that randomly selects neighbours until it finds one with smaller cost or it exceeds the number of tests defined as a parameter nochangebound. This algorithm is presented in Alg. 2.

Algorithm 2 Modified version of simulated annealing

```
x_best ← x
T ← α                                          ▷ initial temperature parameter is α
k ← 0                                          ▷ set the outer loop counter
while T > 1 do
    for m = 0, ..., M − 1 do                   ▷ parameter M is the number of inner runs
        generate a neighbour y ∈ η(x) uniformly
        if f(y) < f(x) then                    ▷ if cost decreased
            x ← y                              ▷ accept the move
            if f(x) < f(x_best) then           ▷ found a new best value
                x_best ← x                     ▷ store the best configuration
                nc ← 0                         ▷ reset the neighbor counter
                if f(x_best) = 0 then          ▷ if we found a solution
                    return x_best              ▷ finish and return it
                end if
            end if
        else                                   ▷ the candidate cost is higher
            nc ← nc + 1
            if (nc > nochangebound) ∧ (exp((f(y) − f(x))/T > rnd[0, 1]) then
                x ← y                          ▷ accept the move
                nc ← 0                         ▷ reset the counter of tested neighbours
            end if
        end if
    end for
    k ← k + 1
    T = α/ log_2(k · M)                        ▷ Logarithmic cooling schedule
end while
```

The relationship between the number of neighbours tested and the time it took to find a solution (measured in the number of neighbours tested) is presented in Fig 1. Values of nochangebound below 25 result in running times exceeding 2^{40} flips. It suggests that the proper choice of nochangebound is critical for the efficiency of the simulated annealing, in particular, it cannot be too small.

6 Experimental results

In this section we report results of our computational experiments with the basic equation system generated by the problem of recovering internal state of Trivium. We took the fully determined system with 954 equations and variables obtained after observing 288 bits of the keystream.

We made some comparisons between exponential and logarithmic cooling schedules and from our limited experience the logarithmic cooling schedule performed better in more cases, so we decided to pick that one for our further tests.

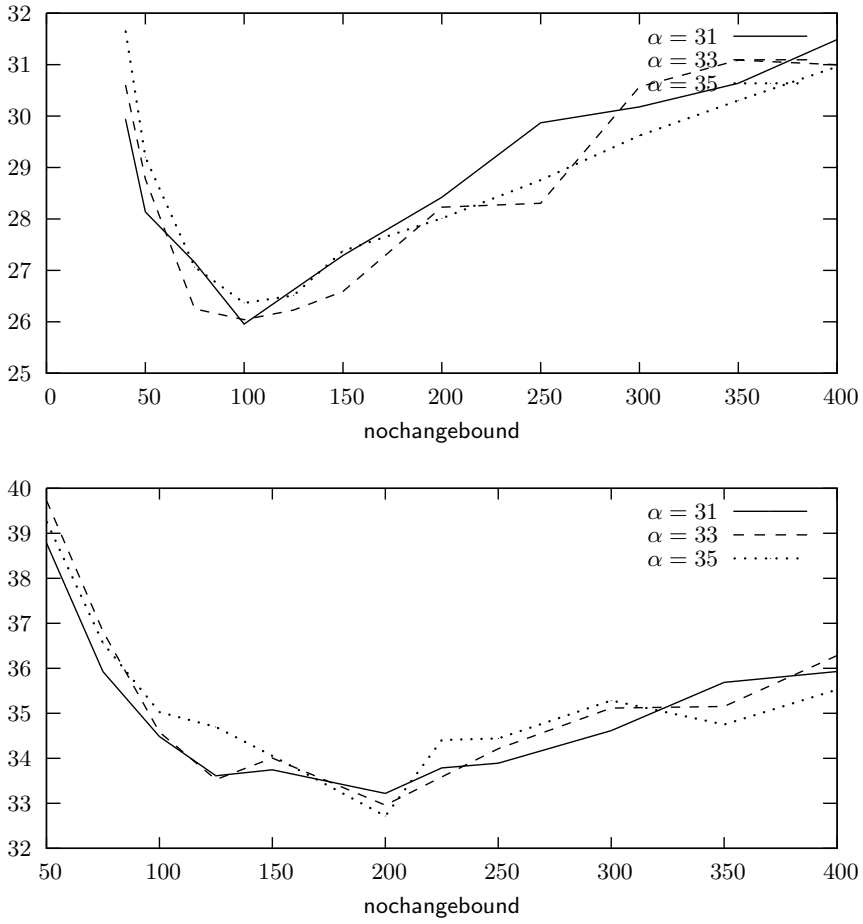


Fig. 1. Influence of nochangebound parameter on the efficiency of simulated annealing applied to basic Trivium system for three values of initial temperature α. Other parameters are $M = 1024$ (cf. Alg. 2), averages are over 10 tests. In the top figure we guessed 200 first bits of the state, in the bottom one 180 bits.

The values of α were picked based on empirical observations. Too large α resulted in prolonged periods of almost-random walks where there was no clear sign that any optimization might occur. Too small values gave the behavior similar to a simple local search when the process was getting stuck in some shallow local optima. After a few trials we decided to use the initial temperature parameter $\alpha = 35$.

For each number of bits of the state fixed to their correct values (preassigned) we ran ten identical tests with different random seeds testing various values of nochangebound parameter (from the set 100, 150, 175, 200, 250, 300). After the test batch finished, we picked that value of nochangebound that yielded lowest search time. We managed to obtain optimal values for nochangebound for 200, 195, 190, 185, 180, 175 and 170 preassigned bits where we set the values of the first bits of the internal state. We use this optimal nochangebound to estimate the total complexity of the attack. The graph is presented in Fig. 2. The total complexity is the product of the num-

ber of guesses we would need to make ($2^{preassigned}$) multiplied by the experimentally obtained running time of the search for the solution.

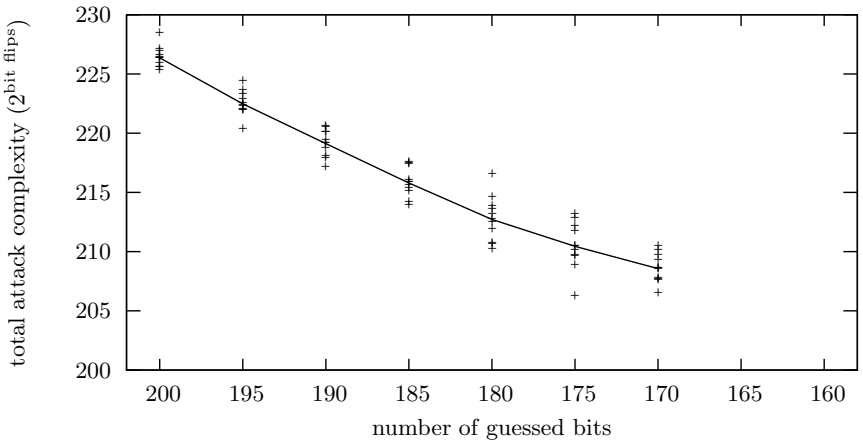


Fig. 2. Running times of the attack based on modified simulated annealing depending on the number of guessed bits. The numbers on the vertical axis are base two logarithms of the total number of moves necessary to find the solution. Crosses represent results of single experiments, the line connects averages.

The results show that the running time of the attack decreases with the smaller number of guessed bits since the increase in time of the search procedure is smaller than the decrease due to the smaller number of bits we have to guess. If the curve goes down below the complexity level corresponding to 2^{80} key setups of Trivium, it would constitute a state-recovery attack. However, our problem is that due to limited computational power we were not able to gather enough results for values of *preassigned* smaller than 170. Our program running on 1.1GHz AMD Opteron 2354 was able to compute 2^{35} bit flips per hour and tests with $preassigned = 170$ required around $2^{38} \sim 2^{39}$ bit flips.

It seems that trying to extrapolate the running times is rather risky, since we do not have any analytical explanation of the complexities we get as often is the case with heuristic search methods. Therefore we do not claim anything about the feasibility of such an attack on full Trivium. We can only conjecture that there might be a set of parameters for which such attack may become possible.

Due to the computational complexity, our experimental results are so far based on only rather small samples of runs for the fixed set of parameters. Therefore, they cannot be taken as a rigorous statisticial analysis but rather as a reconnaissance of the feasibility of this approach. However, we have noticed that for overwhelming fraction of all the experiments, the running times for different runs with the same set of parameters do not deviate from the average exponent of the bit flips by more than ± 2, i.e. most of the experiments have the number of flips between 2^{avg-2} and 2^{avg+2}. Therefore, we believe that the results give some reasonable impression of the actual situation.

7 Some variations

The previous section presented the set of our basic experiments. However, there is a multitude of possible variations of the basic setup which could possibly lead to better results. In this section we mention some variations of the search problem we considered while looking for possible optimizations.

7.1 Guessing strategy

In order to lower the complexity of solving the equation system we set some of the variables to their correct values. However, the search complexity depends on which variables we choose.

We used different guessing strategies for pre-assigning variables and compared the influence on the running time of our algorithm. We used the following strategies to guess subsets of the state bits:

1. Select the first variables of the initial state.
2. Select the first variables of each register of the initial state.
3. Select the last variables of the initial state.
4. Select the last variables of the each register of the initial state.
5. Select the most frequent variables. These are the variables which are introduced by the update function at the beginning of the key stream generation. We guess values for variable s_{289} and the consecutive ones in this case.
6. An adaptive pre-assignment strategy which is similar to the ThreeFour strategy in [11] (see Subsection 7.1).
7. Select the variables in such a way that the equation interdependence measure is minimal. (see Subsection 7.1).

It turns out that the best guessing strategy of the ones we tested is to guess the first bits of the initial state. In addition to a pre-assignment of variables we can determine the value of further variables by considering the linear and quadratic equations (see below). We use this technique in the adaptive pre-assignment strategy.

Table 3. Running time for different pre-assignment strategies. nochangebound=110, 190 bits are preassigned, average taken over ten runs.

	first bits of the initial state	most frequent bits	first bit of every register	last bit of the initial state	last bit of each register
average	**29.5**	33.0	34.5	31.2	36.4

Adaptive pre-assignment strategy In this pre-assignment strategy we use the fact that assigning 5 of the variables in a linear equation will uniquely determine the 6th variable. Starting with an arbitrary linear equation we guess and pre-assign 5 of the 6 variables, determine the value of the remaining variable and assign this to its value. We know that a large fraction of the variables appear in two linear equations. So in the next round of pre-assignment we pick an equation in which at least one variable is already assigned. That means we only have to guess at most 4 variable to get one for free. We continue until we have made the maximum number of guesses or we cannot find an equation in which one variable is already assigned. In the latter case

we just have to pick an equation without preassigned variables and run the algorithm again until we made the maximum number of guesses.

Additionally we also use the quadratic equations to determine the value of variables.

The advantage of this pre-assignment strategy is that we can assign many more variables than we actually have to guess. Table 4 gives us an impression of this advantage.

Table 4. The table shows how many bits additional to the guessed bits can be assigned using the adaptive pre-assignment strategy.

# guessed bits	# assigned bits	additional assigned bits in %
5	6	20%
50	66	32%
100	135	35%
200	281	40.5%

The disadvantage is that we instead of making the equations sparser we fix some equations to be zero. That means that there are less equations left which contain free variables but the maximum number of equations in which a variables appears is still 8. Therefore a variable influences a higher percentage of equations.

Minimizing equation interdependency If all the equations used different sets of variables, it would be trivial to solve the system by a simple local search. However, variables appear in many equations and changing the value of one of them influences other equations at the same time. This suggests the idea of guessing (pre-assigning) bits to minimize the number of variables shared by many equations and thus reduce the degree of mutual relationships between equations.

Capturing this intuition more formally, let E_i be an equation and let $\mathcal{V}(E_i)$ denote the set of *not preassigned* variables that appear in the equation. We can define the measure of interdependence of two equations E_i, E_j as

$$IntrDep(E_i, E_j) = |\mathcal{V}(E_i) \cap \mathcal{V}(E_j)| \ .$$

If the measure is zero, equations use different variables and we can call them separated. Note that pre-assigning any bit that is used by both equations decreases the value of interdependence.

To capture the notion of equations interdependence in the whole system of Trivium equations E, the following measure could be used

$$\sum_{e,g \in E, e \neq g} |\mathcal{V}(e) \cap \mathcal{V}(g)|^2 \ . \tag{2}$$

We used the sum of squares to prefer systems with more equations with only few active (non-preassigned) variables over less equations that have more active variables, but it is possible to use an alternative measure without the squares,

$$\sum_{e,g \in E, e \neq g} |\mathcal{V}(e) \cap \mathcal{V}(g)| \ . \tag{3}$$

The algorithm for pre-assigning bits to minimize the above measure is rather simple. We start with computing the initial interdependence of the system. Then, we

temporarily pick a free variable and assign it to compute the new interdependence of the system. If this value is smaller than the current record, we remember it as a new record. After we test all possible candidates, we pick the record one and assign it for good. We repeat this procedure until we get the required number of preassigned bits.

We performed an experiment that compared the results of the reference pre-assignment strategy fixing the first 190 bits of the state with two variants minimizing (2) and (3). Results presented in Table 5 are interesting. It seems that in spite of significant smoothening of the landscape indicated by higher values of the coefficient ξ the first strategy minimizing (2) significantly worsens the running time. A possible explanation may be that the landscape became more like "golf-course" with large areas without any direction and only very small attraction basins leading to global solution(s). Another possibility is that for such systems, different parameter of nochangebound is preferred. The second variant minimizing (3) seems to be only slightly better than setting the first bits, but more tests would be needed for more parameters to decide any definite advantage.

Table 5. Running times and landscape auto-correlation coefficients ξ for bit pre-assignment strategies minimizing equation interdependence. Experiments used $\alpha = 33$, $M = 1024$, nochangebound $= 110$.

strategy:	reference	Case 1	Case 2
avg:	29.34	38.9	28.72
ξ	90.1	97.4	96.1

7.2 Using overdefined systems

Results on landscape auto-correlation suggest that using overdefined systems may yield landscape with better structural properties. However, this happens at the expense of a larger set of variables and equations we have to deal with. Our experimental results on overdefined systems suggest that the gain we get from a better landscape is offset by the larger system size so search times are actually not better.

7.3 Variable persistence

According to [4, 5] while using simulated annealing to some optimization problems, one can observe a bias in the frequency of assigning values to variables during the simulated annealing procedure. This bias is related to the solution of the system and observing it can give some information on the solution we are looking for.

We made some experiments that investigated if configurations of local minima (states we run into after a long cooling run) have variables correlated with the global minimum state. In our limited experiments with the basic Trivium system we did not observe any such correlations.

8 Conclusions and future directions

We presented a new way of approaching the problem of solving systems of sparse, multivariate equations over the binary field. We represent them as combinatorial optimization problems with the cost function being the number of not satisfied equations

and then we apply some heuristic search methods, such as simulated annealing to solve them.

We showed that such systems may be relevant in cryptography by giving an example of the system generated by the problem of recovering the internal state of the stream cipher Trivium.

Our experimental results were focused on Trivium system and they seem to be promising but for now they do not seem to pose any real threat to the security of this algorithm.

We hope that this paper will serve as a starting point for further research in this direction. There are many open problems in this area, the most obvious ones are the selection of better parameters of the search procedures and analytically estimating the possible complexity of such algorithms.

The other interesting direction seems to be the investigation of alternative cost functions. In all our experiments we use the simplest measure counting the number of not satisfied equations. However, many results in heuristic search literature suggest that the selection of a suitable cost function may dramatically change the efficiency of a search. The question of determining whether in our case there exist measures better than the one we used is still open.

References

1. F. Armknecht and M. Krause. Algebraic attacks on combiners with memory. In *Advances in Cryptology – CRYPTO 2003*, volume 2729 of *LNCS*, pages 162–175. Springer, 2003.
2. E. Biham and A. Shamir. *Differential Cryptanalysis of the Data Encryption Standard*. Springer-Verlag, 1993.
3. J. Borghoff, L. R. Knudsen, and M. Stolpe. Bivium as a mixed-integer linear programming problem. In *Cryptography and Coding, 12th IMA International Conference*, volume 5921 of *LNCS*, pages 133–152. Springer, 2009.
4. P. Chardaire, J. L. Lutton, and A. Sutter. Thermostatistical persistency: A powerful improving concept for simulated annealing algorithms. *European Journal of Operational Research*, 86(3):565–579, Nov. 1995.
5. J. A. Clark and J. L. Jacob. Fault injection and a timing channel on an analysis technique. In *Advances in Cryptology – EUROCRYPT 2002*, volume 2332, pages 181–196. Springer, 2002.
6. D. Coppersmith. The data encryption standard (DES) and its strength against attacks. *IBM Journal of Research and Development*, 38(3):243–250, 1994.
7. N. Courtois, E. Klimov, J. Patarin, and A. Shamir. Efficient algorithms for solving overdefined systems of multivariate polynomial equations. In B. Preenel, editor, *Advances in Cryptology – EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 392–407. Springer-Verlag, 2000.
8. N. Courtois and W. Meier. Algebraic attacks on stream ciphers with linear feedback. In *Advances in Cryptology – EUROCRYPT 2003*, volume 2656 of *LNCS*, pages 644–644. Springer, 2003.
9. N. T. Courtois and J. Pieprzyk. Cryptanalysis of block ciphers with overdefined systems of equations. In *Advances in Cryptology – ASIACRYPT 2002*, volume 2501 of *Lecture Notes of Computer Science*, pages 267–287. Springer-Verlag, 2002.
10. C. De Cannière and B. Preneel. Trivium – a stream cipher construction inspired by block cipher design principles. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/030 (2005-04-29), 2005. http://www.ecrypt.eu.org/stream/papers.html.
11. T. Eibach, E. Pilz, and S. Steck. Comparing and optismising two generic attacks on Bivium. In *SASC*, 2008.
12. A. S. Fraenkel and Y. Yesha. Complexity of solving algebraic equations. *Inf. Process. Lett.*, 10(4/5):178–179, 1980.

13. S. Geman and D. Geman. Stochastic relaxation, Gibbs distributions, and the bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 6(6):721–741, 1984.
14. W. Hordijk. A measure of landscapes. *Evolutionary Computation*, 4(4):335–360, 1996.
15. A. W. Johnson and S. H. Jacobson. A class of convergent generalized hill climbing algorithms. *Applied Mathematics and Computation*, 125(2-3):359–373, 2002.
16. S. A. Kauffmann. *The Origins of Order: Self-Organization and Selection in Evolution*. Oxford University Press, 1993.
17. S. Khazaei. Re: A reformuation of TRIVIUM. Posted on the eSTREAM Forum, 2006.
18. S. Kirkpatrick, J. Gelatt, C. D., and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, 1983.
19. L. R. Knudsen and W. Meier. Cryptanalysis of an identification scheme based on the permuted perceptron problem. In *Advances in Cryptology – EUROCRYPT'99*, volume 1592, pages 363–374. Springer, 1999.
20. M. Matsui. Linear cryptoanalysis method for DES cipher. In *Advances in Cryptology – EUROCRYPT'93*, volume 765 of *LNCS*, pages 386–397. Springer, 1993.
21. A. Maximov and A. Biryukov. Two trivial attacks on Trivium. In *Selected Areas in Cryptography – SAC 2007*, volume 4876 of *LNCS*, pages 36–55. Springer, 2007.
22. H. Raddum. Cryptanalytic results on Trivium. eStream, 03 2006. available from http://www.ecrypt.eu.org/stream/papersdir/2006/039.ps.
23. M. S. Turan and O. Kara. Linear approximations for 2-round trivium. In *Security of Information and Networks – SIN 2007*, pages 96–105. Trafford Publishing, 2007.
24. E. Weinberger. Correlated and uncorrelated fitness landscapes and how to tell the difference. *Biological Cybernetics*, 63(5):325–336, Sept. 1990.