Universität Bremen
Department of Computer Science

Diplomarbeit

Algebraic Attacks on the Courtois Toy Cipher

by

Martin Albrecht

1478887
Dobbenweg 9
28203 Bremen

Thesis Supervisor:
Prof. Dr. Michael Hortmann
Reader:
Prof. Dr. Hans-Jörg Kreowski

Bremen, December 18, 2006

Abstract

Block ciphers are fundamental building block of modern cryptography. Recently, a new technique to attack block ciphers has emerged called "algebraic attacks". These attacks work by expressing block ciphers as quadratic equation systems and solving those systems of equations. In May 2006 Nicolas Courtois – author of many influential research papers on algebraic attacks – presented a toy cipher called CTC and claimed to have broken this cipher in a configuration where the block size is 255-bit and the number of rounds is six.

This thesis presents, discusses, and implements some of the most important algebraic attack algorithms (F_4, DR, XL) and employs them against Courtois' toy cipher. Also CTC is attacked using more specialized algorithms and the experimental results of these attacks are presented.

Contents

Chapter 1

Introduction

In 2001 the Rijndael block cipher [DR02] was chosen by the U.S. National Institute of Standards and Technology (NIST) as the Advanced Encryption Standard (AES) [FIP01]. It was specifically designed to withstand well known attack techniques against blockciphers. Most notably it was designed to resist linear and differential cryptanalysis [DR99]. The specification of Rijndael is – in contrast to many other block ciphers like DES – very simple and algebraically clean: The S-box – the only non-linear part of Rijndael – is a patched inversion in $\mathbb{F}_{2^8}$ where 0 is mapped onto itself. In the following years the AES was consequently reformulated as a multivariate polynomial equation system ($\mathcal{MQ}$) over $\mathbb{F}_2$ [CP02a] and $\mathbb{F}_{2^8}$ [MR02]. If any of those systems was solved faster than exhaustive key search the AES was broken.

This idea is not new. Shannon [Sha49] proclaimed in 1949: "Thus, if we could show that solving a certain system requires at least as much work as solving a system of simultaneous equations in a large number of unknowns, of a complex type, then we would have a lower bound of sorts for the work characteristic". As it is well known that solving random systems of multivariate equations is NP-hard this requirement is reasonable.Therefore, it is not suprising that cryptographic systems can be represented as systems of multivariate equations somehow and there is no reason to assume that solving these systems would be faster than exhaustive key search.

But not all instances of NP-hard problems must be NP-hard themselves. It might be possible to express a cryptosystem in such a way that it is easier to solve than in exponential time: Solving such a system is called an "algebraic attack" in literature (e.g. see [Cou06] and [BPW05]). These attacks are motivated by the fact that the equation systems derived from the AES are both sparse and overdefined.

Please note, the aim of such attacks is mostly to recover the encryption key and therefor this thesis will only deal with *key recovery attacks*.

Several algorithms have be proposed to solve this kind of equation systems: XL [CKPS00], XSL [CP02a], DR [TF05], Zhuang-Zi [DGS06], F_4 [Fau99], and F_5 [Fau02]. The first purpose of this thesis is to present some of the most recognized of these algorithms: F_4 (Section 4.1) and XL (Section 4.3). Also a less recognized algorithm called DR (Section 4.2) will be presented which uses resultants to solve the underlying problem. These presentations will be kept brief wherever possible and for several proofs of involved theorems the reader will be directed to the appropriate literature. However as most of these algorithms compute a Gröbner basis to solve the $\mathcal{MQ}$ problem the presentation is prepended with a brief introduction to the theory of ideals, varieties, and Gröbner bases (Chapter 2).

Up until recently no one claimed to have broken anything but a toy cipher using algebraic attacks on block ciphers. This changed when in May 2006 Nicolas Courtois published (see [Cou06]) the specifications of cipher – Courtois Toy Cipher (CTC) – along with a way to express this cipher

as a multivariate equation system over $\mathbb{F}_2$. He claims and demonstrated to have broken this cipher by solving the associated equation system (called CTC ideal basis in this thesis) faster than exhaustive key search. In particular he claims to have broken a 255-bit blocksize and six round instance of CTC in under one hour on his notebook computer.

Nicolas Courtois calls his attack "fast algebraic attack against block ciphers". However he didn't publish the details of his attack as he was afraid his attack could be extended to break AES quite quickly: "In order to protect the United States government, the financial institutions, mobile phone operators, and hundreds of millions of other people that use AES, from criminals and terrorists, the exact description of the attack will for some time not be published. Public demonstrations of the effectiveness of the attack will be organized instead. However one should understand that the attack is quite simple and fatally will be re-discovered (and published)" [Cou06].

Please note that so far there is no evidence that Nicolas Courtois' attack can be extended to break AES. Also this thesis will not argue in favor or against the claim that AES is broken because there is just not enough data to work with at this point: the "fast algebraic attack against block cipher" is still unpublished and no claim was made by Nicolas Courtois that he can actually break AES.

On the other hand, the assumption of this thesis is that CTC can be broken with algebraic attacks effectively as it was designed for that purpose. But as the actual attack of Nicolas Courtois is unpublished the second purpose of this thesis is to attack CTC and report the results of and observations on these experiments. I hope this work contributes to a better understanding of CTC and thus algebraic attacks on block ciphers.

Open Sources

To perform these experiments, most algorithms presented in this thesis had to be implemented first. Even though e.g. MAGMA [BCP97] provides a very fast implementation of F_4 (See chapter 4.1) and e.g. Toon Segers has implemented F_4 in MAGMA [Seg04] and provides the source code of his implementation no true general purpose and fully functional open-source implementation exists (see section 4.1 for details). "True open source" in this context means an implementation which does not rely on proprietary software like MAGMA to perform any part of the algorithm.

I believe that open sources are a crucial aspect of scientific work not only in computational mathematics. The following quotation of J. Neubüser [Neu95] is meant to emphasize this:

> You can read Sylow's Theorem and its proof in Huppert's book in the library without even buying the book and then you can use Sylow's Theorem for the rest of your life free of charge, but – and for understandable reasons of getting funds for the maintenance [...] – for many computer algebra systems license fees have to be paid regularly for the total time of their use. In order to protect what you pay for, you do not get the source, but only an executable, i. e. a black box. You can press buttons and you get answers in the same way as you get the bright pictures from your television set but you cannot control how they were made in either case.
>
> With this situation two of the most basic rules of conduct in mathematics are violated: In mathematics information is passed on free of charge and everything is laid open for checking. Not applying these rules to computer algebra systems that are made for mathematical research [...] means moving in a most undesirable direction. Most important: Can we expect somebody to believe a result of a program that he is not allowed to see? Moreover: Do we really want to charge colleagues in Moldava several years of their salary for a computer algebra system?

Consequently, all computations in this thesis can be performed using only free-of-charge open-source software which may be inspected, modified, and redistributed. This has a huge impact

speedwise – as mentioned for the case of F_4 earlier – but seems the only way to ensure that the principles of verifiability and royalty-freedom are not violated.

The computer algebra system chosen to implement these algorithms and to perform experiments with is "SAGE: Software for Algebra and Geometry Experimentation" [SJ05] which is provided under the terms of the GNU General Public License. SAGE is described as a "free and open software that supports research and teaching in algebra, geometry, number theory, cryptography, etc." (`http://sage.math.washington.edu/sage`). SAGE includes the following software in the standard distribution:

Group theory and combinatorics	GAP
Symbolic computation and Calculus	Maxima
Commutative algebra	Singular
Number theory	PARI, MWRANK, NTL, Givaro
Graphics	Matplotlib
Numerical linear algebra	Numeric, GSL
Mainstream programming language	Python
Interactive Shell	IPython

Many more open source packages may be installed optionally and used from within SAGE. Also interfaces to a wide range of commercial, closed-source computer algebra systems are provided. For this thesis the Singular [GPS05] computer algebra system has been the most used component of SAGE mainly because it is the fastest Gröbner basis environment in the open-source world.

Plotting Experimental Data

As estimating the runtime of Gröbner basis algorithms is a hard problem timing experiments are carried out in this thesis to compare algorithms, term orders, etc. The results will be presented in plots.

Every plot from timing experiments will show the average run time (bold line with bold points), the minimal runtime (lower edges of the polygone surrounding the bold line), and the maximal runtime (the upper edges of the polygone surrounding the bold line) as occurred during the experiment of the algorithm. Additionally an exponential least-square fit may be plotted for the average runtime. So e.g. a plot might look like Figure 1.1 on the following page:

All timing experiments were – unless stated otherwise – performed on William Stein's SAGE Sandbox `http://sage.math.washington.edu` with 64GB of RAM and 8 dual-core AMD 1.8 Ghz Opteron processors.

Structure of this Thesis

The structure of this thesis is as follows: Chapter 2 introduces notation and the mathematical background, Chapter 3 presents quadratic equation systems for the CTC, Chapter 4 describes algebraic attack algorithms and their utilization against CTC, Chapter 5 explains how to use the software included with this thesis, and Chapter 6 summarizes the results of this work. Please note that Appendix A contains a full source code listing.

Acknowledgments

I would like to thank Prof. Michael Hortmann for the supervision of this thesis, for the mathematical education he provided, and his encouragement. I would also like to thank Prof. William Stein

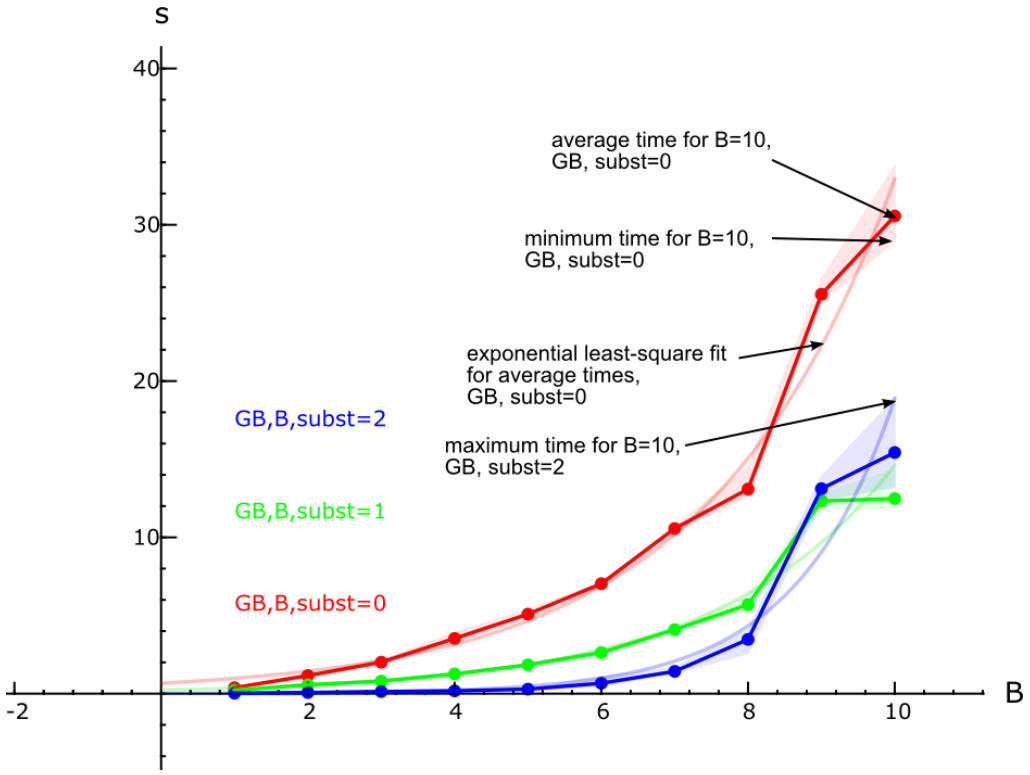

Figure 1.1: Example Plot

for the SAGE computer algebra system which was used to implement this thesis, fruitful discussions, comments on this thesis, and the permission to use his powerful `sage.math.washington.edu` machine[1] to perform my experiments. Furthermore, I would like to thank Torben Gerkensmeyer, Georg Lippold, and Ralf-Phillip Weinmann for proof-reading my thesis. Of course, any remaining errors are my responsibility. Silke Jahn did not have any direct impact on this thesis and it's content but it would be a very different thesis without her.

[1]The purchase of this machine was supported by William Stein's NSF grant No. 0555776.

Chapter 2

Mathematical Background

As many algorithms presented in this thesis compute a Gröbner basis at some point this chapter briefly states the main theorems which link Gröbner bases and solutions to multivariate polynomial systems. This roughly follows Section 2 of A.J.M. Seger's Master's thesis ([Seg04]) and so as "Ideals, Varieties, and Algorithms" ([CLO05]) by David Cox, John Little, and Donal O'Shea. As the focus of this thesis is to attack CTC this chapter will not provide a comprehensive introduction to the theory of ideals, varieties, and Gröbner bases. Instead the main theorems necessary to understand Gröbner basis attacks are stated briefly and references to appropriate literature are provided for the interested reader.

2.1 Notation

The following notation is used throughout the thesis:

- k or $\mathbb{F}$ is the base field of our polynomial ring. $\overline{k}$ represents the algebraic closure of k.
- $\mathbb{F}_p$ is the finite field of characterstic p with p prime; $\mathbb{F}_{p^n}$ the finite extension field of degree n over $\mathbb{F}_p$.
- P, R are polynomial rings: $k[x_0, \ \ldots \ , x_{n-1}]$.
- Any polynomial p is identified with the equation $p = 0$ where appropriate. It should be clear from the context which representation is referred to.
- I denotes an ideal in P (see Definition 2.3.4).
- We call $m = x_0^{\alpha_0} x_1^{\alpha_1} \ldots x_n^{\alpha_n}$ a monomial and $t = c \cdot m$ with $c \in k$ a term. $M(F)$ is the set of monomials that appear in the set F of polynomials and $T(F)$ the set of terms that appear in the same set F.
- $\alpha(m)$ is the exponent vector $(\alpha_0, \alpha_1, \ldots, \alpha_n)$ of the monomial $m = x_0^{\alpha_0} x_1^{\alpha_1} \ldots x_n^{\alpha_n}$.
- $multideg(f)$ is the largest exponent vector of a polynomial f with respect to some monomial order (defined below). So for a monomial m: $multideg(m) = \alpha(m)$.
- $deg(f) \ = \ \sum_{\alpha_i \in multideg(f)} \alpha_i$
- $\mathrm{LC}(f) \ = \ a_{multideg(f)} \ \in \ k$ is the leading coefficient of the polynomial f.
- $\mathrm{LM}(f) \ = \ x^{multideg(f)}$ is the leading monomial of the polynomial f. $LM(F)$ is defined as $\{LM(f_i) : f_i \in F\}$ where F is a finite set of polynomials.

- $\mathrm{LT}(f) = \mathrm{LC}(f) \cdot \mathrm{LM}(f)$ is the leading term of of the polynomial f. $LT(F)$ is defined as $\{LT(f_i) : f_i \in F\}$ where F is a finite set of polynomials.
- $A_{i,j}$ represents the element in row i and column j in the matrix A.
- $f \ \% \ g$ denotes the modulo operation f mod g.

Whereever possible examples are provided for a given theorem, algorithm, or proposition. The canonical example in this thesis is going to be the following set of polynomials in $\mathbb{F}_{127}[x_0, x_1]$ with term order *lex* (see Definition 2.1.1).

$$0 = 114 + 80x_0x_1 + x_0^2,$$
$$0 = 29 + x_1^2 + 107x_0x_1$$

This example can be produced using the software on the provided CD as follows:

```
sage: attach 'mq.py'
sage: R.<x0,x1> = PolynomialRing(GF(127),2,order='lex')
sage: F = MQ(R,[114 + 80*x0*x1 + x0^2, 29 + x1^2 + 107*x0*x1])
```

In the above example a monomial ordering was fixed to construct a polynomial system. The most important monomial orderings for this thesis are *lex* and *degrevlex* explained below:

Definition 2.1.1 (Lexicographic monomial ordering *lex*)**.** *Let $\alpha = (\alpha_0, \ldots, \alpha_{n-1})$ and $\beta = (\beta_0, \ldots, \beta_{n-1}) \in \mathbb{Z}^n_{\geq 0}$. We say $\alpha \underset{lex}{>} \beta$ if, in the vector difference $\alpha - \beta \in \mathbb{Z}^n$, the left most nonzero entry is positive. We will write $x^\alpha \underset{lex}{>} x^\beta$ if $\alpha \underset{lex}{>} \beta$.*

It will be shown later that *lex* is the order which allows to "read" the solution to a multivariate equation system from the Gröbner basis. But computing a lexicographical Gröbner basis usually takes significantly longer than computing a *degrevlex* Gröbner basis:

Definition 2.1.2 (Degree reverse lexicographic monomial ordering *degrevlex*)**.** *Let $\alpha = (\alpha_0, \ldots, \alpha_{n-1})$ and $\beta = (\beta_0, \ldots, \beta_{n-1}) \in \mathbb{Z}^n_{\geq 0}$. We say $\alpha \underset{degrevlex}{>} \beta$ if*

$$deg(\alpha) > deg(\beta), \text{ or } deg(\alpha) = deg(\beta)$$

and the rightmost nonzero entry in the vector difference $\alpha - \beta \in \mathbb{Z}^n$ is negative. We will write $x^\alpha \underset{degrevlex}{>} x^\beta$ if $\alpha \underset{degrevlex}{>} \beta$.

For example consider the polynomial

$$1 + y_1 + x_2 + x_1 + x_0 + x_0x_1.$$

With respect to the lexicographical monomial ordering and a variable order where $y_i > x_i$ the leading monomial is y_1 but with respect to a graded degree reverse lexicographical ordering the leading monomial is x_0x_1.

If not stated otherwise the *degrevlex* monomial ordering will be used.

2.2 Coefficient Matrices and Systems of Polynomial Equations

Many algorithms presented in this thesis construct coefficient matrices from finite lists of polynomials defined as follows:

Every finite list $F = [f_0 \dots f_{s-1}]$ of polynomials in P may be represented as the pair A_F, v_F where A_F is the *coefficient matrix* of F and v_F is the *monomial vector* of F with the following definitions: Fix a monomial ordering in P and let $v = [m_0, \dots, m_{n-1}]$ be the ordered set of all monomials occurring in F. Let A_{ij} be the coefficient of m_j in f_i and set the $s \times n$ matrix $A_F = (A_{ij})$. Then F is represented by A_F, v_F as

$$F = A_F * v_F$$

So for example,

$$0 = 114 + 80x_0x_1 + x_0^2,$$
$$0 = 29 + x_1^2 + 107x_0x_1$$

in $\mathbb{F}_{127}[x_0, x_1]$ with term order *lex* may be expressed as:

$$\begin{pmatrix} 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 114 + 80x_0x_1 + x_0^2 \\ 29 + x_1^2 + 107x_0x_1 \end{pmatrix} = \begin{pmatrix} 1 & 80 & 0 & 144 \\ 0 & 107 & 1 & 29 \end{pmatrix} \cdot \begin{pmatrix} x_0^2 \\ x_0x_1 \\ x_1^2 \\ 1 \end{pmatrix}.$$

The same calculation using SAGE:

```
sage: attach 'mq.py'
sage: R.<x0,x1> = PolynomialRing(GF(127),2,order="lex")
sage: F = MQ(R,[114 + 80*x0*x1 + x0^2, 29 + x1^2 + 107*x0*x1])
sage: A,v = F.coeff_matrix()
sage: A

[
1, 80, 0, 114,
0, 107, 1, 29
]

sage: v
 (x0^2, x0*x1, x1^2, 1)
```

2.3 Gröbner Bases and Solutions to $\mathcal{MQ}$ Problems

"Gröbner bases are standard bases of polynomial ideals that can be used for solving systems of polynomial equations. What Gaussian elimination does for systems of linear equations, Gröbner basis algorithms try to emulate for polynomial systems. Unfortunately the computational complexity of Gröbner basis algorithms for nonlinear systems is no longer polynomial." [BPW05]

To link Gröbner bases to solutions of a multivariate polynomial system an affine variety needs to be defined. For this the notion of the n-dimensional affine space is needed as this is where the solutions and cipher states are defined:

Definition 2.3.1. *Given a field k and a positive integer n, we define the n-dimensional* affine space *to be the set*

$$k^n = \{(a_0, \dots, a_{n-1}) : a_0, \dots, a_{n-1} \in k\}.$$

Evaluating a polynomial f at $(a_0, \dots, a_{n-1}) \in k^n$ is a function

$$f : k^n \to k,$$

where every x_i is replaced by a_i, for $0 \leq i < n$.

The set of all solutions to a system of equations

$$f_0(x_0, \dots, x_{n-1}) = \dots = f_{m-1}(x_0, \dots, x_{n-1}) = 0$$

is called an *affine variety*, explicitly defined as follows.

Definition 2.3.2. *Let k be a field and let $f_0, \ldots, f_{m-1}$ be polynomials in $k[x_0, \ldots, x_{n-1}]$. We define*

$$V(f_0, \ldots, f_{m-1}) = \{(a_0, \ldots, a_{n-1}) \in k^n : f_i(a_0, \ldots, a_{n-1}) = 0 \text{ for all } 0 \leq i < m\}.$$

We call $V(f_0, \ldots, f_{m-1})$ the affine variety defined by $f_0, \ldots, f_{m-1}$.

Definition 2.3.3 ($\mathcal{MQ}$)**.** *Given a finite list F of (at most) quadratic multivariate polynomials in P we call $\mathcal{MQ}$ the problem of finding the affine variety of F.*

To compute with affine varieties, the notion of ideals is also needed.

Definition 2.3.4 (Ideal)**.** *A subset $I \subset P$ is an ideal if it satisfies:*

1. *$0 \in I$;*
2. *If $f, g \in I$, then $f + g \in I$;*
3. *If $f \in I$ and $h \in P$, then $h \cdot f \in I$.*

The ideal generated by a finite number of polynomials is defined as follows:

Definition 2.3.5. *Let $f_0, \ldots, f_{m-1}$ be polynomials in P . Define the ideal*

$$\langle f_0, \ldots, f_{m-1} \rangle = \left\{ \sum_{i=0}^{m-1} h_i f_i : h_0, \ldots, h_{m-1} \in P \right\}.$$

If there exists a finite set of polynomials in P that generates a given ideal, this set is called a basis. The fundamental theorem in commutative algebra – the Hilbert Basis Theorem – states that every ideal in P is finitely generated:

Theorem 2.3.1 (Hilbert's Basis Theorem)**.** *Every ideal $I \subset P$ has a finite generating set. That is, $I = \langle g_0, \ldots, g_{m-1} \rangle$ for some $g_0, \ldots, g_{m-1} \in I$.*

Proof. See [CLO05, p. 74].

Please note, that a given ideal may have many different bases.

Lemma 2.3.2. *If $f_0, \ldots, f_{s-1}$ and $g_0, \ldots, g_{t-1}$ are bases of the same ideal in P, so that*

$$\langle f_0, \ldots, f_{s-1} \rangle = \langle g_0, \ldots, g_{t-1} \rangle$$

, then

$$V(f_0, \ldots, f_{s-1}) = V(g_0, \ldots, g_{t-1}).$$

Proof. Every $f \in \langle f_0, \ldots, f_{s-1} \rangle$ is also in $\langle g_0, \ldots, g_{t-1} \rangle$ and can therefore be expressed as

$$f = h_0 g_0 + \cdots + h_{t-1} g_{t-1}.$$

Hence, every $a = (a_0, \ldots, a_{n-1}) \in V(g_0, \ldots, g_{t-1})$ satisfies $f(a) = 0$ and vice versa for all $g \in \langle g_0, \ldots, g_{t-1} \rangle$. This shows that both varieties consist of the same points. □

Hilbert's Basis Theorem has two important consequences for Gröbner basis calculations. The first is that a nested increasing sequence of ideals $I_0 \subset I_1 \subset \ldots$ in P stabilizes at a certain point in time. Explicitly:

Theorem 2.3.3 (Ascending Chain Condition)**.** *Let*

$$I_0 \subset I_1 \subset I_2 \subset \dots$$

be an ascending chain of ideals in P. Then there exists an $N \geq 1$ such that

$$I_N = I_{N+1} = I_{N+2} = \dots \ .$$

Proof. See [CLO05, p.76].

A second consequence is that the variety corresponding to a set of polynomials F equals the variety of the ideal spanned by this set of polynomials.

Definition 2.3.6. *Let $I \subset P$ be an ideal. We define $V(I)$ to be the set*

$$\{(a_0, \dots, a_{n-1}) \in k^n : f(a_0, \dots, a_{n-1}) = 0 \text{ for all } f \in I\}.$$

Proposition 2.3.4. *$V(I)$ is an affine variety. In particular, if $I = \langle f_0, \dots, f_{m-1} \rangle$, then $V(I) = V(f_0, \dots, f_{m-1})$.*

Proof. See [CLO05, p.77]

So the set of equations – the $\mathcal{MQ}$ problem – may be considered as a basis of an ideal I. If there was a basis for the same ideal where the solution, i.e., the variety $V(I)$, can be read from directly, the $\mathcal{MQ}$ problem was solved.

Such a basis actually exists and is called a *reduced lexicographical Gröbner basis*. Thus, Gröbner bases and Buchberger's algorithm will now be introduced as building blocks to perform such basis transformations on ideals.

2.3.1 Gröbner Bases

Gröbner bases are defined as:

Definition 2.3.7 (Gröbner Basis)**.** *Fix a monomial order. A finite subset $G = \{g_0, \dots, g_{m-1}\}$ of an ideal I is said to be a* Gröbner basis *or standard basis if*

$$\langle LT(g_0), \dots, LT(g_{m-1}) \rangle = \langle LT(I) \rangle.$$

For instance a Gröbner basis of the canonical example is:

$$\begin{aligned} 0 &= 67 + 74x_1^2 + x_1^4, \\ 0 &= 17x_1 + 24x_1^3 + x_0. \end{aligned}$$

Gröbner bases have several interesting properties: The remainder r of the division of any $f \in P$ by G is unique and *reduced* Gröbner bases are a unique representation of an ideal with respect to a monomial ordering.

Definition 2.3.8 (Reduced Gröbner Basis)**.** *A* reduced Gröbner basis *for a polynomial ideal I is a Gröbner basis for G such that:*

1. *$LC(f) = 1$ for all $f \in G$;*
2. *For all $f \in G$, no monomial of f lies in $\langle LT(G - \{f\}) \rangle$.*

To compute a Gröbner basis and a reduced Gröbner basis in SAGE the following commands may be executed:

```
sage: I = F.ideal()
sage: gb = I.groebner_basis() # returns a list
sage: rgb = Ideal(gb).reduced_basis()
```

In 1965 Bruno Buchberger formulated an algorithmically verifiable criterion if a set of polynomials forms a Gröbner basis. This criterion naturally leads to Buchberger's algorithm for computing a Gröbner basis from a given ideal basis, so the main concepts of his criterion are introduced in the following section.

2.3.2 Buchberger's Criterion and Algorithm

By the definition of a Gröbner basis if an element in $\langle \mathrm{LT}(I) \rangle$ which satisfies $\notin \langle \mathrm{LT}(f_1), \ldots, \mathrm{LT}(f_t) \rangle$ can be constructed then G is not a Gröbner basis. So two appropriate elements of G may be chosen such that for the term $ax^\alpha f_i - bx^\beta f_j$ the $\mathrm{LT}(f_i)$ and $\mathrm{LT}(f_j)$ cancel each other out such that $\mathrm{LT}(ax^\alpha f_i - bx^\beta f_j) \notin \langle \mathrm{LT}(f_1), \ldots, \mathrm{LT}(f_t) \rangle$. As on the other hand $ax^\alpha f_i - bx^\beta f_j \in I$ the following is true: $\mathrm{LT}(ax^\alpha f_i - bx^\beta f_j) \in \langle \mathrm{LT}(I) \rangle$. So if these cancellations can be constructed this shows that G was not a Gröbner basis. S-polynomials are a way to construct these kinds of cancellations:

Definition 2.3.9 (S-Polynomial)**.**

Let $f, g \in k[x_1, \ldots, x_n]$ be polynomials $\neq 0$.

1. *If $\alpha = multideg(f)$ and $\beta = multideg(g)$ then let $\gamma = (\gamma_1, \ldots, \gamma_n)$ where $\gamma_i = max(\alpha_i, \beta_i)$ for every $i \leq n$. x^γ is then the least common multiple of $\mathrm{LM}(f)$ and $\mathrm{LM}(g)$, written as $x^\gamma = LCM(\mathrm{LM}(f), \mathrm{LM}(g))$.*

2. *The S-polynomial of f and g is defined as*

$$S(f,g) = \frac{x^\gamma}{\mathrm{LT}(f)} \cdot f - \frac{x^\gamma}{\mathrm{LT}(g)} \cdot g.$$

The following example illustrates that $S(f_i, f_j)$ is constructed in a way to allow cancellation of leading terms.

Example 2.3.1. *Let $f_1 = x^3 - 2xy$ and $f_2 = x^2y - 2y^2 + x$. The leading monomials with respect to degrevlex are $\mathrm{LM}(f_1) = x^3$ and $\mathrm{LM}(f_2) = x^2y$, so that $x^\gamma = x^3y$. The S-polynomial is:*

$$\begin{aligned}
S(f_1,f_2) &= \frac{x^\gamma}{\mathrm{LT}(f_1)} \cdot f_1 - \frac{x^\gamma}{\mathrm{LT}(f_2)} \cdot f_2 \\
S(f_1,f_2) &= \frac{x^3y}{x^3} \cdot (x^3 - 2xy) - \frac{x^3y}{x^2y} \cdot (x^2y - 2y^2 + x) \\
S(f_1,f_2) &= y \cdot (x^3 - 2xy) - x \cdot (x^2y - 2y^2 + x) \\
S(f_1,f_2) &= x^3y - 2xy^2 - x^3y + 2y^2x - x^2 \\
S(f_1,f_2) &= -x^2
\end{aligned}$$

The following lemma states that whenever terms chancel each other out in a polynomial this cancellation may be accounted to S-polynomials.

Lemma 2.3.5. *Let every element of $\Sigma_{i=1}^{t} c_i x^{\alpha(i)} g_i$ with constants $c_1, \ldots, c_n$, have multidegree δ if $c_i \neq 0$: $\alpha(i) + multideg(g_i) = \delta \in \mathbb{N}_0^n$. Now if the the multidegree of the sum is smaller then there exist constants c_{jk} such that*

$$\sum_{i=1}^{t} c_i x^{\alpha(i)} g_i \quad = \quad \sum_{j=1}^{t-1} c_{jk} x^{\delta - \gamma_{jk}} S(g_j, g_k) \tag{2.1}$$

Where $k = j+1$ and $x^{\gamma_{jk}} = LCM(\mathrm{LM}(g_j), \mathrm{LM}(g_k))$. Furthermore we have $multideg(f) < \delta$ for every $f = x^{\delta - \gamma_{jk}} S(g_j, g_k)$.

Proof. See [CLO05, p.81ff].

On the left hand side of Equation 2.1 the degrees get canceled after the addition while on the right hand side the terms already have lower multidegree, i.e., the terms are already canceled out. So the S-polynomials must be responsible for the cancellation.

Using S-polynomials and Lemma 2.3.5 on the preceding page one can formulate a criterion to decide whether a given set of equations is a Gröbner basis or not.

Theorem 2.3.6 (Buchberger's Criterion)**.** *Let I be an ideal. $G = \{g_1, \ldots, g_t\}$ is a Gröbner basis for I, if and only if for all pairs $i \neq j$, the remainder r of the division of $S(g_i, g_j)$ by G (listed in some order) is zero, written as*

$$\overline{f}^{G} = 0.$$

Proof. See [CLO05, p.82ff]

But there is another criterion which can be checked to verify if a given set of polynomials forms a Gröbner basis or not. For that criterion the expression *f reduces to zero modulo G* is needed.

Definition 2.3.10. *[CLO05, p.100] Fix a monomial order and let $G = \{g_0, \ldots, g_{m-1}\} \subset P$. Given a polynomial $f \in P$, we say that f reduces to zero modulo G, written*

$$f \underset{G}{\longrightarrow} 0,$$

if f can be written in the form

$$f = a_0 g_0 + \cdots + a_{m-1} g_{m-1},$$

such that whenever $a_i g_i \neq 0$, we have

$$multideg(f) \geq multideg(a_i g_i).$$

Please note, that $\overline{f}^{G} = 0$ implies $f \underset{G}{\longrightarrow} 0$ but the converse is false in general. Using this definition Buchberger's Criterion may be reformulated as follows:

Theorem 2.3.7. *A basis $G = \{g_0, \ldots, g_{m-1}\}$ for an ideal I is a Gröbner basis if and only if $S(g_i, g_j) \underset{G}{\longrightarrow} 0$ for all $i \neq j$.*

The proof of this theorem follows directly from the proof of Buchberger's criterion in [CLO05]. This criterion and the Ascending Chain Condition leads to the following algorithm for computing Gröbner basis:

Algorithm 1 (Buchberger's Algorithm)**.**

```
def buchberger(F):
  """
  INPUT:
      F -- a finite subset of P[x]

  OUTPUT:
      A Groebner basis for the ideal <F>.
  """
  G = F
  G2 = set()
  while G2!=G
    G2 = G
    for p in G2:
      for q in G2:
        if p!=q:
          S = "S-pol(p,q) reduced modulo G2"
          if S != 0:
            G.add(S)
  return G
```

The correctness and termination of this algorithm may be derived from the following three observations:

1. At every stage of the algorithm, $G \subset I$ and $\langle G \rangle = I$ hold;
2. If $G2 = G$ then $S(p,q) \xrightarrow[G2]{} 0$ for all $p, q \in G$ and, by Buchberger's criterion, G is a Gröbner basis at this moment;
3. The equality $G2 = G$ occurs in finitely many steps since the ideals $\{LT(G)\}$, from successive iterations of the loop, form an ascending chain. Due to the Ascending Chain Condition, this chain of ideals stabilizes after a finite number of iterations and at that moment $\langle LT(G) \rangle = \langle LT(G2) \rangle$ holds.

Even though this algorithm terminates eventually it is well known that it's runtime is not polynomial as the intermediate bases $G2$ grow exponentially during the calculations. However, Buchberger's algorithm leaves a lot of freedom when implemented. The runtime is heavily influenced by the following choices:

- the order in which the critical pairs p, q are selected,
- a criterion to avoid useless reductions to 0,
- the monomial ordering of P. This influences the run time of the algorithm dramatically. Normally calculating a degree reverse lexicographical Gröbner basis is way faster than computing a lexicographical Gröbner basis. Algorithms exist (see Jean-Charles Faugère, P. Gianno, P. Lazard, and T. Mora [FGLM93] so as S. Collart, M. Kalkbrener, and D. Mall [CKM97]) to convert a Gröbner basis (of a zero-dimensional ideal) in one monomial order to a Gröbner basis in another monomial order.

Gröbner bases turn out to be a useful tool to solve the $\mathcal{MQ}$. The following section describes this relationship.

2.3.3 Solving $\mathcal{MQ}$ with Gröbner Bases

First, more notation needs to be established: Given an ideal I in a polynomial ring $k[x_0, \ldots, x_{n-1}]$ over a field k and a number $j \in \{0, \ldots, n-2\}$, consider the set of all polynomials in I which involve only the indeterminates $x_0, \ldots, x_j$. This set $I \cap k[x_0, ..., x_j]$ is an ideal in $k[x_0, \ldots, x_j]$. It is called the elimination ideal of I with respect to the indeterminates $x_j, \ldots, x_{n-1}$ because passing from I to this ideal means eliminating all polynomials in which one of these latter indeterminates occurs.

Definition 2.3.11 (Elimination Ideal)**.** *Given $I = \langle f_0, \ldots, f_{m-1} \rangle \subset k[x_0, \ldots, x_{n-1}]$, the l-th elimination ideal I_l is the ideal of $k[x_{l+1}, \ldots, x_{n-1}]$ defined by*

$$I_l = I \cap k[x_{l+1}, \ldots, x_{n-1}].$$

Furthermore, the notion of a perfect field will be needed:

Definition 2.3.12 (Perfect Field)**.** *A field k is called a perfect field if either its characteristic is 0 or its characteristic is $p > 0$ and $k = k^p$, i.e. every element has a p-th root in k.*

Please note, that finite fields $k = GF(q)$, where $q = p^e$ and $e > 0$, are perfect since the map $x \to x^{p^{e-1}}$ provides the p-th roots, because $(x^{p^{e-1}})^p = x$ for all $x \in k$.

It turns out to be important whether the system of equations corresponding to the cryptographic problem describes a finite set of solutions. The ideal spanned by the corresponding polynomials of such a system will be called *zero-dimensional*. The following proposition provides an algorithmic criterion for finiteness.

Lemma 2.3.8 (Finiteness Criterion)**.** *Let $>$ be an ordering on the monomials $M(P)$ of the polynomial ring $P = k[x_0, \ldots, x_{n-1}]$. For a system of equations corresponding to an ideal $I = f_0, \ldots, f_{m-1}$, the following conditions are equivalent.*

1. *The system of equations has only finitely many solutions.*
2. *For $i = 0, \ldots, n-1$, we have $I \cap k[x_i] \neq 0$.*
3. *The set of monomials $M(P) \setminus \{LM_>(f) : f \in I\}$ is finite.*
4. *The k-vector space P/I is finite-dimensional.*

Proof. See [KR00, p.243ff].

Notice that Buchberger's Algorithm is able to test condition 3 of this lemma.

Furthermore, appending the field equations to an ideal will assure that the ideal is zero-dimensional as in this case condition 2 is satisfied. Those field equations are defined as follows:

Definition 2.3.13. *Let k be a field with order $q = p^n$, p prime and $n > 0$. Then the* field polynomials *of the ring $k[x_0, \ldots, x_{n-1}]$ are defined as the set*

$$\{x_0^q - x_0, \ldots, x_{n-1}^q - x_{n-1}\}.$$

The ideal spanned by this set

$$\langle x_0^q - x_0, \ldots, x_{n-1}^q - x_{n-1} \rangle$$

is called the field ideal *of $k[x_0, \ldots, x_{n-1}]$.*

Corollary 2.3.9. *Let I be an ideal in $k[x_0, \ldots, x_{n-1}]$. The ideal spanned by the generators of I and generators of the field ideal has the same variety over k as the ideal I but excludes all elements from $\overline{k}$, the algebraic closure of k.*

Proof. Every finite field k satisfies $x^q = x$ for every $x \in k$ where q is the order of k. Thus the equations $x_i^q - x_i = 0 : 0 \leq i < n$ are satisfied for every possible value in k and especially for every element of $V(I)$. Also $x_i^q - x_i = 0$ factors completely over k and thus no element of $\overline{k}$ satisfies it. □

For information about the possible polynomials occurring in the ideal described by a set of polynomials, the following theorem is of great importance. It states that a polynomial over an algebraically closed field having common zeros with the polynomials in $F = \{f_0, \ldots, f_{m-1}\}$, occurs to some power in the ideal spanned by F .

Theorem 2.3.10 (Hilbert's Nullstellensatz)**.** *Let k be an algebraically closed field. If f and $f_0, \ldots, f_{m-1} \in P$ are such that $f \in I(V(f_0, \ldots, f_{m-1}))$, then there exists an integer $e \geq 1$ such that*

$$f^e \in \langle f_0, \ldots, f_{m-1} \rangle$$

and conversely.

Proof. See [CLO05, p.171].

The set of polynomials satisfying this condition are called the radical of the ideal I.

Definition 2.3.14. *Let $I \subset P$ be an ideal. The radical of I denoted by $\sqrt{I}$, is the set*

$$\{f : f^e \in I \text{ for some integer } e \geq 1\}.$$

Lemma 2.3.11. *$\sqrt{I}$ is an ideal.*

Proof. See [CLO05, p.174].

Consider a cryptosystem over $k = GF(q)$, for q the power of a prime p. Suppose $F = \{f_0, \ldots, f_{m-1}\} \in \overline{k}[x_0, \ldots, x_{n-1}]$ and the equations

$$\begin{aligned} y_0 &= f_0(x_0, \ldots, x_{n-1}) \\ y_1 &= f_1(x_0, \ldots, x_{n-1}) \\ &\vdots \\ y_{m-1} &= f_{m-1}(x_0, \ldots, x_{n-1}) \end{aligned}$$

represent the key and state bits of a block cipher. Since the state and key bits are elements of k, possible solutions existing in $\overline{k} \setminus k$ are not of interest. Therefore – due to Seidenberg's Lemma –, appending the set

$$\{x_i^q - x_i : 0 \leq i < n\}$$

to F, creates a radical ideal from which the state and key bits are still solvable.

Proposition 2.3.12 (Seidenberg's Lemma)**.** *Let k be a field, let $P = k[x_0, \ldots, x_{n-1}]$, and let $I \subset P$ be a zero-dimensional ideal. Suppose that for every $i \in 0, \ldots, n-1$ there exists a non-zero polynomial $g_i \in I \cap k[x_i]$ such that the greatest common divisor (GCD) of g_i and its derivative equals 1. Then I is a radical ideal.*

Proof. See [KR00, p.250ff]

By adding the field equations there exist g_i as defined in the previous proposition which are relatively prime to their derivative. Therefore, the ideal I is radical and, due to the Finiteness Criterion, zero-dimensional. Furthermore, since $x_i^q - x_i$ factors completely over k, the corresponding variety V does not contain points $p \in V$ with coordinates in $\overline{k} \setminus k$.

As an example consider $\mathbb{F}_7[x]$:

```
sage: P.<x> = PolynomialRing(GF(7))
sage: f = x^7 - x # field polynomial
sage: f.factor()
 x * (x + 1) * (x + 2) * (x + 3) * (x + 4) * (x + 5) * (x + 6)
sage: f.diff(x)
 6
sage: gcd(f,f.diff(x))
 1
```

The following Shape Lemma shows that the lexicographic Gröbner basis of the ideal I has a triangular form.

Theorem 2.3.13 (The Shape Lemma)**.** *Let k be a perfect field, let $I \subset P$ be a zero-dimensional radical ideal. Let $g_{n-1} \in k[x_{n-1}]$ be the monic generator of the elimination ideal $I \cap k[x_{n-1}]$, and let $d = deg(g_{n-1})$. Then the following statements are true:*

1. *The reduced Gröbner basis of the ideal I with respect to the lexicographic ordering $x_0 > \cdots > x_{n-1}$ is of the form*
$$\{x_0 - g_0, \ldots, x_{n-2} - g_{n-2}, g_{n-1}\},$$
where $g_0, \ldots, g_{n-1} \in k[x_n]$;
2. *The polynomial g_{n-1} has d distinct zeros $a_0, \ldots, a_{d-1} \in k$, and the set of zeros of I is $\{(g_0(a_i), \ldots, g_{n-2}(a_i), a_i) : i = 0, \ldots, d-1\}$.*

Proof. See [KR00, p.257]

This lemma states that a lexicographical Gröbner basis G for the zero-dimensional radical ideal spanned by the polynomials of the $\mathcal{MQ}$ problem and the generators of the field ideal allows to read the solution to the $\mathcal{MQ}$ problem from G. To illustrate consider this example.

```
sage: attach "ctc.py"
sage: ctc=CTC(B=1,Nr=1)
sage: F = ctc.MQ_factory(p=[1,0,0],k=[0,1,0],order="degrevlex")
sage: I = F.ideal()
sage: I += sage.rings.ideal.FieldIdeal(F.ring) # field equations
sage: I.radical() == I # I is a radical ideal
 True
sage: I.dimension() # I is zero-dimensional
 0
sage: gb = I.groebner_basis()
sage: Ideal(gb).reduced_basis() # reduced Groebner basis
[K000002,
 1 + K000001,
 K000000,
 K001002,
 K001001,
 1 + K001000,
 Z001002,
 1 + Z001001,
 Z001000,
 1 + Y001002,
 Y001001,
 Y001000,
 X001002,
 1 + X001001,
 1 + X001000]
```

The solution $(0, 1, 0)$ may be read directly from the equations $0 = K_{000000}$, $0 = 1 + K_{000001}$, and $0 = K_{000002}$ which are included in the calculated reduced lexicographical Gröbner basis.

A general Gröbner basis attack algorithm on the $\mathcal{MQ}$ problem may then be described as follows:

Algorithm 2 (Gröbner basis Attack)**.** [BPW05]

1. Set up a polynomial system $P = \{p_i = 0\}$ for the cipher in question. The system P consists of both cipher and key schedule equations.
2. Request a plaintext/ciphertext pair $((P_0, \dots P_{Bs-1}), (C_0, \dots, C_{Bs-1}))$. This gives rise to the following additional system of linear equations $G = \{g_i = 0\}$:

$$\begin{array}{ccc} x_{0,0} + P_0 = 0 & \dots & x_{N_r,0} + C_0 = 0 \\ \vdots & & \vdots \\ x_{0,Bs-1} + P_0 = 0 & \dots & x_{N_r,Bs-1} + C_0 = 0 \end{array}$$

   Let I be the ideal generated by the set of polynomials $L = (\bigcup_i \{p_i\}) \cup (\bigcup_i \{g_i\}) \cup K$, where K is the set of field equations for every variable occurring in every p_i. We call this ideal the key recovery ideal.
3. Compute a degree-reverse lexicographic Gröbner basis $G_{degrevlex}$ of I. For ciphers using a multiplicative inverse as S-Box function, the system may be inconsistent, resulting in $G_{degrevlex} = 1$.
4. If $G_{degrevlex} = 1$ go to Step 2, otherwise proceed.
5. Use a Gröbner basis conversion algorithm to obtain a lexicographical Gröbner basis G_{lex} from $G_{degrevlex}$. The variable ordering should be such that the key variables of the first round are the least elements.
6. Compute the variety Z of I using the Gröbner basis G_{lex}.
7. Request another plaintext/ciphertext pair (P, C).
8. Try all elements $k \in Z$ as key candidates to encrypt P. If k does not encrypt P to C, remove k from Z, otherwise retain.
9. If Z contains more than one element, go to step 7.
10. Terminate

This algorithm is very general. Many alterations and tweaks are possible: Leaving the field equations out, only calculating the *degrevlex* Gröbner basis, or computing a Gröbner basis with respect to a totally different monomial ordering than *lex* and *degrevlex*.

Please note, that in the step where the variety Z of I is computed a considerable amount of complexity is hidden. This step requires to factor univariate polynomials and substitute their roots in other equations to check whether this root is a solution to other equations as well. If a system has many solutions the complexity of this step increases rapidly. However, the number of solutions to those systems dealt with in this thesis is considered to be low as it is equal to the number of distinct keys which encrypt the same plaintext to the same ciphertext.

Chapter 3

Equation Systems for the CTC

In this chapter the Courtois Toy Cipher (CTC) is presented. It will be shown how to derive a quadratic equation system or an ideal from it. This ideal is called the CTC ideal in this thesis. This description is based on [Cou06]. Also a quick presentation of results from a linear cryptanalysis of CTC (see [DK]) will be presented, which show that the cipher is not resistant against linear cryptanalysis. Finally, alternative representations of CTC as multivariate polynomial systems are presented of which one is a zero-dimensional Gröbner basis representation of low degree not found in literature so far.

3.1 The Courtois Toy Cipher (CTC)

In this thesis a variable denoted K_{001002} is identified with the variable $K_{1,2}$ i.e. the first three digits of the six digit index represent the first index and the next three digits represent the second index. Also $CTC_{3,b,nr}$ represents a CTC equation system with $B = b$ and $N_r = nr$ where B denotes one third of the block size and N_r denotes the number of rounds as explained below.

3.1.1 Design Rationales

In [Cou06] Nicolas Courtois presents the following design rationales for the Courtois Toy Cipher:

1. "It should be very simple, practical, and be implemented with a minimal effort.
2. It should be in general very much like any other known block cipher. If the parameters are large enough it should evidently be secure against all known attacks on block ciphers.
3. For simplicity, the key size should be equal to block size.
4. It should have a variable number of rounds and variable number of S-boxes in each round. However since it is a "research cipher" it is not required that it must encrypt 128-bit blocks. It can use for example 129-bit blocks (in fact it will be any multiple of 3).
5. The S-box should be chosen as a random permutation, and thus have no special structure.
6. Yet this S-box should exhibit an "algebraically vulnerability", by which we mean that it should be described by a small system of multivariate non-linear equations. This is made possible in spite of (5.) because the size of the S-box is quite small.

7. The diffusion should be very good: full avalanche effect should be achieved after about 3-4 rounds.
8. However, at the same time, the diffusion should not be too good, so that the linear parts of the cipher can still be described by (linear) equations that remain quite sparse. (In CTC each bit in the next round is an XOR of two bits from the outputs of two S-boxes from the previous round).
9. Finally and importantly, the cipher should allow to handle complete experimental algebraic attacks on block ciphers using a standard PC, with a reasonable quantity of RAM, and not more than a handful of plaintext / ciphertext pairs."

[Cou06]

To summarize, CTC is designed to be broken by an algebraic attack while it is supposed to be secure against any other attack. Section 3.2 shows that the latter requirement is not fullfilled by CTC.

3.1.2 Cipher Description

The cipher operates on block sizes which are multiples of 3. So the block size is $B \cdot s$ where $s = 3$ and B may be chosen. The cipher is defined in rounds where each round performs the same operation on the input data, except that a different round key is added each time. The number of rounds is denoted by N_r. The output of round $i-1$ is the input of round i. Each round consists of parallel applications of B S-boxes (S), the application of the linear diffusion layer (D), and a final key addition of the round key. Also, a round key K_0 is added to the plaintext block before the first round. The plaintext bits $p_0 \dots p_{Bs-1}$ are identified with $Z_{0_0} \dots Z_{0_{Bs-1}}$ and the ciphertext bits $c_0 \dots c_{Bs-1}$ are identified with $X_{N_r+1_0} \dots X_{N_r+1_{Bs-1}}$ to have an uniform notation. A graphical representation of this cipher is given in figure 3.1.

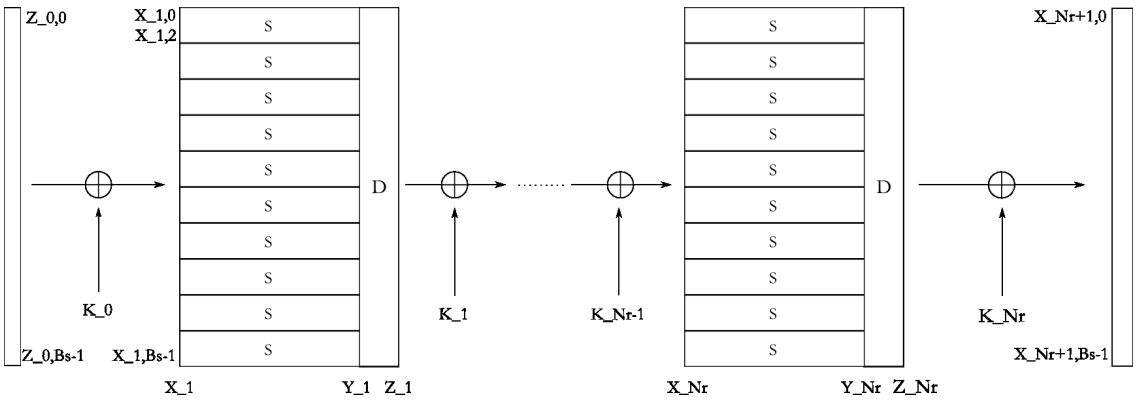

Figure 3.1: CTC Overview for B=10

S-box

The S-box (S) is defined over $\mathbb{F}_{2^3}$ as the non-linear random permutation $[7, 6, 0, 4, 2, 5, 1, 3]$. The transformation from $(\mathbb{F}_2)^3$ to $\mathbb{F}_{2^3}$ is the "natural"-mapping $x = 4X_3 + 2X_2 + X_1$ and $y = 4Y_3 + 2Y_2 + Y_1$ where x and y are the input and the output of the S-box respectively and X_1, X_2, X_3 and

Y_1, Y_2, Y_3 are the input and output bits respectively.This S-box gives $r = 14$ quadratic equations in $t = 22$ terms over $\mathbb{F}_2$. These equations are according to [Cou06]:

$$\begin{aligned}
0 =& X_1X_2 + Y_1 + X_3 + X_2 + X_1 + 1,\\
0 =& X_1X_3 + Y_2 + X_2 + 1,\\
0 =& X_1Y_1 + Y_2 + X_2 + 1,\\
0 =& X_1Y_2 + Y_2 + Y_1 + X_3,\\
0 =& X_2X_3 + Y_3 + Y_2 + Y_1 + X_2 + X_1 + 1,\\
0 =& X_2Y_1 + Y_3 + Y_2 + Y_1 + X_2 + X_1 + 1,\\
0 =& X_2Y_2 + X_1Y_3 + X_1,\\
0 =& X_2Y_3 + X_1Y_3 + Y_1 + X_3 + X_2 + 1,\\
0 =& X_3Y_1 + X_1Y_3 + Y_3 + Y_1,\\
0 =& X_3Y_2 + Y_3 + Y_1 + X_3 + X_1,\\
0 =& X_3Y_3 + X_1Y_3 + Y_2 + X_2 + X_1 + 1,\\
0 =& Y_1Y_2 + Y_3 + X_1,\\
0 =& Y_1Y_3 + Y_3 + Y_2 + X_2 + X_1 + 1,\\
0 =& Y_2Y_3 + Y_3 + Y_2 + Y_1 + X_3 + X_1.
\end{aligned}$$

The Lagrange interpolation polynomial of this S-box in $\mathbb{F}_{2^3}[x]$ – with a being a root of the minimal polynomial $z^3 + z + 1$ – is

$$f = x^6 + ax^5 + (a+1)\,x^4 + \left(a^2 + a + 1\right)x^3 + \left(a^2 + 1\right)x^2 + (a+1)\,x + a^2 + a + 1.$$

This S-box has also been used in [CP02a] by Nicolas Courtois to describe a toy cipher. [BDC03] describes a way to construct a basis for a quadratic equation system from a given S-box using this S-box as an example.

The Diffusion Layer (D)

The diffusion layer (D) is defined as:

$$\begin{aligned}
Z_{i,(257\%Bs)} =& Y_{i,0} \text{ for all } i = 1 \ldots N_r,\\
Z_{i,(j\cdot 1987 + 257\%Bs)} =& Y_{i,j} + Y_{i,(j+137\%Bs)} \text{ for } j \neq 0 \text{ and all } i.
\end{aligned}$$

where $Y_{i,j}$ represents input bits and $Z_{i,j}$ represents output bits.

Key Addition

Key addition is performed bit-wise, so:

$$X_{i+1,j} = Z_{i,j} + K_{i,j} \text{ for all } i = 0 \ldots N_r \text{ and } j = 0 \ldots Bs - 1,$$

Where $Z_{i,j}$ represents output bits of the previous diffusion layer, $X_{i+1,j}$ the input bits of the next round, and $K_{i,j}$ the bits of the current round key. These round keys are generated in the key schedule.

Key Schedule

The key schedule is a simple permutation of wires, defined by:

$$K_{i,j} = K_{0,j+i\%Bs} \text{ for all } i \text{ and } j.$$

3.1.3 Example

For an illustration how to put these equations together consider the following example for $B = 1$ and $N_r = 1$. The initial key addition is expressed through:

$$\begin{aligned}0 &= K_{000000} + Z_{000000} + X_{001000},\\ 0 &= K_{000001} + Z_{000001} + X_{001001},\\ 0 &= K_{000002} + Z_{000002} + X_{001002}.\end{aligned}$$

The S-Box is represented as:

$$\begin{aligned}0 &= 1 + Y_{001000} + X_{001002} + X_{001001} + X_{001000} + X_{001000}X_{001001},\\ 0 &= 1 + Y_{001001} + X_{001001} + X_{001000}X_{001002},\\ 0 &= 1 + Y_{001001} + X_{001001} + X_{001000}Y_{001000},\\ 0 &= Y_{001001} + Y_{001000} + X_{001002} + X_{001000}Y_{001001},\\ 0 &= 1 + Y_{001002} + Y_{001001} + Y_{001000} + X_{001001} + X_{001001}X_{001002} + X_{001000},\\ 0 &= 1 + Y_{001002} + Y_{001001} + Y_{001000} + X_{001001} + X_{001001}Y_{001000} + X_{001000},\\ 0 &= X_{001001}Y_{001001} + X_{001000} + X_{001000}Y_{001002},\\ 0 &= 1 + Y_{001000} + X_{001002} + X_{001001} + X_{001001}Y_{001002} + X_{001000}Y_{001002},\\ 0 &= Y_{001002} + Y_{001000} + X_{001002}Y_{001000} + X_{001000}Y_{001002},\\ 0 &= Y_{001002} + Y_{001000} + X_{001002} + X_{001002}Y_{001001} + X_{001000},\\ 0 &= 1 + Y_{001001} + X_{001002}Y_{001002} + X_{001001} + X_{001000} + X_{001000}Y_{001002},\\ 0 &= Y_{001002} + Y_{001000}Y_{001001} + X_{001000},\\ 0 &= 1 + Y_{001002} + Y_{001001} + Y_{001000}Y_{001002} + X_{001001} + X_{001000},\\ 0 &= Y_{001002} + Y_{001001} + Y_{001001}Y_{001002} + Y_{001000} + X_{001002} + X_{001000}.\end{aligned}$$

The diffusion layer consists of three linear equations:

$$\begin{aligned}0 &= Z_{001000} + Y_{001001} + Y_{001000},\\ 0 &= Z_{001001} + Y_{001002} + Y_{001001},\\ 0 &= Z_{001002} + Y_{001000}.\end{aligned}$$

The key addition of the first round:

$$\begin{aligned}0 &= K_{001000} + Z_{001000} + X_{002000},\\ 0 &= K_{001001} + Z_{001001} + X_{002001},\\ 0 &= K_{001002} + Z_{001002} + X_{002002}.\end{aligned}$$

Finally the key schedule equations:

$$\begin{aligned}0 &= K_{001000} + K_{000001},\\ 0 &= K_{001001} + K_{000002},\\ 0 &= K_{001002} + K_{000000}.\end{aligned}$$

The commands to produce these equation systems using the provided implementation of this thesis are:

```
sage: attach "ctc.py" # load CTC implementation
sage: ctc = CTC(B=1,Nr=1)
sage: R = ctc.ring_factory(pc=True) # treat plain/ciphertext as variables
sage: F = ctc.MQ_factory()
sage: F.gens() # returns polynomials as listed above
...
sage: F,s = ctc_MQ(B=1,Nr=1) # random MQ with pc=False
sage: s # solution
 {K000000: 1, K000001: 1, K000002: 1}
```

plaintext	key	ciphertext	plaintext	key	ciphertext
0	0	1	4	0	4
0	4	5	4	4	0
0	2	4	4	2	6
0	6	7	4	6	5
0	1	4	4	1	5
0	5	4	4	5	5
0	3	3	4	3	5
0	7	4	4	7	2
2	0	0	6	0	2
2	4	3	6	4	1
2	2	5	6	2	0
2	6	1	6	6	4
2	1	7	6	1	1
2	5	0	6	5	6
2	3	0	6	3	1
2	7	0	6	7	1
1	0	6	5	0	7
1	4	6	5	4	7
1	2	1	5	2	7
1	6	6	5	6	0
1	1	3	5	1	6
1	5	7	5	5	2
1	3	6	5	3	4
1	7	5	5	7	7
3	0	5	7	0	3
3	4	2	7	4	4
3	2	2	7	2	3
3	6	2	7	6	3
3	1	2	7	1	0
3	5	1	7	5	3
3	3	7	7	3	2
3	7	3	7	7	6

Figure 3.2: Plaintext, Key, Ciphertext Tuples for $CTC_{3,1,1}$

3.1.4 The Number of Solutions

As stated earlier the number of solutions to the $\mathcal{MQ}$ problem has an impact on the performance of algebraic attack techniques. As CTC ideals are defined over $\mathbb{F}_2$ every polynomial may have at most two distinct solutions/zeros if those solutions from the algebraic closure are excluded which are not interesting for the purpose of breaking CTC. It might be tempting to consider only equations with an unique solution – 0 or 1 in $\mathbb{F}_2$ – for each variable as other equations don't restrict the solution space at all. For examples, equations of the form $x^2 + x = 0$ do not restrict the solution space as both 1 and 0 are solutions to that equation. This strategy is for example chosen in [Seg04] but unfortunately will not work for many CTC instances. As an example a list of all plaintext, key, and ciphertext tuples for the configuration $B = 1$ and $N_r = 1$ is provided in Figure 3.2. In this table (0,0,0) is identified with 0, (0,0,1) is identified with 1 and so on up to (1,1,1) which is identified with 7.

Figure 3.2 shows that for the configuration $Nr = 1$ and $B = 1$ CTC encrypts one plaintext to the same ciphertext for up to four distinct keys. This shows that it is not safe to assume that a

key bit may only be either zero or one. Thus polynomials with two distinct roots and may not be discarded.

As an example consider a $CTC_{1,1,1}$ instance which encrypts the bits $0,0,0$ using the key $0,0,1$. The ciphertext are the bits $1,0,0$. The reduced lexicographical Gröbner basis for the matching CTC ideal is:

$$
\begin{aligned}
&K_{000002} + K_{000002}^2,\\
&1 + K_{000002} + K_{000001} + K_{000001}K_{000002},\\
&K_{000001} + K_{000001}^2,\\
&K_{000000} + K_{000000}K_{000002},\\
&1 + K_{000002} + K_{000001} + K_{000000}K_{000001},\\
&K_{000000} + K_{000000}^2,\\
&K_{000000} + K_{001002},\\
&K_{000002} + K_{001001},\\
&K_{000001} + K_{001000},\\
&K_{000000} + Z_{001002},\\
&K_{000002} + Z_{001001},\\
&1 + K_{000001} + Z_{001000},\\
&1 + K_{000002} + K_{000001} + K_{000000} + Y_{001002},\\
&1 + K_{000001} + K_{000000} + Y_{001001},\\
&K_{000000} + Y_{001000},\\
&K_{000002} + X_{001002},\\
&K_{000001} + X_{001001},\\
&K_{000000} + X_{001000}.
\end{aligned}
$$

This example was calculated using the following commands in SAGE:

```
sage: attach "ctc.py"
sage: ctc=CTC(Nr=1,B=1)
sage: R = ctc.ring_factory(term_order="lex")
sage: F = ctc.MQ_factory(R,p=[0,0,0],k=[0,0,1])
sage: gb = Ideal(F.ideal().groebner_basis())
sage: gb.reduced_basis()
  ...
```

3.2 Linear and Differential Cryptanalysis of CTC

In [Cou06] Nicolas Courtois expresses the assumption that CTC is resistant against all known attack techniques besides algebraic attacks. However, later Orr Dunkelman and Nathan Keller showed how to break CTC using linear cryptanalysis. Their results are presented in this section.

Recall that the S-box of the CTC is $S[i] = [7,6,0,4,2,5,1,3]$. Now consider the relationship between the most significant input bit X_3 and the least significant output bit Y_1 of this S-box. They are equal with probability $\frac{1}{2} + \frac{1}{4}$. Now consider a CTC instance with 85 S-boxes per round and six rounds; This is the instance Nicolas Courtois has broken in his public demonstration. Now consider bit 2 of the first round which is the most significant bit of the first S-box. It is equal to bit 0 (the least significant output bit of the first S-box) with bias $\frac{1}{4}$. Please note, that by construction $Z_{i,(257\%3*85)} = Z_{i,2} = Y_{i,0}$ for all $i = 1 \ldots N_r$ so that bit 2 of the output of the first round equals bit 2 of the input to the second round with bias $\frac{1}{4}$. So this linear approximation is an iterative one and can be extended across as many rounds as needed.

For an r-round approximation from bit 2 of the input to bit 2 of the output, the basic approximation is concatenated r times, resulting in a linear approximation with bias $2^{-(r+1)}$. According to [DK] this approximation can be used to attack $r+1$ with about 2^{2r+4} known plaintexts, and time complexity of about $2^{2r+4} \cdot 2^3$ partial decryptions of one S-box (about $\frac{2^{(2r+4)}}{10r}$ full r-round encryptions). This attack retrieves the equivalent of 3 key bits and the parity of another r key bits. Thus, the attack on a 85 S-box, 6 round CTC – denoted $CTC_{3,85,6}$ – requires about 2^{14} known plaintexts, and has a running time of about 2^8 encryptions.

The authors of [DK] furthermore state: "We note that if the difference distribution table of the S-box used in $CTC_{3,85,6}$ had a non-zero probability in the entry corresponding to input difference in the middle bit and output difference in the most significant bit, an iterative differential characteristic could be constructed. This characteristic would be based on having an input difference in bit 136, that becomes a difference in bit 137 after the S-box, and returns to a difference in bit 136 after the linear transformation." [DK]

Following these results, CTC is not "secure against all known attacks" [Cou06] as initially hoped by Nicolas Courtois. However, it is unclear if this has any impact on the performance of the "Fast Algebraic Attack against Blockciphers" or the speculations Nicolas Courtois made in [Cou06] regarding the applicability of his attack against AES which is secure against linear and differential cryptanalysis.

3.3 Quotient Rings and the Field Ideal

So far CTC ideals were defined in the polynomial ring $P = \mathbb{F}_2[x_0, \ldots, x_{n-1}]$ in this thesis. Another representation is achieved by defining them in the quotient ring $Q = R/FI$ where FI denotes the field ideal of P. Reasons to switch to this representation are the ability to represent polynomials in a more performant way in the computer or the possibility to benefit from specialized algorithms or implementations over the ring Q.

To further motivate this section, please note that Michael Brickenstein [Bri06] provided a Singular script which transforms polynomial equation systems over $\mathbb{F}_{2^n}$ to polynomial equation systems over $\mathbb{F}_2$ by using the "natural mapping" between $\mathbb{F}_{2^n}$ and $(\mathbb{F}_2)^n$. So for example consider $k = \mathbb{F}_{2^3}$ with the generator a. First the element x from $k[x]$ is mapped to $a^2x_2 + ax_1 + x_0$ and then the three bit components are treated separately as x_2, x_1, x_0. So the ideal in $\mathbb{F}_2$ matching the ideal $\langle x \rangle$ in $\mathbb{F}_{2^3}$ is $\langle x_0, x_1, x_2 \rangle$. Using this conversion, ideals over finite extension field with characteristic 2 may benefit from any computational progress made in the quotient ring Q. For example BES-style ideals (see [MR02]) may be translated to ideals over $\mathbb{F}_2$ using Michael Brickenstein's idea and implementation, possibly resulting in an alternative way of describing AES over $\mathbb{F}_2$.

To present the concept of quotient rings the term *congruency modulo an ideal* needs to be defined.

Definition 3.3.1. *Let $I \subset P$ be an ideal, and let $f, g \in P$. We say f and g are* congruent modulo *I, written*

$$f \equiv g \ \% \ I,$$

if $f - g \in I$.

It might be counter intuitive at first that no division is involved in this relationship but consider that e.g. $15 \equiv 1 \bmod 7$ and that $15 - 1 = 14 = 2 \cdot 7$. So 14 is in the ideal spanned by 7.

This definition defines an equivalence relation on P:

Proposition 3.3.1. *[CLO05, p.219] Let $I \subset P$ be an ideal. The congruence modulo I is an equivalence relation on P.*

Proof. See [CLO05, p.219]

An equivalence relation on the set S partitions this set into a collection of disjoint subsets called equivalence classes. For any $f \in P$, the class of f is the set

$$[f] = \{g \in P : g \equiv f \ \% \ I\}$$

Definition 3.3.2. *The quotient of $k[x_0, \ldots, x_{n-1}]$ modulo I, written $k[x_0, \ldots, x_{n-1}]/I$, is the set of equivalence classes for congruence modulo I:*

$$k[x_0, \ldots, x_{n-1}]/I = \{[f] : f \in k[x_0, \ldots, x_{n-1}]\}.$$

In $P = k[x_0, \ldots, x_{n-1}]/I$ addition and multiplication may be defined as follows:

$$\begin{aligned} [f] + [g] &= [f + g] \\ [f] \cdot [g] &= [f \cdot g]. \end{aligned} \tag{3.1}$$

These definitions are independent from the choice of the representant of $[f]$ and $[g]$: f, g.

Proposition 3.3.2. *[CLO05, p.220] The operations defined in equations (3.1) yield the same classes in P/I on the right hand sides no matter which $f' \in [f]$ and $g' \in [g]$ we use. (We say that the operations on classes given in (3.1) are well-defined on classes.)*

Proof. [CLO05, p.220] If $f' \in [f]$ and $g' \in [g]$, then $f' = f + a$ and $g' = g + b$, where $a, b \in I$. Hence,

$$f' + g' = (f + a) + (g + b) = (f + g) + (a + b).$$

Since we also have $a + b \in I$ (I is an ideal), it follows that $f' + g' \equiv f + g \ \% \ I$, so $[f' + g'] = [f + g]$. Similarly,

$$f' \cdot g' = (f + a) \cdot (f + b) = fg + ag + fb + ab.$$

Since $a, b \in I$, we have $ag + fb + ab \in I$. Thus, $f' \cdot g' \equiv f \cdot g \ \% \ I$ and $[f + g] = [f' + g']$. □

As the operations (3.1) are well-defined it is easy to see that all axioms of a commutative ring are satisfied for P/I, as all operations may be reduced to operations in P which is a commutative ring.

Theorem 3.3.3. *[CLO05, p.221] Let I be an ideal in $k[x_0, \ldots, x_{n-1}]$. The quotient*

$$k[x_0, \ldots, x_{n-1}]/I$$

is a commutative ring under the sum and product operations given in (3.1).

Consequently $Q = P/I = k[x_0, \ldots, x_{n-1}]/I$ may be called a *quotient ring*. In this thesis $P = k[x_0, \ldots, x_{n-1}]$ is called its cover ring and I its defining ideal.

As Q is a commutative ring, ideals may be constructed in it with the usual properties of ideals. These ideals have a close relationship with ideals in the cover ring P.

Theorem 3.3.4. *[CLO05, p.223] Let I be an ideal in $k[x_0, \ldots, x_{n-1}]$. The ideal in the quotient ring $k[x_0, \ldots, x_{n-1}]/I$ are in one-to-one correspondence with the ideal of $k[x_0, \ldots, x_{n-1}]$ containing I (that is, the ideals J satisfying $I \subset J \subset P$).*

Proof. [CLO05, p.223] First, we give a way to produce an ideal in $k[x_0, \ldots, x_{n-1}]/I$ corresponding to each J containing I in $k[x_0, \ldots, x_{n-1}]$: Given an ideal J in $k[x_0, \ldots, x_{n-1}]$ containing I, let J/I denote the set $\{[j] \in k[x_0, \ldots, x_{n-1}]/I : j \in J\}$. We claim that J/I is an ideal in $k[x_0, \ldots, x_{n-1}]/I$.

To prove this, first note that $[0] \in J/I$ since $0 \in J$. Next, let $[j], [k] \in J/I$. Then $[j]+[k] = [j+k]$ by definition of the sum in $k[x_0, \ldots, x_{n-1}]/I$. Since $j, k \in J$ we have $j + k \in J$ as well. Hence, $[j] + [k] \in J/I$. Finally, if $[j] \in J/I$ and $[r] \in k[x_0, \ldots, x_{n-1}]/I$, then $[r] \cdot [j] = [r \cdot j]$ by the definition of the product in $k[x_0, \ldots, x_{n-1}]/I$. But $r \cdot j \in J$ since J is an ideal in $k[x_0, \ldots, x_{n-1}]$. Hence, $[r] \cdot [j] \in J/I$. As a result J/I is an ideal in $k[x_0, \ldots, x_{n-1}]/I$.

If $\tilde{J} \in k[x_0, \ldots, x_{n-1}]/I$ is an ideal, we next show how to produce an ideal $J \subset k[x_0, \ldots, x_{n-1}]$ which contains I. Let $J = \{j \in k[x_0, \ldots, x_{n-1}] : [j] \in \tilde{J}\}$. Then we have $I \subset J$ since $[i] = [0] \in \tilde{J}$ for any $i \in I$. It remains to show that J is an ideal of $k[x_0, \ldots, x_{n-1}]$. First note that $0 \in I \subset J$. Furthermore, if $j, k \in J$, then $[j], [k] \in \tilde{J}$ implies that $[j] + [k] = [j + k] \in \tilde{J}$. It follows that $j + k \in J$. Finally, if $j \in J$ and $r \in k[x_0, \ldots, x_{n-1}]$, then $[j] \in \tilde{J}$, so $[r][j] = [rj] \in \tilde{J}$. But this means $rj \in J$, and, hence, J is an ideal in $k[x_0, \ldots, x_{n-1}]$.

This shows that there are correspondences between the two collections of ideals:

$$\begin{aligned} \{J : I \subset J \subset k[x_0, \ldots, x_{n-1}]\} \qquad & \qquad \{\tilde{J} \subset k[x_0, \ldots, x_{n-1}]/I\} \\ J \longrightarrow \qquad & \qquad J/I = \{[j] : j \in J\} \\ J = \{j : [j] \in \tilde{J}\} \longleftarrow \qquad & \qquad \tilde{J}. \end{aligned}$$

From the proof of each direction it is easy to see that each of these arrows is the inverse of the other. This gives the desired one-to-one correspondence. □

This shows that Gröbner basis calculations may be performed in the quotient ring P/I where I the field ideal of P under the condition that the field equations are allowed to be added to the ideal.

3.3.1 Representing Monomials in P/I as Bitstrings

The one-to-one correspondence presented in the last section allows a much more effective representation of polynomials and thus speeds up computations involving them. By performing calculations in the quotient ring, every variable may have at most degree $q - 1$, with q being the order of the field. Thus the degree of every polynomial in the course of all involved calculations is bound to $n \cdot (q - 1)$ if n is the number of variables in the ring $k[x_0, \ldots, x_{n-1}]$.

This is especially useful over $\mathbb{F}_2$ – the base ring of CTC ideals – as the highest possible degree of a variable is bound to 1. Therefore, monomials in $\mathbb{F}_2[x_0, \ldots, x_{n-1}]$ may be represented as bitstrings of length n. This idea is used by the F_4 implementation provided in [Shi] and also for the class `MPolynomialGF2` provided with this thesis.

As an example consider monomials in $\mathbb{F}_2[x_0, \ldots, x_3]/\langle x_0^2 + x_0, \ldots, x_3^2 + x_3\rangle$. Multiplication may be identified with bitwise logical OR as e.g.,

$$\begin{aligned} x_0x_2 \cdot x_1x_2 &= x_0x_1x_2 \\ 0b1010 \; OR \; 0b0110 &= 0b1110 \end{aligned}$$

Furthermore, bitwise logical XOR may be identified with division if and only if f is divisible by g:

$$\begin{aligned} x_0x_2/x_0 &= x_2 \\ 0b1010 \; XOR \; 0b1000 &= 0b0010 \end{aligned}$$

To test for divisibility [(left XOR right) AND (NOT left)] may be performed:

$$x_0|x_0x_2 = True$$
$$(0b1000\ XOR\ 0b1010\ AND\ (NOT\ 0b1010)) = True$$

Addition is performed by equality check:

$$x_0 + x_0 = 0$$
$$\text{if f == g then 0 else \{f,g\}.}$$

On top of that monomial representation, polynomials may be represented as lists, sets, balanced binary trees, etc. of monomials.

Using this representation, monomial multiplication of monomials of up to 32 variables may be performed in one CPU instruction on a 32-bit CPU, 64 variables on a 64-bit CPU, and 128 variables when using enhanced instruction sets like SSE2 on Intel CPUs or AltiVec on PowerPCs. Even though this only provides a constant speed-up factor for e.g. Gröbner basis calculations, it is noticeable in practice as the following example suggests. However, to put these benchmarks in perspective, please note that the current multivariate polynomial arithmetic implemented in SAGE is not very fast.

```
sage: attach "ctc.py"
sage: attach "polyf2.spyx"
sage: attach "f4.py"
sage: ctc = CTC(Nr=2,qring=True)
sage: R = ctc.ring_factory()
sage: F = ctc.MQ_factory(k=[1,0,1],p=[0,1,0])
sage: f4=F4()
sage: time gb = f4.groebner(F,Update=f4.update_pairsGF2)
CPU times: user 0.14 s, sys: 0.00 s, total: 0.14 s
Wall time: 0.15

sage: MixInSAGE()
sage: ctc = CTC(Nr=2)
sage: R = ctc.ring_factory()
sage: F = ctc.MQ_factory(k=[1,0,1],p=[0,1,0])
sage: time gb = f4.groebner(F,Update=f4.update_pairs)
CPU times: user 1.78 s, sys: 0.00 s, total: 1.78 s
Wall time: 1.78
```

3.4 Reduced Size CTC Ideals

In [MR02] Sean Murphey and Matt Robshaw note: "We can, of course, immediately reduce the sizes of these multivariate quadratic systems by using the linear equations to substitute for state and key variables, though the resulting system is slightly less sparse." [MR02]

If algorithms are employed which do not exploit the sparseness of the attacked system this strategy may be provide a slight speed-improvement. This is especially true for CTC as its linear equations are very sparse. In the provided CTC implementation the user may chose three different substitution levels: `subst=0` is equivalent to no substitution at all, i.e., the equation system as described earlier. `subst=1` is equivalent to a substitution as defined by the linear equations in

Section 3.1.2, e.g. the $Y_{i,j}$ variables are used to substitute $Z_{i,j}$. `subst=2` is equivalent to choosing three equations per S-box to substitute quadratic equations as well. Those equations are exactly:

$$\begin{aligned} Y_1 &= X_1 * X_2 + X_3 + X_2 + X_1 + 1, \\ Y_2 &= X_1 * X_3 + X_2 + 1, \\ Y_3 &= X_2 * X_3 + Y_2 + Y_1 + X_2 + X_1 + 1. \end{aligned}$$

All substitutions are performed as long as the result of the substitution does not match the input before the substitution. Thus, `subst=2` results in an equation system in the key variables only. However, as three equations out of 14 are chosen per S-box, it is not guaranteed that only solutions are found in any following attack which also solve the original set of equations. This is because there are fewer constrains on the variables in any `subs=2` system than in the matching `subst=0` system. So correctness is not guaranteed with `subst=2`.

Random substituted equation systems may be constructed as follows:

```
sage: F,s = ctc_MQ(B=1,Nr=1,subst=0)
sage: F,s = ctc_MQ(B=1,Nr=1,subst=1)
sage: F,s = ctc_MQ(B=1,Nr=1,subst=2)
```

Please note, that for $N_r > 1$ `subst=2` may take very long as the substitution code is very inefficient. However, those equation systems may be constructed in the substituted form in the first place instead of substituting the generated equation system. Consequently, the time used to construct the systems doesn't have to be considered attack time.

These substitutions may improve some algebraic attacks as Figure 3.3 suggests for an XL attack. XL was chosen for this demonstration as it takes no advantage at all of the sparseness of the systems. Figure 3.3 shows XL attacks against $CTC_{3,B,1}$ for `subst=0`, `subst=1`, and `subst=2`. Five trials were taken per run.

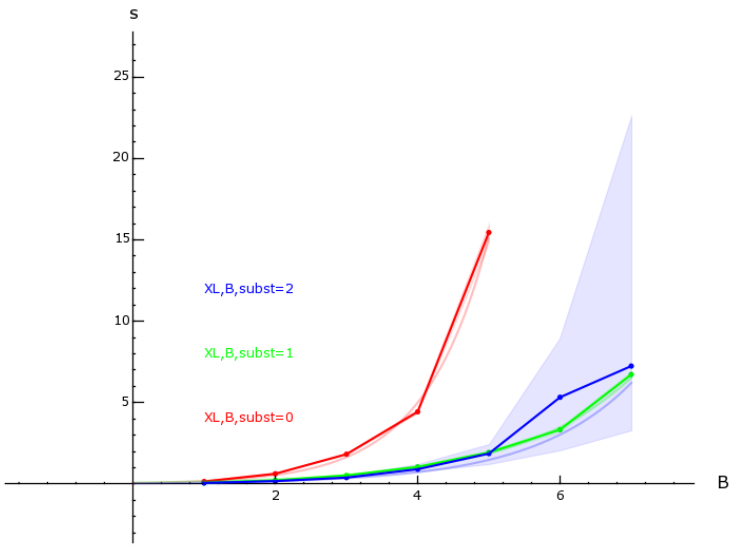

Figure 3.3: XL attack against CTC with $N_r = 1$ for $subst = 0 \ldots 2$

The exponential least-square fits of the average runtime of these three experiments are

$$t = 0.059 * e^{1.109*B} \text{ if } subst = 0,$$
$$t = 0.070 * e^{0.650*B} \text{ if } subst = 1,$$
$$t = 0.058 * e^{0.709*B} \text{ if } subst = 2.$$

Figure 3.3 also shows that the best and the worst case for `subst=2` differ significantly which may be accounted to Step 3 (see 12 on page 56) in the XL algorithm after the row reduction. As many univariate polynomials found have multiple roots which do not solve the system, most time is spent testing and neglecting these wrong solutions. An improved technique for choosing solutions and testing them would improve these attacks. Also using several plaintext - ciphertext pairs at once could help: As these systems only contain key variables, one may choose to produce more of them with different plaintext and ciphertext pairs and identify the key variables in each system with the key variables in any other system to put more constraints on those key variables. `subst=1` does not seem to suffer from the instability of `subst=2`, though its best case is worse than `subst=2`'s best case. The exponential least-square fits suggest that `subst=1` is the best choice as it has the smallest exponent.

Figure 3.4 shows the times Singular needs to compute a *degrevlex* Gröbner basis for $CTC_{3,B,1}$ with *subst* in $0, 1, 2$. Singular's Buchberger algorithm also seems to perform better with `subst=1` than with `subst=0` for these CTC ideal bases. However, these plots are less conclusive than the XL plots as the computational time to compute a Gröbner basis is for example affected by the monomial ordering including the ordering of the variables ($x > y$ vs. $y > x$).

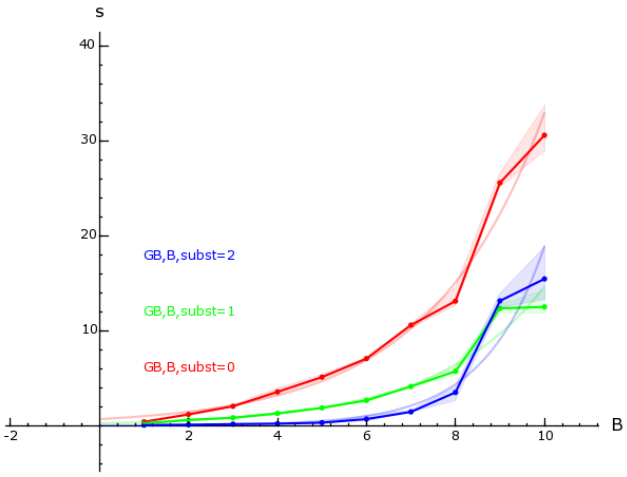

Figure 3.4: Calculating a *degrevlex* Gröbner Basis for CTC with $N_r = 1$ for $subst = 0 \ldots 2$

Overall, it seems like `subst=1` is a good choice for substitution if CTC ideal bases are attacked with an algorithm that does not take advantage of the sparseness of the polynomial system.

The Figures 3.5 and 3.6 support these assumptions about `subst=1` for CTC ideals with $B = 1$ and N_r variable. However if B is fixed and N_r variable, `subst=2` is worse than `subst=0`. Please note, that N_r was cut at 3 as the substitution code is too slow for configurations with $N_r > 3$.

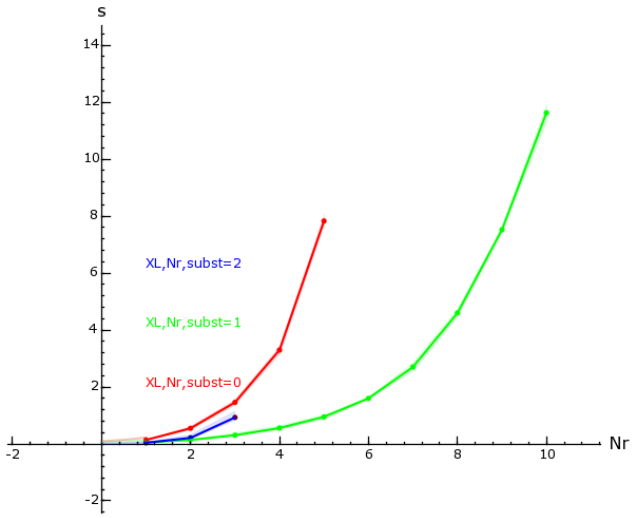

Figure 3.5: XL attack against CTC with $B = 1$ for $subst = 0 \ldots 2$

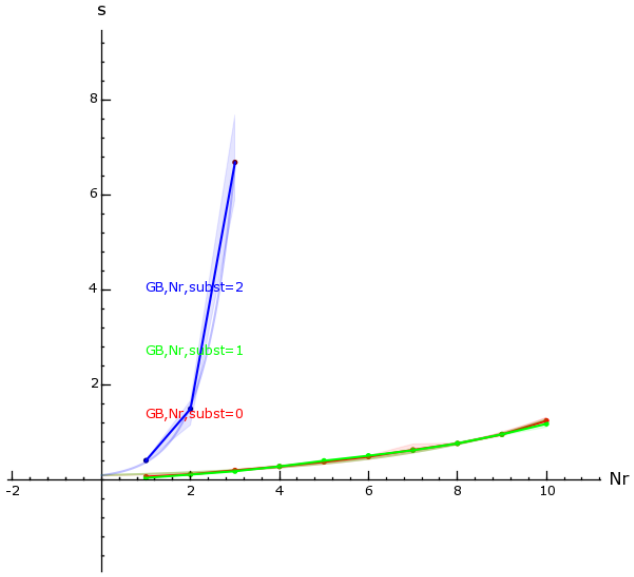

Figure 3.6: Calculating a *degrevlex* Gröbner Basis for CTC with $B = 1$ for $subst = 0 \ldots 2$

3.5 Variable Ordering

When computing a Gröbner basis variable ordering may have a huge impact on the runtime of the calculation. Consider this example:

```
sage: ctc=CTC(Nr=6)
sage: R = ctc.ring_factory(order="lex")
sage: F = ctc.MQ_factory(R,p=[1,1,0],k=[1,0,1])
sage: time gb1 = F.ideal().groebner_basis()
CPU times: user 0.76 s, sys: 0.05 s, total: 0.81 s
Wall time: 3.97

sage: ctc=CTC(Nr=6)
sage: R = ctc.ring_factory2(order="lex") #notice the _2_
sage: F = ctc.MQ_factory(R,p=[1,1,0],k=[1,0,1])
sage: time gb2 = F.ideal().groebner_basis()
CPU times: user 0.67 s, sys: 0.05 s, total: 0.72 s
Wall time: 13.51
```

`ctc.ring_factory` and `ctc.ring_factory2` produce the same rings but with different variable orderings. The default variable ordering in this thesis is:

$$\begin{aligned} &X_{1,0} > \cdots > X_{1,Bs-1} > \cdots > X_{i,0} > \cdots > X_{i,Bs-1} > \ldots X_{N_r,0} > \cdots > X_{N_r,Bs-1} \\ &> \cdots > Y_{0,j} > \cdots > Y_{i,j} > \cdots > Y_{N_r,j} > \cdots > Z_{i,j} \\ &> K_{N_r,0} > \cdots > K_{N_r,Bs-1} > \cdots > K_{0,0} > \ldots K_{0,Bs-1}. \end{aligned}$$

This means that every $X_{i,j} > Y_{i,j} > Z_{i,j} > K_{i,j}$ for all $0 \leq i \leq N_r$ and $0 \leq j < Bs$. Also $X_{i,j} > X_{i,j+1}$, $Y_{i,j} > Y_{i,j+1}$, $Z_{i,j} > Z_{i,j+1}$, and $K_{i,j} > K_{i,j+1}$ for a given i, j and $X_{i,j} > X_{i+1,j}$, $Y_{i,j} > Y_{i+1,j}$, $Z_{i,j} > Z_{i+1,j}$ but $K_{i+1,j} > K_{i,j}$. This is the variable ordering which is produced by the `ctc.ring_factory` method.

`ctc.ring_factory2` on the other hand produces a variable ordering which may be used to construct a CTC ideal basis which is a Gröbner basis, as shown in the next section.

3.6 Gröbner Basis Equation Systems for the CTC

In [BPW05] Johannes Buchmann, Andrei Pychkine, and Ralf-Philipp Weinmann present a way to bring multivariate polynomial equation systems derived from block ciphers to a "Gröbner basis form" without a single polynomial reduction. This sections presents the necessary background to perform this construction and a zero-dimensional Gröbner basis for CTC ideals. This approach is also used in [BPW06] and [CMR06] to construct Gröbner bases of degree 254 for the AES without polynomial reduction. Please note, that the system constructed in this section for CTC is still quadratic.

To prove that the constructed system is actually a Gröbner basis, Buchberger's criterion needs to be revisited.

Lemma 3.6.1 (First Buchberger Criterion)**.** *[BPW05, p.11] Suppose that we have $f, g \in G$, such that the leading monomials of f and g are pairwise prime. Then the S-polynomial of f and g reduces to zero.*

Please note, that the statement that $LM(f)$ and $LM(g)$ are pairwise prime is equivalent to the statements:

- $LCM(LM(f), LM(g)) = LM(f) \cdot LM(g)$, and
- (by definition) the greatest common divisor of $LM(f)$ and $LM(g)$ is 1.

The proof of Buchberger's first criterion follows:

Proof. [CLO05, p.101] For simplicity, assume that f, g have been multiplied by appropriate constants to make $LC(f) = LC(g) = 1$. Write $f = LM(f) + p, g = LM(g) + q$. Then, since $LCM(LM(f), LM(g)) = LM(f) \cdot LM(g)$, the following statement is true:

$$\begin{aligned} S(f,g) &= LM(g) \cdot f - LM(f) \cdot g \\ &= (g - q) \cdot f - (f - p) \cdot g \\ &= g \cdot f - q \cdot f - f \cdot g + p \cdot g \\ &= p \cdot g - q \cdot f \end{aligned} \tag{3.2}$$

The claim is:

$$deg(S(f,g)) = max(deg(p \cdot g), deg(q \cdot f)). \tag{3.3}$$

Note that (3.2) and (3.3) would imply $S(f,g) \underset{G}{\longrightarrow} 0$ since $f, g \in G$. To prove (3.3), observe that in the last polynomial of (3.2), the leading monomials of $p \cdot g$ and $q \cdot f$ are distinct and, hence, cannot cancel. If the leading monomials were the same, we would have

$$LM(p) \cdot LM(g) = LM(q) \cdot LM(f)$$

which is impossible if $LM(f)$ and $LM(g)$ are relatively prime: from the last equation, $LM(g)$ would have to divide $LM(q)$, which is absurd since $LM(g) > LM(q)$. □

Theorem 2.3.7 on page 17 stated that a set G is a Gröbner basis if all $S(f,g)$ in G reduce to zero for $f, g \in G$ and $f \neq g$. Thus, if all $f, g, \in G : f \neq g$ have pairwise prime leading monomials, G is a Gröbner basis. A monomial ordering ensuring that all leading monomials are pairwise prime thus would provide a Gröbner basis without any polynomial reduction.

Such a monomial ordering may be chosen for CTC if slight alterations to the involved equations are allowed. Consider any CTC ideal basis. The following steps are iterated N_r times during a CTC encryption: the diffusion layer, the S-boxes, the key schedule, and the subkey addition. Now consider each separately:

diffusion layer This is a linear layer with $Y_{i,j}$ as input variables and $Z_{i,j}$ as output variables where $0 \leq j < Bs$ and $1 \leq i \leq N_r$. There are Bs equations each relating up to two input variables to one output variable. So every monomial ordering with $Y_{i,j} < Z_{i,j}$ for all $0 \leq j < Bs$ and $1 \leq i \leq N_r$ produces Bs equations with Bs pairwise prime leading monomials. The head monomials are $Z_{i,j}$.

key schedule The key schedule relates variables $K_{i,j}$ to $K_{0,j}$ for $0 \leq j < Bs$ and $1 \leq i \leq N_r$. So every monomial ordering with $K_{i+1,j} > K_{i,j}$ for $0 \leq i < N_r$ and $0 \leq j < Bs$ will produce Bs equations with Bs pairwise prime leading monomials. The head monomials are $K_{i,j}$.

key addition Similar to the diffusion layer these linear equations relate variables $Z_{i-1,j}$, $K_{i-1,j}$, and $X_{i,j}$ for $0 \leq j < Bs$ and $1 \leq i \leq N_r$. More specifically these equations relate sums of $K_{i-1,j}$ and $Z_{i-1,j}$ variables to $X_{i,j}$. So every monomial order with $Z_{i-1,j} < X_{i,j}$ and $K_{i-1,j} < X_{i,j}$ will produce Bs equations with pairwise prime leading monomials. The head monomials are $X_{i,j}$.

S-boxes The approach of just ensuring that the output variables $Y_{i,j}$ are greater than the input variables $X_{i,j}$ – with respect to the chosen monomial order – doesn't work here. But note that there are several S-box equations with univariate monomials in $Y_{i,j}$. Three of those per S-box – one for each output bit of one S-box – may be chosen and the monomial ordering fixed to be *lex* with $Y_{i,j} > X_{i,j}$. A total degree monomial order – such as *degrevlex* – doesn't work

in this case as there are always higher degree monomials in $X_{i,j}$ than univariate monomials $Y_{i,j}$ involved in any S-box equation. As shown below this approach does not provide a zero-dimensional ideal due to problems in the last round when using a *lex* monomial ordering.

However, every $Y_{i,j}$ may be replaced with $Y_{i,j}^2$ in the S-box equations as in $\mathbb{F}_2$ the equation $x^2 = x$ is true for all $x \in \mathbb{F}_2$. In that case the univariate monomials $Y_{i,j}^2$ are greater than any $X_{i,j}X_{i,k}$ monomial for $0 \leq j, k < Bs$ and $1 \leq i \leq N_r$ if $Y_{i,j} > X_{i,j}$. The head monomials are $Y_{i,j}$.

To summarize, a monomial ordering as

$$K_{0,j} < \cdots < X_{i,j} < Y_{i,j} < Z_{i,j} < K_{i,j} < \cdots < X_{N_r,j} < Y_{N_r,j} < Z_{N_r,j} < K_{N_r,j}$$

is required. That is exactly the order in which the variables appear during the encryption process.

Please note, that special care has to be taken of the last round. Here, in the final key addition step, no variables $X_{N_r+1,j}$ are available (as they are constants) and both $Z_{N_r,j}$ and $K_{N_r,j}$ have already been used as head monomials. Again $x^2 = x : \forall x \in \mathbb{F}_2$ may be used: Use the relationship between $K_{N_r,j}$ and $K_{0,j}$ to replace $K_{N_r,j}$ in the last key addition equations with $K_{0,j}$. Now replace all $K_{0,j}$ by $K_{0,j}^2$ in those equations to ensure those are the leading monomials. The $K_{0,j}$ variables have not yet been used as head monomials so this approach is valid. However, this requires to use a total degree monomial order like *deglex* or *degrevlex*.

Call the polynomial ring with the presented *degrevlex* variable ordering P. Then the approach produces $4Bs \cdot N_r + Bs$ equations with $4Bs \cdot N_r + Bs$ different univariate and thus pairwise prime leading monomials. In this thesis the ideal generated by those $4Bs \cdot N_r + Bs$ equations is denoted the *CTCgb ideal*. The presented basis is a Gröbner basis which immediately follows from the fact that all leading monomials are pairwise prime and Buchberger's first criterion holds. The CTCgb ideal is furthermore zero-dimensional as $I \cap k[x_i] \neq 0$ for every variable x_i in the ring P. Also all terms appearing in the basis of the just constructed CTCgb ideal are at most quadratic. Please note, that identity of the original CTC ideal and the ideal spanned by the CTCgb Gröbner basis may not occur as information was omitted about the S-boxes when picking only Bs equations from $14 \cdot B$ possible S-box equations.

As an example consider $CTC_{3,1,1}$. Fix a *degrevlex* term ordering as above with $K_{001002} > K_{001001} > K_{001000} > Z_{001002} > Z_{001001} > Z_{001000} > Y_{001002} > Y_{001001} > Y_{001000} > X_{001002} > X_{001001} > X_{001000} > K_{000002} > K_{000001} > K_{000000}$. If $p = [1, 0, 1]$ and $k = [0, 1, 1]$ then the following equation system is produced:

The initial key addition is unmodified:

$$\begin{aligned} 0 &= 1 + K_{000000} + X_{001000}, \\ 0 &= K_{000001} + X_{001001}, \\ 0 &= 1 + K_{000002} + X_{001002}. \end{aligned}$$

Only three S-box equations are used and all $Y_{i,j}$s are squared:

$$\begin{aligned} 0 &= X_{001002} + Y_{001000}^2 + Y_{001001} + Y_{001001} * X_{001000}, \\ 0 &= 1 + X_{001001} + X_{001002} * X_{001000} + Y_{001001}^2, \\ 0 &= X_{001000} + Y_{001001} * Y_{001000} + Y_{001002}^2. \end{aligned}$$

The diffusion layer equations are untouched:

$$\begin{aligned} 0 &= Y_{001000} + Y_{001001} + Z_{001000}, \\ 0 &= Y_{001001} + Y_{001002} + Z_{001001}, \\ 0 &= Y_{001000} + Z_{001002}. \end{aligned}$$

Again, the key addition equations are untouched:

$$0 = K_{000001} + K_{001000},$$
$$0 = K_{000002} + K_{001001},$$
$$0 = K_{000000} + K_{001002}.$$

In the last round the variables $K_{0,j}$ are used as leading monomials as described above.

$$0 = 1 + K_{000001}^2 + Z_{001000},$$
$$0 = K_{000002}^2 + Z_{001001},$$
$$0 = K_{000000}^2 + Z_{001002}.$$

The same example may computed using the provided software:

```
sage: ctc = CTC(B=1,Nr=1)
sage: P = ctc.ring_factory2(order='degrevlex') # choose variable ordering
sage: F = ctc.MQgb_factory(P, p=[1,0,1], k=[0,1,1]) # choose equations
sage: I = F.ideal()
sage: I.is_groebner()
 True
sage: I.dimension()
 0

sage: Ir = Ideal(I.reduced_basis()) # FGLM needs reduced
sage: Il = Ideal(Ir.transformed_basis('fglm')) # lex ordering
sage: Il = Il + sage.rings.ideal.FieldIdeal(Il.ring()) # more readable
sage: Il.reduced_basis()
[K000000,  1 + K000001,  1 + K000002,  1 + X001000,  1 + X001001,  X001002,
 Y001000,  Y001001,  1 + Y001002,  Z001000,  1 + Z001001,  Z001002,
 1 + K001000,  1 + K001001,  K001002]
```

Another bigger example:

```
sage: ctc=CTC(B=3,Nr=6)
sage: p=[1,0,1,0,1,0,1,0,1] ; k=[0,1,1,0,1,1,0,1,1]
sage: P = ctc.ring_factory2(order='degrevlex') # choose variable ordering
sage: F = ctc.MQgb_factory(P, p=p, k=k) # choose equations
sage: F
 Multivariate polynomial equation system with 225 variables \
 and 225 polynomials (gens).
sage: I = F.ideal()
sage: I.is_groebner()
 True
sage: I.dimension()
 0
```

An analysis if this Gröbner basis for the CTC can be used to successfully attack CTC is found in Section 4.4.3.

Chapter 4

Algorithms for Algebraic Attacks

This chapter presents standard algebraic attack algorithms (Section 4.1, Section 4.2, Section 4.3) so as some specialized attacks (Section 4.4) against CTC ideals. After each algorithm has been described an implementation is benchmarked against CTC ideals to provide an estimate of the performance. Also theoretical performance measures are provided where appropriate. However, for several algorithms presented in this thesis only toy implementations – i.e. not very optimized implementations – are available such that the performance presented through benchmarks may be misleading.

4.1 Linking Linear Algebra to Gröbner Bases: F_4

This section briefly describes the basic and the improved version of Faugère's F_4 algorithm and roughly follows [Seg04, Section 4] for this. F_4 was first described by its author Jean-Charles Faugère in his paper "A new efficient algorithm for computing Gröbner bases (F_4)" [Fau99], where he introduces a powerful reduction strategy for Gröbner basis algorithms. This reduction strategy is based on linking Gröbner Bases to linear algebra and enables us to reduce several S-polynomials at once instead of one by one.

4.1.1 The Original F_4

Given a finite list F of polynomials in R, call the (reduced) Gröbner basis of these polynomials $\tilde{F}$. A coefficient matrix $\tilde{A}$ may be constructed for $\tilde{F}$. This matrix $\tilde{A}$ is the (reduced) row echelon form of A and $\tilde{F}$ is called the *row echelon basis* of F.

Conversely, $A = A_F$ may be constructed for F and the (reduced) row echelon form for A called $\tilde{A}$ may be computed. Then $\tilde{F}$ constructed from $\tilde{A}$ is called the *row echelon form* of F. One interesting property of row echelon forms of F is:

Let $\tilde{F}^+$ denote the set

$$\{g \in \tilde{F} : LM(g) \notin LM(F)\}.$$

The elements of $\tilde{F}^+$ are joined with a subset H of the original F, such that:

$$LM(H) = LM(F) \text{ and } |H| = |LM(F)|$$

holds. Then the ideal $\langle F \rangle$ is spanned by $H \cup \tilde{F}^+$. Formally:

Theorem 4.1.1. *[Fau99, p.4] Let k be a field and F a finite set of elements in $R = k[x_0, \ldots, x_{n-1}]$. Let A be the coefficient matrix of F and $\tilde{A}$ the row echelon form of this matrix. Finally, let $\tilde{F}$ be the finite list of polynomials corresponding to $\tilde{A}$.*

For any subset $H \subseteq F$ such that $LM(H) = LM(F)$ and $|H| = |LM(F)|$, $G = \tilde{F}^+ \cup H$ is a triangular basis of the $R-$module V_A generated by F. That is to say, for all $f \in V_A$ there exists $(\lambda_k)_k$ elements of R and $(g_k)_k$ elements of G such that $f = \sum_k \lambda_k g_k$, $LM(g_1) = LM(f)$, and $LM(g_k) > LM(g_{k+1})$.

Proof. [Seg04, p.58] Write $G = \tilde{F}^+ \cup H$. All elements g of G have distinct leading terms and are linear combinations of elements of F. Hence, the matrix $A_{\tilde{F^+} \cup H}$ has full rank and spans a subspace of the space spanned by the matrix A_F. Also $LM(G) = LM(\tilde{F}^+) \cup LM(H) = LM(\tilde{F})$ holds, which implies $|LM(G)| = |LM(\tilde{F})|$ and the theorem follows. □

Instead of computing the reduction of every S-polynomial individually, F_4 creates a selection of critical pairs $p_{ij} = (f_i, f_j)$, for f_i, f_j in the intermediate basis G' and passes the two polynomials

$$\frac{LCM(LM(f_i), LM(f_j))}{LM(f_i)} \cdot f_i, \frac{LCM(LM(f_i), LM(f_j))}{LM(f_j)} \cdot f_j$$

to the reduction function. The selection strategy recommended by Faugère in [Fau99] is the *normal selection strategy*:

Definition 4.1.1 (Normal Strategy)**.** *Let P be a list of critical pairs and let $LCM(p_{ij})$ denote the least common multiple of the leading monomials of the two parts of the critical pair $p_{ij} = (f_i, f_j)$. Further let $d = min\{deg(LCM(p)), p \in P\}$ denote the minimal degree of those least common multiples of p in P. Then the normal selection strategy selects the subset P_d of P with $P_d = \{p \in P | deg(LCM(p)) = d\}$.*

Definition 4.1.2. *Let p_{ij} denote a critical pair f_i, f_j as above. $Left(p_{ij})$ denotes the pair $(m_i, f_i) \in T \times R$ where $m_i = LCM(p_{ij})/LM(f_i)$ and $Right(p_{ij})$ denotes the pair (m_j, f_j) where $m_j = LCM(p_{ij})/LM(f_j)$. These definitions are extended to sets of critical pairs by applying them to their members individually. L_d denotes the union of $Left(P_d) \cup Right(P_d)$.*

Now that critical pairs to reduce are selected, reductors need to be added to the intermediate basis G' to reduce those pairs. The addition of reductors is done by a routine called *Symbolic Preprocessing*.

Definition 4.1.3 (Reductor)**.** *During the execution of an algorithm to compute Gröbner Bases, a reductor r of the set F is a polynomial satisfying*

$$LM(r) \in M(F) \setminus LM(F).$$

Algorithm 3 (*Symbolic Preprocessing$_o$*)**.**

```
def symbolic_preprocessing(L,G):
  """
  INPUT:
      L -- a finite subset of M x R
      G -- a finite subset of R

  OUTPUT:
      a finite subset of R
  """
  F = set([ t * f for (t,f) in L])
  Done = LM(F)
  while LM(F) != Done:
    m = (M(F).difference(Done)).pop()
    Done.add(m)
    if m "is divisible by an element" g in LM(G):
      m2 = m/LM(g)
      F.add(m2*f)
  return F
```

Symbolic Preprocessing is used by a function called *Reduction* that simultaneously reduces polynomials corresponding to several critical pairs.

Algorithm 4 (*Reduction*$_o$)**.**

```
def reduction(L,G):
  """
  INPUT:
      L -- a finite subset of M x R
      G -- a finite subset of R

  OUTPUT:
      a finite subset of R
  """
  F = symbolic_preprocessing(L,G)
  Ftilde = "Reduction to Row Echelon Form of F w.r.t. <"
  Ftildeplus = set([f for f in Ftilde if LM(f) not in LM(F)])
  return Ftildeplus
```

S-polynomials that do not reduce to zero in Buchberger's Algorithm, extend the ideal spanned by the leading terms of the intermediate basis. This way, an ascending chain of leading term ideals is obtained. Similarly, the leading terms of the elements of $\tilde{F}^+$ contribute to the ideal spanned by the leading terms of the intermediate basis. This is formalized in the following lemma.

Lemma 4.1.2. *[Seg04, p.59] Let $\tilde{F}^+$ denote the output of Reduction applied to L_d with respect to G. For all $f \in \tilde{F}^+$, $LM(f)$ is not an element of $\langle LM(G)\rangle$.*

Proof. [Seg04, p.59] Let F the set computed by the algorithm *Symbolic Preprocessing*(L_d, G). Assume for a contradiction that $\exists h \in \tilde{F}^+$ such that $t = LM(h) \in \langle LM(G)\rangle$. Hence $LM(g)$ divides t for some $g \in G$. So t is in $M(\tilde{F}^+) \subset M(\tilde{F}) \subset M(F)$ and is top reducible by g, hence $\frac{t}{LM(g)}g$ is inserted in F by *Symbolic Preprocessing* (or another product with the same head monomial). This contradicts the fact that $LM(h) \notin LM(F)$. □

The next lemma assures that the elements that are added to the intermediate basis, are members of the ideal $\langle G\rangle$.

Lemma 4.1.3. *[Seg04, p.59] Let $\tilde{F}^+$ be as in Lemma 4.1.2. Then $\tilde{F}^+ \subset \langle G\rangle$.*

Proof. [Seg04, p.60] Every $f \in \tilde{F}^+$ is a linear combination of elements of L_d and reductors R, which are both subsets of $\langle G\rangle$. □

The following lemma states that all S-polynomials in the set of possible k-linear combinations of L_d reduce to zero by a subset of $\tilde{F}^+ \cup G$. This is used to prove the correctness of the algorithm by the criterion stated in Theorem 2.3.7 on page 17.

Lemma 4.1.4. *[Seg04, p.60] Let $\tilde{F}^+$ be as in Lemma 4.1.2. For all k-linear combinations f of elements of L_d, $f \underset{\tilde{F}^+\cup G}{\longrightarrow} 0$.*

Proof. [Seg04, p.60] Let f be a linear combination of elements of L_d. Suppose F is the output of the *Symbolic Preprocessing* of L_d and G. By construction, L_d is a subset of F and, therefore due to Theorem 4.1.1 on page 41, these elements are a linear combination of the triangular basis $\tilde{F}^+ \cup H$ for a suitable subset $H \subset F$. Elements of H are either elements of L_d or (by construction in *Symbolic Preprocessing*) of the form $x^\alpha g$, for $g \in G$ and $\alpha \in \mathbb{Z}^n$, and f can thus be written as

$$f = \sum_i a_i f_i + \sum_j a_j x^{\alpha_j} g_j,$$

for $f_i \in \tilde{F}^+$ and $g_j \in G$, $a_i, a_j \in k$ and $\alpha_j \in \mathbb{Z}_n$. Thus the division algorithm gives a remainder equal to 0 for a suitable tuple of elements in $\tilde{F}^+ \cup G$, hence there exists a reduction chain to 0. □

Using these results everything is in place to formulate a first version of F_4 and prove its correctness.

Algorithm 5 ($F4_o$)**.**

```
def f4(F,Sel):
  """
  INPUT:
      F -- a finite subset of R
      Sel -- a selection strategy, e.g. the normal strategy
  OUTPUT:
      a Groebner basis for the ideal spanned by F
  """
  G = F
  Ftildeplus[0] = F
  d = 0
  P = set([Pair(f,g) for f, g in G with f != g])
  while P != set():
    d += 1
    Pd = Sel( P )
    P = P.difference( Pd )
    Ld = Left( Pd ).union( Right(Pd) )
    Ftildeplus[d] = reduction(Ld , G)
    for h in Ftildeplus[d]:
      P = P.union( set([ Pair(h, g) for g in G ]) )
      G = G.add( h )
  return G
```

Theorem 4.1.5. *Algorithm 5 computes a Gröbner basis G for an ideal spanned by F, such that $F \subseteq G$, in a finite number of steps.*

Proof. [Fau99, p.8] Termination and correctness need to be proven:

Termination Assume for a contradiction that the while-loop does not terminate. There exists an ascending sequence (d_i) of natural numbers such that $\tilde{F}^+_{d_i} \neq \emptyset$ for all i. Say that $q_i \in \tilde{F}^+_{d_i}$ (hence q_i can be any element in $\tilde{F}^+_{d_i}$). Let $\bigcup_i$ be $\bigcup_{i-1} + \langle LM(q_i) \rangle$ for $i > 1$ and $\bigcup_0 = \{0\}$. From Lemma 4.1.2 on the previous page ($LM(h) \notin LM(G)$) follows $U_{i-1} \subsetneq U_i$ as the elements of $\tilde{F}^+_{d_i}$ are added to G at the end of every loop. This infinite chain of ideals contradicts the fact that R is noetherian.

Correctness G is $\bigcup_{d \geq 0} \tilde{F}^+_d$. The claim is that the following statement are loop invariants of the while-loop: G is a finite subset of $k[x_0, \ldots, x_{n-1}]$ such that $F \subset G \subset \langle F \rangle$, and the S-polynomials for all $g_0, g_1 \in G$ reduce to zero with respect to G such that $\{g_0, g_1\} \notin P$, the set of critical pairs. The first claim is an immediate consequence of Lemma 4.1.3 on the preceding page. For the second one, if $\{g_0, g_1\} \notin P$, this means that $Pair(g_0, g_1)$ has been selected in a previous step (say d) by the function *Sel*. Hence $Left(Pair(g_0, g_1))$ and $Right(Pair(g_0, g_1)$ are in L_d, so the S-polynomial of g_0, g_1 is an element of the R-module generated by L_d hence by Lemma 4.1.4 on the previous page it reduces to zero with respect to G.

□

4.1.2 The Improved F_4

In [Fau99] Faugère also presents an improved version of his algorithm "in order to obtain an efficient algorithm" [Fau99]. This algorithm has the Buchberger Criteria "inserted". Faugére suggests to use the Gebauer and Möller installation [GM88] as F_4 is not concerned with improving the Buchberger Criteria (this is dealt with in [Fau02]).

Algorithm 6 (F_4).

```
def f4(F, Sel, Update):
  """
  INPUT:
      F -- a finite subset of R
      Sel -- a selection strategy, e.g., the normal strategy
      Update -- selects the pairs to compute, as in Buchberger's algorithm

  OUTPUT:
      a Groebner basis for the ideal spanned by F
  """
  G = set()
  P = set()
  d = 0
  Fd = list()
  while F != set():
    f = first(F)
    F.remove(f)
    G,P = Update(G,P,f)

  while P != set():
    d = d+1
    Pd = Sel(P)
    P = P.difference(Pd)
    Ld = Left(Pd).union( Right(Pd) )
    Fdp,Fd[d] = reduction(Ld,G,Fd)
    for h in Fdp:
      G,P = Update(G,P,h)
  return G
```

In this algorithm the subroutine *First* simply picks the largest polynomial w.r.t. to the term order. The routines *Reduction* and *Symbolic Preprocessing* are adapted as follows:

Algorithm 7 (Symbolic Preprocessing).

```
def symbolic_preprocessing(L,G):
  """
  INPUT:
      L -- a finite subset of M x R
      G -- a finite subset of R
      Fset -- (F_k)k=1...(d-1), where F_k is a finite subset of R

  OUTPUT:
      a finite subset of R
  """
  F = set([ mul(simplify(m,f,Fset)) for (t,f) in L])
  Done = LM(F)
  while LM(F) != Done:
    m = (T(F).difference(Done)).pop()
    Done.add(m)
    if m "is divisible by an element" g in LM(G):
      m2 = m/LM(g)
      F.add(mult(simplify(m2,f,F)))
  return F
```

Algorithm 8 (Reduction).

```
def reduction(L,G):
  """
  INPUT:
      L -- a finite subset of M x R
      G -- a finite subset of R
      Fset -- (F_k)k=1...(d-1), where F_k is a finite subset of R

  OUTPUT:
      (a finite subset of R, a finite subset of R)
  """
  F = symbolic_preprocessing(L,G,Fset)
  Ftilde = "Reduction to Row Echelon Form of F w.r.t. <"
  Ftildeplus = set([f for f in Ftilde if LM(f) not in LM(F)])
  return (Ftildeplus,F)
```

Algorithm 9 (Simplify)**.**

```
def simplify(t,f,F):
  """
  INPUT:
       t -- \in M a monomial
       f -- \in R[x] a polynomial
       F -- (F_k)k=1,\dots,(d-1), where F_k is finite subset of R[x]

  OUTPUT:
       a non evaluated product, i.e. an element of T x R[x]
  """
  for u in "all divisors of " t:
    if "exists j(1<=j<d) such that (u*f) in F_j":
       """
       F~_j is the row echelon form of F_j w.r.t. <
       there exists a (unique) p \in F~_j such that LM(p) = LM(u*f)
       """
       if u!=t:
         return simplify(t/u,p,F)
       else:
         return (1,p)
  return (t,f)
```

As the *Simplify* subroutine is the most visible change to the algorithm the main theorem about the *Simplify* algorithm is stated and proven below.

Lemma 4.1.6. *[Fau99, p.10] If (t', f') is the result of* Simplify$(t, f, \mathcal{F})$*, then $LM(t' \cdot f') = LM(t \cdot f)$. Moreover if $\tilde{\mathcal{F}}^+$ denotes $(\tilde{F}_k^+)_{k=1,\dots,d-1}$, then there exists $0 \neq \lambda \in R$, and $r \in R-$ module$(\tilde{\mathcal{F}}^+ \cup F)$ such that $tf = \lambda \cdot t' \cdot f' + r$ with $LM(r) < LM(t \cdot f)$.*

Proof. [Fau99, p.10] Again termination and correctness need to be proven:

Termination Simplify constructs a sequence (t_k, f_k) such that $t_0 = t$, $f_0 = f$ and $t_{k+1} < t_k$ except perhaps for the last step. M, the set of monomials in R, is noetherian, this implies that the algorithm stops after r_k steps. In the last step where $t_{k+1} = t_k$ may occur, the algorithm terminates anyway.

Correctness The first part is true since $LM(u_k f_k) = LM(f_{k+1})$ so that

$$LM(t_k f_k) = LM(\frac{t_k}{u_k} f_{k+1}) = LM(t_{k+1} f_{k+1})$$

.

The proof is by induction on the step number. Suppose $r_k = 1$, $t' = \frac{t}{u}$ and $uf \in F_j, f' \in \tilde{F}_j$ for some j with $LM(f') = LM(uf)$.

The set $F_- = \{uf\}$ can be supplemented by other elements of F_j such that $LM(F_-) = LM(F)$ and $|F_-| = |LM(F)|$. Apply Theorem 4.1.1 on page 41 and find $(\alpha_k) \in R$, $g_k \in F_- \cup (\tilde{F}_j)_+$, such $f' = \sum_k \alpha_k g_k$ and $LM(g_1) = LM(f')$ and $LM(f') > HT(g_k)$ for $k > 2$. By construction of $F_-, g_1 = uf$. Hence $f' = \alpha_1 uf + r$ with $LM(r) < LM(f')$, consequently $\alpha_1 \neq 0$ and we have $tf = \frac{1}{\alpha} t' f' - \frac{1}{\alpha} t' r$.

□

The interested reader is referred to [Fau99] for the proof that the improved F_4 algorithm actually computes a Gröbner basis for a given set of generators in a finite number steps.

4.1.3 A Toy Example for F_4

As an example consider the ideal $\langle 29 + x_1^2 + 107 x_0 x_1, 114 + 80 x_0 x_1 + x_0^2 \rangle \subset \mathbb{F}_{127}[x_0, x_1]$ with respect to a *lex* ordering. When the main loop is entered: $P = P_1 = [(x_0^2 x_1, 29 + x_1^2 + 107 x_0 x_1, 114 +$

$80x_0x_1 + x_0^2)]$ and $G = [29 + x_1^2 + 107x_0x_1, 114 + 80x_0x_1 + x_0^2]$. Consequently $L_1 = [(x_1, 114 + 80 * x_0 * x_1 + x_0^2), (x_0, 29 + x_1^2 + 107 * x_0 * x_1)]$.

Symbolic Preprocessing returns $[29x_1 + x_1^3 + 107x_0x_1^2, 114x_1 + 80x_0x_1^2 + x_0^2x_1, 29x_0 + x_0x_1^2 + 107x_0^2x_1]$ or in matrix form:

$$F = A_F \cdot v_F = \begin{pmatrix} 0 & 107 & 0 & 1 & 29 \\ 1 & 80 & 0 & 0 & 114 \\ 107 & 1 & 29 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} x_0^2x_1 \\ x_0x_1^2 \\ x_0 \\ x_1^3 \\ x1 \end{pmatrix}.$$

The row echelon form of F is

$$\tilde{F} = \tilde{A}_F \cdot v_F = \begin{pmatrix} 1 & 0 & 0 & 4 & 103 \\ 0 & 1 & 0 & 19 & 43 \\ 0 & 0 & 1 & 24 & 17 \end{pmatrix} \cdot \begin{pmatrix} x_0^2x_1 \\ x_0x_1^2 \\ x_0 \\ x_1^3 \\ x1 \end{pmatrix}.$$

or as a set of polynomials $\tilde{F} = [17x_1 + 24x_1^3 + x_0,\ 43x_1 + 19x_1^3 + x_0x_1^2,\ 103x_1 + 4x_1^3 + x_0^2x_1]$. Those polynomial whose leading monomials are not in F are $\tilde{F}^+ = [17x_1 + 24x_1^3 + x_0]$.

During the next iteration:

$$\begin{aligned}
P = P_2 = {} & [(x_0x_1, 17x_1 + 24x_1^3 + x_0, 29 + x_1^2 + 107x_0x_1), \\
& (x_0^2, 17x_1 + 24x_1^3 + x_0, 114 + 80x_0x_1 + x_0^2)], \\
G = {} & [17x_1 + 24x_1^3 + x_0], \\
L_2 = {} & [(1, 29 + x_1^2 + 107x_0x_1), (1, 114 + 80x_0x_1 + x_0^2), \\
& (x_1, 17x_1 + 24x_1^3 + x_0), (x_0, 17x_1 + 24x_1^3 + x_0)], \\
F = {} & [17x_1^2 + 24x_1^4 + x_0x_1, 29 + x_1^2 + 107x_0x_1, \\
& 17x_1^4 + 24x_1^6 + x_0x_1^3, 114 + 80x_0x_1 + x_0^2, 17x_0x_1 + 24x_0x_1^3 + x_0^2], \\
\tilde{F} = {} & [67 + 74x_1^2 + x_1^4, 122 + 52x_1^2 + x_1^6, 43 + 19x_1^2 + x_0x_1, \\
& 124 + 34x_1^2 + x_0x_1^3, 103 + 4x_1^2 + x_0^2], \\
\tilde{F}^+ = {} & [67 + 74x_1^2 + x_1^4, 122 + 52x_1^2 + x_1^6].
\end{aligned}$$

The third is the last iteration and the involved sets are as follows:

$$\begin{aligned}
P = P_3 = {} & [(x_1^6, 67 + 74x_1^2 + x_1^4, 122 + 52x_1^2 + x_1^6)], \\
G = {} & [67 + 74x_1^2 + x_1^4, 17x_1 + 24x_1^3 + x_0], \\
L_3 = {} & [(1, 122 + 52x_1^2 + x_1^6), (x_1^2, 67 + 74x_1^2 + x_1^4)], \\
F = {} & [67 + 74x_1^2 + x_1^4, 122 + 52x_1^2 + x_1^6, 67x_1^2 + 74x_1^4 + x_1^6], \\
\tilde{F} = {} & [67 + 74x_1^2 + x_1^4, 122 + 52x_1^2 + x_1^6], \\
\tilde{F}^+ = {} & \emptyset.
\end{aligned}$$

As no critical pairs are left to choose the algorithm terminates and returns the Gröbner basis

$$G = [17x_1 + 24x_1^3 + x_0, 67 + 74x_1^2 + x_1^4].$$

This example was produced using the F_4 implementation provided with this thesis and the `protocol=True` option.

4.1.4 Complexity of F_4

In [Fau99] Faugére states that the complexity of his algorithm is $\mathcal{O}\left(d^{3n}\right)$ for normal cases and $\mathcal{O}\left(2^{2^n}\right)$ for some "pathological cases" where d is the degree of the polynomials in the initial polynomial set and n is the number of invariants. In the case of CTC ideal bases $d = 2$ and $n = 4 \cdot Bs \cdot N_r + Bs$ as $K_{i,j}, X_{i,j}, Y_{i,j}, Z_{i,j}$ are added per round and one additional key addition with $K_{0,j}$ is performed. Consequently, F_4 is supposed to have a complexity of $\mathcal{O}\left(2^{4 \cdot Bs \cdot N_r + Bs}\right)$ operations which is worse than exhaustive key search which has a complexity of $\mathcal{O}\left(2^{Bs}\right)$ operations.

For more fine graded performance data in comparison to other Gröbner basis algorithms, benchmarks are provided in [Fau99] which suggest F_4 "is at least one order of magnitude faster than all previously implemented algorithms" [Fau99]. However, this claim needs to be backed up by a very efficient open-source implementation which does not exist at this point for the general case.

4.1.5 Implementations of F_4

Probably the most well-know F_4 implementation was provided by Faugère as a binary-only, closed-source implementation on his website [Fau06] for use with the computer algebra system Maple [MGH+05]. Also, this implementation is provided for academic use only. The most widely used F_4 implementation in cryptography (e.g., [CMR05], [BPW05]) is provided with the computer algebra system MAGMA [BCP97] and is not freely available or open source. MAGMA's implementation is especially optimized for ideals in $\mathbb{F}_2[x_0, \ldots, x_{n-1}]$. Toon Segers also provided an F_4 implementation using MAGMA in his Master's thesis [Seg04].

The open-source computer algebra system Singular [GPS05] provides a command `slimgb` to compute a Gröbner basis using the *SlimGB* algorithm. This algorithm is sometimes, e.g., in [TF05], understood to be F_4, and thus Singular is believed to provide a free and open source implementation of F_4. However, this is not true, as the following quote by the author of both the algorithm and the command in Singular shows:

> "I have implemented two algorithms for computing Groebner basis in Singular:
>
> F_4 (which is slow like a dog without some decent linear algebra and deactivated for this reason)
>
> slimgb (which is my algorithm) Slimgb borrows some ideas from F_4, generalizes them, does some new things, but is at the moment the opposite of F_4. It is in no way specialized on the easy case (dp, homogeneous, field is Z/p), not optimized on char 0 at the moment (which is something I work on). It is very good on function fields (rings with parameters) and it is also good in elimination orderings (in very recent versions, the sources file Singular-3-0-0-4.tar.gz for example). In opposite to F_4 it works on modules. I even have some fast noncommutative version in CVS." [Bri05a]

For more information on the *SlimGB* algorithm the interested reader is referred to [Bri05b].

Both Singular and Macaulay2 [GS] contain code which provides an F_4 implementation but the implementation is either disabled because it is too slow or it is not yet stable.

The only fully functional and fast open-source implementation of F_4 I am aware of is called "hotaru" or IPA-SMW [Shi] and is provided under a BSD-style license by Mitsunari Shigeo. It is limited to the quotient ring

$$\mathbb{F}_2[x_0, \ldots, x_{127}]/\langle x_0^2 + x_0, \ldots, x_{127}^2 + x_{127} \rangle$$

and seems to be hardly recognized outside Japan. But the author claims that it is faster than MAGMA in this ring and it was used to break Toyocrypt.

command	time in quotient ring	time without quotient ring
std	0.35 s	165.52 s
stdhilb	74.86 s	105.54 s
groebner (heuristic)	4.91 s	41.32 s
stdfglm	not applicable	0.68 s
slimgb	not applicable	503.75 s

Figure 4.1: Time it takes Singular's algorithms to compute a Gröbner basis for $CTC_{3,3,2}$.

Finally this thesis provides an open-source implementation of F_4 over finite fields with order p=prime. This implementation is written in pure Python. It provides very readable source code and allows the user to provide his/her own *Update* and *Sel* functions. Also a specialized version for the quotient ring described in Section 3.3 is available which is influenced by [Shi] but is more flexible as it is not limited to 128 variables.

An example usage of this F_4 implementation is listed below:

```
sage: attach "f4.py"
sage: MixInSAGE()
sage: F = f4.example_Faugere() #(Cyclic 4)
sage: gb = f4.groebner(F)
sage: Ideal(gb).is_groebner()
True

# ideals in P/FieldIdeal are also supported
sage: attach "polyf2.spyx"
sage: R.<a,b,c,d> = MPolynomialRingGF2(4)
sage: F = f4.example_Faugere(R)
sage: gb = f4.groebner(F, Update=f4.update_pairsGF2)
sage: Ideal(gb).is_groebner()
True
```

4.1.6 Benchmarks

The F_4 implementation provided with this thesis only supports Gröbner basis calculations over $\mathbb{F}_p$ where p is prime. This implementation is also generally slow. However, if calculations are performed in the quotient ring as described in Section 3.3, it is faster than Singular's fastest algorithm for some CTC instances. This is mainly because several optimization ideas were taken from [Shi].

Before benchmarking this F_4 implementation against Singular, Singular's fastest Gröbner basis algorithm for CTC ideals (in the quotient ring) has to be found. Table 4.1.6 lists the time Singular needs to calculate a *lex* Gröbner basis for $CTC_{3,3,2}$. It shows that Singular's heuristic which determines the algorithm to use does not choose the fastest algorithm for this problem.

Consequently, consider some timing examples, e.g. the time it takes Singular's `std` and this F_4 implementation to compute a *lex* Gröbner basis for $CTC_{3,3,3}$. Singular's `std` command – which is a heavily optimized Buchberger implementation - takes 117 s while this F_4 implementation takes around 25 s. Computing a $CTC_{3,4,3}$ *lex* Gröbner basis takes about 1,100 s using F_4 while the calculation using Singular had to be interrupted after 10,800 s.

Please note, that more careful benchmarks are needed to estimate the actual performances of both implementations. However, these superficial benchmarks already support the claim that F_4 is a powerful algorithm.

4.2 Using Resultants: DR

Using Dixon Resultants as a technique to solve a $\mathcal{MQ}$ problem was recently introduced by Tang and Feng [TF05]. This technique does not depend on any special structure of the underlying $\mathcal{MQ}$ problem to solve it but works for any instance of a $\mathcal{MQ}$ problem over a finite field where m – the number of equations – equals n – the number of variables. In this section, the concept of Dixon resultants will be presented, so as the extended Dixon resultants which were introduced by [KSY94]. Finally the DR algorithm as proposed by [TF05] is described. A lot of the following is based on the Master's thesis of Adam Thomas Feldmann [Fel05] which describes Dixon resultants and the DR algorithm not only in more detail but also corrects some minor errors from the original paper by Tang and Feng [TF05].

4.2.1 Dixon Polynomial, Dixon Matrix and Dixon Resultant

First, some notation needs to be established: A *generic ndegree* polynomial is a multivariate polynomial f which can be written as

$$f(x_0, x_1, \ldots, x_{n-1}) = \sum_{i_0=1}^{k_0} \sum_{i_1=1}^{k_1} \cdots \sum_{i_{n-1}=1}^{k_{n-1}} a_{i_0,i_1,\ldots i_{n-1}} x_0^{i_0} x_1^{i_1} \ldots x_{n_1}^{i_{n-1}}$$

for some positive integers $k_0, k_1, \ldots, k_n$. The feature of such polynomials is that every monomials contains every variable as all $i_k \geq 1$.

Now let $F = \{f_0(x_0, x_1, \ldots, x_{n-1}), \ldots f_n(x_0, x_1, \ldots, x_{n-1})\}$ be a set of $n+1$ such generic ndegree polynomials and consider the following determinant $\Delta(x_0, x_1, \ldots, x_{n-1}, \alpha_0, \alpha_1, \ldots, \alpha_{n-1}) =$

$$det \begin{bmatrix} f_0(x_0, x_1, \ldots, x_{n-1}) & f_1(x_0, x_1, \ldots, x_{n-1}) & \ldots & f_n(x_0, x_1, \ldots, x_{n-1}) \\ f_0(\alpha_0, x_1, \ldots, x_{n-1}) & f_1(\alpha_0, x_1, \ldots, x_{n-1}) & \ldots & f_n(\alpha_0, x_1, \ldots, x_{n-1}) \\ f_0(\alpha_0, \alpha_1, \ldots, x_{n-1}) & f_1(\alpha_0, \alpha_1, \ldots, x_{n-1}) & \ldots & f_n(\alpha_0, \alpha_1, \ldots, x_{n-1}) \\ \vdots & \vdots & & \vdots \\ f_0(\alpha_0, \alpha_1, \ldots, \alpha_{n-1}) & f_1(\alpha_0, \alpha_1, \ldots, \alpha_{n-1}) & \ldots & f_n(\alpha_0, \alpha_1, \ldots, \alpha_{n-1}) \end{bmatrix}. \tag{4.1}$$

In this determinant $\alpha_0, \alpha_1, \ldots, \alpha_{n-1}$ are new variables. The *Dixon polynomial* δ of F is then given by:

$$\delta(x_0, x_1, \ldots, x_{n-1}, \alpha_0, \alpha_1, \ldots, \alpha_{n-1}) = \frac{\Delta(x_0, x_1, \ldots, x_{n-1}, \alpha_0, \alpha_1, \ldots, \alpha_{n-1})}{(x_0 - \alpha_0)(x_1 - \alpha_1)\ldots(x_{n-1} - \alpha_{n-1})}.$$

Please note, δ is still a polynomial: if each x_i in Δ is replaced by α_i two identical rows are produced in the matrix 4.1. Thus the determinant of the matrix becomes zero, and division by $(x_i - \alpha_i)$ corresponds to removing a root of Δ.

If the matrix 4.1 is evaluated at a common zero of F the first row will become identical zero. Thus, the determinant is also zero. Therefore, the polynomial δ in $x_0, x_1, \ldots, x_{n-1}, \alpha_0, \alpha_1, \ldots, \alpha_{n-1}$ vanishes if evaluated at a common zero of F regardless what the values of the α_i are. Now, consider δ to be a polynomial in $\alpha_0, \alpha_1, \ldots, \alpha_{n-1}$ whose coefficients are polynomials in $x_0, x_1, \ldots, x_{n-1}$. By fixing an ordering for these polynomials, calling the resulting vector of these polynomials ϵ, and viewing each power product of $x_0, x_1, \ldots, x_{n-1}$ as a new variable v_i $(i = 0, \ldots, s-1)$the coefficient matrix D can be constructed satisfying the following equality:

$$\epsilon = D \cdot (v_0, v_1, \ldots, v_{s-1})^T = (0, 0, \ldots, 0)^T$$

The matrix D is called *Dixon Matrix*, $(v_0, v_1, \ldots, v_{s-1})^T$ is called V, and $det(D)$ is called the *Dixon Resultant*.

If the input system consists exclusively of generic ndegree polynomials [KSY94], note that $|\epsilon| = |V|$ and the determinant of the square coefficient matrix can be calculated. The interesting property of the Dixon resultant is that it vanishes if there exists a common zero for F as the Dixon polynomial is zero then. If this calculation is performed over a polynomial ring with parameter coefficients instead of $\mathbb{F}$, the Dixon resultant provides information on the necessary conditions for this parameters so that F has a common zero, because $det(D)$ is a polynomial in these parameters not identically zero.

However, when attacking block ciphers the polynomials are most likely not of type generic ndegree. In this case the Dixon matrix is often singular, yielding no information for the polynomials. But Kapur, Saxena, and Yang [KSY94] extended the Dixon resultant to work with this general case. This extension only works if a condition called "Rank Submatrix Construction Criteria" (RSC) holds. Both [KSY94] and [TF05] state that this condition always held in their experiments while neither provides an argument why this is the case.

4.2.2 The KSY Dixon Matrix and the Extended Dixon Resultant

The technique introduced by [KSY94] constructs the Dixon matrix D as in the previous paragraph. If this matrix has rank r a $r \times r$ submatrix of D has to be found that is also of rank r. This new matrix is called the *KSY Dixon Matrix* and its determinant is called the *Extended Dixon Resultant*.

From now on, the calculations are also performed in a polynomial ring with parameter coefficients, i.e. the coefficient ring is $\mathbb{F}[a_0, a_1, \ldots, a_{l-1}]$ for some $l > 0$.

Let G be a $\mathcal{MQ}$ problem with parameter coefficients. Let D be the $s_1 \times s_2$ Dixon matrix of G where s_1 may differ from s_2. The columns of D are represented by $m_0, m_1, \ldots, m_{s-1}$. Let $monom(m_i) = v_i$ and let C be a set of constrains on the variables $x_0, x_1, \ldots, x_{n-1}$ of the form

$$x_{i_0} \neq 0 \wedge x_{i_1} \neq 0 \wedge \cdots \wedge x_{i_{k-1}} \neq 0,$$

for some $0 \leq i_0, i_1, \ldots, i_{k-1} < n$. Let $nvcol(C)$ denote the set of all columns m_i, such that $C \rightarrow monom(m_i) \neq 0$. Let $N_1 = \{X | X$ is an $s_1 \times (s_2 - 1)$ submatrix of the Dixon matrix D obtained by deleting a column which belongs to $nvcol(C)$ from $N\}$. Let $\phi : a_0, \ldots a_{m-1} \rightarrow \overline{\mathbb{F}}$ be a mapping which assigns values to the parameters from the algebraic closure of the base field $\mathbb{F}$. $\phi(N_1), \phi(D), \phi(G)$ are the result of these mappings applied to N_1, D, and G respectively. Finally, let $R = \{Y | Y$ is an $r \times r$ nonsingular submatrix of $D\}$. According to [KSY94] it holds that:

Theorem 4.2.1. *If $\exists X \in N_1$ such that $rank(X) < rank(D)$ then for all $Y \in R$, $\phi(det(Y))$ vanishes if $\phi(F)$ has a common affine zero which satisfies C.*

Proof. See [KSY94].

One can now obtain an algorithm as follows: Check the *RSC Criteria* i.e. if $\exists X \in N_1$ such that $rank(X) < rank(D)$. If this is true, any element of R is called *KSY Dixon Matrix* and the determinant of that element of R is called the *Extended Dixon Resultant*. This provides the required polynomial to determine the conditions on the coefficient parameters for G to have a common zero.

4.2.3 The DR Algorithm

This leads to the follow algorithm:

Algorithm 10 (DR Algorithm)**.**

Input A system A of m multivariate quadratic equations with n variables over a finite field $\mathbb{F}$, and $m = n$.

Output At least one common solution of the input system.

Step 1 Taking $x_0 \dots x_{n-2}$ as variables and x_{n-1} as parameter, compute the Dixon matrix of A.

Step 2 Choose an appropriate value for *maxtrials*. Run the subprogram RSC to check the RSC Criteria. If RSC returns failure, substitute the partial solution $x_{n-1} = p$ into A and rerun DR with $n-1$ variables and $m-1$ equations. If RSC did not return failure, select rows and columns that are needed to construct the *KSY Dixon Matrix*.

Step 3 Construct the *KSY Dixon Matrix*.

Step 4 Compute the determinant of the *KSY Dixon Matrix*.

Step 5 Solve the equation from *Step 4* over $\mathbb{F}$ (e.g., with Berlekamp's algorithm). There may be several roots, the set of these roots is called s.

Step 6 For each root for x_{n-1}, substitute it into the *KSY Dixon Matrix* from step 3, then solve the linear equation to find the values of all the other monomials.

Step 7 Use these monomials v_i to find the values for the variables $x_0, x_1, \dots, x_{n-1}$.

Step 8 If *Step 6 &7* failed to find a common solution for A, let $s = \{0, p\}$ (p as used in subprogram RSC), and run *Step 6 & 7*.

The subprogram *RSC* is defined as:

Algorithm 11 (RSC Criteria)**.**

Input A Dixon matrix M of dimension $s_1 \times s_2$ and a threshold *maxtrials*.

Output The rows and columns that are needed to construct the KSY Dixon matrix.

Step 0 Set $i = 0$

Step 1 $i = i + 1$; Substitute a random value $p \in \mathbb{F}$ for x_{n-1} in the Dixon matrix M.

Step 2 Bring the matrix from *Step 1* to row echelon form, assume the result is M' and the rank of M' is r.

Step 3 If M' is a square and full rank matrix then return all the rows and columns in M.

Step 4 For each column m of matrix M' construct a submatrix M_s of M' of dimension $s_1 \times (s_2 - 1)$ by deleting m; if $rank(M_s) < r$ then break this loop;

Step 5 If *Step 4* found a submatrix M_s, whose rank is less than r then choose the columns needed to construct a $r \times r$ submatrix of M whose rank is r; transpose M and bring it to row echelon form, then choose the rows needed to construct a $r \times r$ submatrix of M and whose rank is r. Return the rows and columns. Else if $i < maxtrials$ goto step 1 else return failure and p.

Please note that the algorithms *DR* and *RSC* presented here are slight modifications of the algorithms presented in [TF05]. The versions in the original paper do not terminate for all equation systems while the versions presented here do. For details see Proof 4.2.3 on the facing page.

The run-time of the algorithm may be reduced over a small field $\mathbb{F}$ by replacing the steps 4,5 in the DR algorithm by this step:

step 4&5 : for each value p of $\mathbb{F}$ substitute p for x_{n-1} in the KSY Dixon matrix and call the result M'. If $det(M') = 0$ go on to step 6.

Testing all possible values is supposed to be faster as numeric calculations are much faster than symbolic calculations if there are few values to check. As CTC ideals are defined over $\mathbb{F}_2$ this approach is definitely faster.

Theorem 4.2.2. *Given a finite set of polynomials A in $\mathbb{F}[x_0, \ldots, x_{n-1}]$ DR returns at least one common solution for A in a finite number of steps.*

Proof. Correctness and termination need to be proven.

Correctness The correctness of this algorithm can only be proven if it is assumed that the *RSC* criteria always holds. Both [KSY94] and [TF05] state that they have not met the case in their experiments when the *RSC* criteria didn't hold. However, as stated earlier no explanation for this situation is provided in both papers. Thus, there is no complete proof the of the correctness of the DR algorithm. Most of the remaining proof follows [Fel05].

Step 1 of DR merely computes a Dixon matrix.

The random element p chosen in *Step 1* of *RSC* is equivalent to the function ϕ of Theorem 4.2.1 on page 51. *Steps 2* through *5* of RSC check the RSC Criteria of the same theorem using an implicitly chosen set of constraints $C = \{x_0 \neq 0, \ldots, x_{n-1} \neq 0\}$. It is assumed that the *RSC* Criteria will hold. Note that the equality of *Step 3* occurs only if the original Dixon matrix M was a square matrix of full rank r. If the loop of *Step 4* completes without being broken to go to *Step 5*, then the choice of $x_{n-1} = p$ was bad. The choice of p can only be bad if $x_{n-1} = p$ is part of a solution $x_0, x_1, \ldots, x_{n-2}$, p to the system G. The solution is to choose a different p and repeat the process as long as a given threshold is not reached. If it is reached the solution p is accepted and DR is rerun with $n-1$ variables and $m-1$ equations. This threshold is necessary for the algorithm to terminate in some cases.

Step 3 of *DR* computes a matrix as in Theorem 4.2.1 on page 51, which is the KSY Dixon matrix. *Step 4* computes the KSY Dixon resultant. *Step 5* is used to determine for which values of x_{n-1} the KSY Dixon resultant is 0. Then it is known that the function ϕ must evaluate x_{n-1} to one of the values found in *Step 5* in order for Theorem 4.2.1 on page 51 to hold. *Step 6* attempts to solve the linear system given by the KSY Dixon matrix for some of the indeterminates $v_0, v_1, \ldots, v_{s_2-1}$. Here, without loss of generality, the monomials are called $v_0, v_1, \ldots, v_{r-1}$. *Step 6*, similar to *Step 4* of RSC, can fail. It can fail if an affine zero exists only for $x_{n-1} = 0$, which the constraints C forbid, and therefore do not check. It can also fail if an affine zero exists for $x_{n-1} = p$. As the assumption is that an affine zero exists and that the RSC Criteria holds, if *Step 6* fails to find a necessary condition for an affine zero, then one of these last two values must provide such a necessary condition. The answer here is to retry *Step 6* using the last two possibilities for x_{n-1}, namely $0, p$, as indicated by *Step 8*.

Step 7 uses the indeterminates $v_0, v_1, \ldots v_{r-1}$ to determine the indeterminates $x_0.x_1, \ldots, x_{n-1}$ by recalling that each v_i is a monomial in $x_0, x_1, \ldots, x_{n-1}$. In fact, it is likely that some of the v_i are equivalent to monomials x_j, allowing to read of the indeterminates $x_0, x_1, \ldots, x_{n-1}$ from the indeterminates $v_0, v_1, \ldots .v_{r-1}$. *Step 7* simply checks all the potential common roots found.

Note that *Step 6* is inexact. It is possible that the column corresponding to the monomial $1 = x_0^0 x_1^0 \ldots x_{n-1}^0$ in the Dixon matrix M is a linear combination of the other columns, in which case it is not possible to fully define the solution set of A, as there will be no constant term to "start" a fully defined solution using the KSY Dixon matrix. It is also possible that the indeterminates $v_0, v_1, \ldots .v_{r-1}$ are not equivalent to monomials in such a way to compute the entirety of $x_0, x_1, \ldots, x_{n-2}$. In this case, the remaining terms can be chosen using a brute force search.

Termination There are two places in DR where an infinite loop might occur. The subroutine RSC loops as long as it only finds values p which are part of the solution for x_{n-1}. Thus, if all values in $\mathbb{F}$ are solutions for x_{n-1} RSC would loop forever if the threshold variable wasn't introduced to prevent this behavior. If the threshold *maxtrials* is reached the algorithm recurses. This recursion eventually terminates as it is run with $n-1$ variables and $m-1$ equations and both m and n are finite. Thus, the algorithm terminates after a finite number of steps.

□

4.2.4 Complexity of DR

In [TF05] Tang and Feng give the complexity of DR as follows. They state that the complexity depends on the size of the Dixon matrix. For this matrix the row echelon form is computed at least twice (if $p \notin V(A)$) and several times for the KSY dixon matrix depending on the number of roots of the univariate polynomial. The Dixon matrix generated by DR is approximately a square matrix so the complexity is given by $\mathcal{O}(min(s_1, s_2)^\omega)$ where ω is 3 for Gaussian reduction (or below for improved algorithms).

If the $\mathcal{MQ}$ problem is full – i.e. every possible monomial occurs in the initial set of equations – they argue that the Dixon matrix is bounded by 4^n where n is the number of variables. So the complexity of DR is given by $\mathcal{O}\left(4^{3n}\right)$ operations. However, full systems never occur in practice as a simple Gaussian reduction on them transforms them to systems with less monomials.

If the $\mathcal{MQ}$ problem is sparse the runtime of DR depends on the *mixed volume* of the problem, in [TF05] Tang and Feng conclude: "The runtime of DR is

- Polynomial if $m = n$ and the system is sparse and the system's *mixed volume* is polynomial;
- $2^{\omega \times n}$ if $m = n$ and the system is sparse and system's *mixed volume* is exponential;
- $C^{\omega \times n}$ if $m = n$ and the system is general, $C \to 4$."

More details and benchmarks can be found in [TF05] and the next section.

4.2.5 Benchmarks

To show that DR benefits from sparse systems, equation systems of "type A" are introduced in [TF05]. Equation systems of type A and size n are those equation systems, where each of the n polynomial p_i is of the form $p_i = x_i + x_{(i \ \% \ n)+1} \cdot x_{((i+1) \ \% \ n)+1} + b_i$. The following table shows that for these equation systems DR is faster than Singular's Gröbner basis engine.

n	term ordering	DR time	Gröbner basis time	FGLM time
11	*lex*	4.19 s	5.65 s	-
12	*lex*	7.11 s	35.16 s	-
12	*degrevlex*	7.40 s	0.09 s	3.56 s
13	*degrevlex*	12.95 s	0.08 s	12.95 s
14	*degrevlex*	33.69 s	0.18 s	87.73 s

The crossover point for *lex* term ordering is at $n = 11$. Computing a *lex* Gröbner basis using a *degrevlex* Gröbner basis and converting it with FGLM is faster and the crossover point where DR beats Singular is $n = 13$ in these experiments.

These timings, however, don't seem to apply to CTC ideals. Using DR against $CTC_{3,3,1}$ takes 37.17 s while computing a Gröbner basis with *lex* term ordering takes 1.1 s. As the timing difference increases if B increases, DR seems to perform worse than Singular's Gröbner basis algorithm. Please note, that in order to use DR against CTC the S-box equations need to be reduced.

4.2.6 Attacking CTC Ideals with DR

The equation systems derived from the CTC block cipher are overdefined and the base ring they are defined over is very small: $\mathbb{F}_2$. As DR is designed to work with systems where the number of equations m equals the number of variables n, three S-box equations out of 14 per S-box need to be chosen. This arbitrary choice is

$$\begin{aligned} Y1 &= X1 * X2 + X3 + X2 + X1 + 1, \\ Y2 &= X1 * X3 + X2 + 1, \\ Y3 &= X2 * X3 + Y2 + Y1 + X2 + X1 + 1. \end{aligned}$$

in the provided implementation.

An example calculation using DR is given below:

```
sage: attach 'dr.py'
sage: attach 'ctc.py'
sage: ctc = CTC()
sage: F = ctc.MQ_factory(p=[1,1,0],k=[1,0,0])
sage: strip_sboxes(F)
 Multivariate polynomial equation system with 15 variables and 15 polynomials (gens).
sage: dr = DR()
sage: dr.attack(F)
 {K000002: 0, K001001: 0} # only partial solver implemented for Step 7
```

4.3 The XL Family of Algorithms

The XL algorithm was proposed by Nicolas Courtois, Alexander Klimov, Jacques Patarin, and Adi Shamir in [CKPS00]. It builds on the re-linearization algorithm by Kipnis and Shamir [KS99]. In this section, the idea behind XL will be presentend so as the XL algorithm itself. An overview of some improvements introduced later will also be given and a proof that XL is in fact a redundant version of F_4 (see Section 4.1). At the end of this section XL will be applied against several CTC instances.

4.3.1 The XL Algorithm

Given a multivariate polynomial equation system F in $P = k[x_0, \ldots, x_{n-1}]$ with exactly one solution. To solve this system using the *Linearization* technique every monomial $x_i x_j$ is considered to be a new variable y_{ij}. If the the number of monomials $M(F)$ is $\leq$ the number of equations the system F may be solved by solving the linear system in the variables y_{ij} and substituting back. In general the number of monomials will be larger than the number of equations and thus Linearization does not work.

The idea behind *Relinearization* is to add trivial equations to the set F of the form:

$$(x_a x_b)(x_c x_d) = (x_a x_d)(x_b x_c).$$

As four variables are involved in this equation, this Relinearization is call *Fourth Degree Relinearization*. In [CKPS00] the authors show that many of the equations added by Relinearization are linear dependend and thus that the algorithm is less efficient than initially hoped. Also in [CKPS00] a new algorithm called *eXtended Linearization (XL)* is introduced which is simpler and more powerful than Relinearization:

Algorithm 12 (XL)**.**

For a positive integer $D \geq 2$, execute the following steps:

Multiply Generate all the products $\prod_{j=1}^{r} x_{i_j} \cdot f_i$ for $r \leq D-2$.

Linearize Consider each monomial term in the x_i of degree $\leq D$ as a new indeterminate and create a system of linear equations. Perform Gaussian elimination on these linear equations, using a monomial ordering that eliminates all the terms containing one indeterminate (say, x_1) last.

Solve Assuming that step 2 yields at least one univariate equation in the powers of x_1, solve this equation over the finite field.

Repeat Simplify the equations and repeat the process to find the values of the other indeterminates.

The authors of [CKPS00] claim that XL finds a solution to the original system F given that the parameter D is large enough (which later turned out not to be true for all systems). The runtime of the algorithm is dominated by Gaussian elimination in step 2 of the system generated in step 1. The size of the system depends on the parameter D and the overall runtime of XL is exponential in D.

4.3.2 Choosing D

In [CKPS00] the parameter D is estimated by the following heuristic:

Proposition 4.3.1. *[Seg04, p.49] Let $F = \{f_0, \ldots, f_{m-1}\} \subset k[x_0, \ldots, k_{n-1}]$ be a set of m quadratic polynomials describing a problem in cryptography. Under the assumption that almost all of the equations of the for $x^{\alpha} f_i$ of degree $\leq D$ generated by XL are linearly independent, XL has estimated complexity:*

$$\mathcal{O}\left(\left(\frac{n^D}{D!}\right)^{\omega}\right) \text{ with } D = \mathcal{O}\left(\frac{n}{\sqrt{(}m}\right) \text{ and } \omega = 2.376.$$

Proof. See [Seg04, p.49].

In this proposition ω represents the exponent in $\mathcal{O}(n^{\omega})$: The complexity of the row reduction of a matrix. ω is 3 for naïve Gauss elimination and 2.376 for the Coppersmith-Winograd algorithm [CW87]. In [Bar06] Greogy Bard shows that Coppersmith-Winograd is not very efficient for boolean matrices due to a massive constant factor and presents the "Method of the four Russians" inversion which has a complexity of $\mathcal{O}\left(\frac{n^3}{log(n)}\right)$.

The crucial assumption in Proposition 4.3.1 is that the produced equations are linearly independent which turns out not to be true as experiments have shown later [Moh01] (also see [CMR06]). Due to the difficulties to determine a minimal working D a priori, several incremental version of XL have been developed [SKI04]:

Simple Begin with $D = 1$. Do XL described as in Algorithm 12 on the preceding page for F. If you cannot obtain the solution, set $D = D + 1$ and do XL again for F with the new D.

Iterative Begin with $D = 1$. Iterate 'Multiply' and 'Linearize' described as in Algorithm 12 for F by adding new equations obtained by 'Linearize' to F. If you cannot solve the resulting system, then return to the original F, set $D = D + 1$ and iterate the same procedure as for $D = 1$. Repeat until you obtain the solution.

Incremental Begin with $D = 1$. Do XL described as in Algorithm 12 for F. If you cannot obtain the solution, then set $D = D + 1$, replace F by the resulting system obtained by 'Linearize' in the previous XL and do XL again for the new F and D. Repeat until you obtain the solution.

Iterative and Incremental Begin with $D = 1$. Iterate 'Multiply' and 'Linearize' described as in Algorithm 12 for F by adding new equations obtained by 'Linearize' to F. If you cannot solve the resulting system F' then replace F by F' , set $D = D + 1$ and iterate the same procedure as for $D = 1$. Repeat until you obtain the solution.

4.3.3 Example

As an example consider the example used throughout this thesis. Let $p_1 = [114 + 80x_0x_1 + x_0^2$ and $p_2 = 29 + x_1^2 + 107x_0x_1$ in $\mathbb{F}_{127}[x_0, x_1$. XL is run on $[p_1, p_2]$with $D = 4$. Then the row reduced set of polynomials produced is:

$$
\begin{aligned}
&115 + 106x_1^2 + x_0^4,\\
&28 + 66x_1^2 + x_0^3x_1,\\
&109x_1 + 119x_1^3 + x_0^3,\\
&113 + 61x_1^2 + x_0^2x_1^2,\\
&103x_1 + 4x_1^3 + x_0^2x_1,\\
&103 + 4x_1^2 + x_0^2,\\
&124 + 34x_1^2 + x_0x_1^3,\\
&43x_1 + 19x_1^3 + x_0x_1^2,\\
&43 + 19x_1^2 + x_0x_1,\\
&17x_1 + 24x_1^3 + x_0,\\
&67 + 74x_1^2 + x_1^4
\end{aligned}
$$

The last polynomial is univariate allowing to extract the solution: $x_0 = 89$ and $x_1 = 91$. The same calculation using the provided implementation:

```
sage: attach 'xl.py'
sage: attach 'xl_pyx.spyx'
sage: xl = XL()
sage: F = xl.example_Courtois_et_al()
sage: xl.attack(F,D=4)
 {x1: 91, x0: 89}
```

4.3.4 Later improvements on XL

Due to its general nature it is easy to adapt XL to new scenarios to increase its performance. Thus several improved XL variants have been introduced in the last years. The following overview is based on [Din06].

The XL' Variant The computational procedure of XL' [CP03] is similar to that of XL. The main difference is that in the third step of the procedure it tries to use elimination in order to find r equations that involve only monomials in a set of r variables, say $x_0, \ldots, x_{r-1}$. In the normal XL algorithm $r = 1$. It then solves this system of r equations by brute-force for these r variables. Finally, it solves the remaining equations by substituting the values of these variables.

The FXL and XFL Variant The "F" here stands for "fix" [CKPS00]; that is, the values of a small number of variables are guessed at random. This is motivated by the fact that XL performs much better if the equation system is overdefined. After guessing values for each of these variables, XL is run and tested for a valid solution. In [Cou04], a new suggestion was proposed. In this suggestion, after the second step of the XL procedure as given above, the elimination procedure should be run as far as possible before guessing another variable. This was first called "improved FXL" then the name XFL was suggested.

The XLF Variant In [Cou04], Courtois proposed another variation. This variation tries to utilize the field relation

$$x_i^q = x_i,$$

where $q = 2^l$ is the size of a finite field for some $l \geq 1$ (the field is of characteristic two). By treating the terms

$$x_i^{2^1} = x_{i_1}, x_i^{2^2} = x_{i_2}, \ldots, x_i^{2^{l-1}} = x_{i_{l-1}}$$

as independent new variables, additional equations are derived by repeatedly squaring the original equations and by using the equivalence of identical monomials as extra equations, for example

$$x_i^2 = x_{i_1}.$$

This variant is called XLF, where here "F" stands for "field" or "Frobenius equations".

The XSL Variants XSL stands for "eXtended Sparse Linearization". This variation by [CP02a] is a linearization-based approach designed to solve over-defined systems of sparse quadratic equations. Instead of multiplying with all monomials up to a certain degree only "carefully selected monomials" are used. It is unclear how exactly those monomials have to be selected as there are several XSL algorithms: "There are different versions of the algorithm (two attacks are given in [CP02a], which are substantially different from the attack proposed in [CP02b]), and in all cases, the description given leaves some room for interpretation." [CL05]. Also in [CL05] it is shown that the expected behavior of the XSL algorithm against AES is much worse than expected in [CP02a] and [CP02b].

The T'-Method Also in [CP02a] introduces the "T'-method" as a final step to either XL or XSL to produce more equations without increasing the number of monomials that appear in these equations. This method is best suited for $\mathbb{F}_2$ as the identity $x_i^2 = x_i$ is used to reduce produced monomials.

The XL2 Variant The XL2 algorithm was first proposed in [CP03] and works over the field $\mathbb{F}_2$. The basic idea is that since we work in $\mathbb{F}_2$, we should then automatically add the field equations $x_i^2 = x_i$, which is essentially that we should work in the function ring and not the polynomial ring. This idea was reformulated by [YCC04], and this method can be viewed as a way to more efficiently manage the elimination process.

4.3.5 XL is a Redundant F_4 Variant

In [SKI04] Makto Sugita, Mitsuru Kawazoe, and Hideki Imai show that XL may be viewed as a redundant version of F_4. To establish the relationship between XL and F_4 some pre-assumptions have to be made about XL. Consider attacking a multivariate polynomial system F over $k = \mathbb{F}_q$.

First, assume that the field equations of the finite field $k = \mathbb{F}_q$ (i.e., $x_i^q - x_i$) are implicitly or explicitly contained in the equation system F as we are not interested in any solutions from the algebraic closure. Secondly, assume that the equation system F has exactly one solution. This assumption is actually implicitly given in the original XL paper: "In this paper we are interested in the problem of solving overdefined systems of multivariate polynomial equations in which the number of equations m exceeds the number of variables n. Random systems of equations of this type are not expected to have any solutions, and if we choose them in such a way that one solution is known to exist, we do not expect other interference solutions to occur." [CKPS00] In that case the reduced Gröbner basis of F is exactly:

$$\{x_0 - a_0, \ldots, x_{n-1} - a_{n-1}\}$$

which follows directly from the uniqueness of reduced Gröbner bases.

To give an F_4-like description of XL define:

Definition 4.3.1. *If $R = k[x_0, \ldots, x_{n-1}]$, $F \subset R$, $p = (f, g) \in P$, M the set of monomials in R, and $d \in \mathbb{N}$, then define $XLLeft$ and $XLRight$ as follows:*

$$XLLeft(p, d) = XLRight(p, d) = \{(t, f) | t \in M, det(t * g) \leq d\}$$
$$XLLeft(P, d) = XLRight(P, d) = \bigcup_{p \in P} XLLeft(p, d) = \bigcup_{p \in P} XLRight(p, d)$$

With this notation in place XL may be expressed in an F_4 fashion by:

Algorithm 13 (XL_{F_4}).

```
def xlf4(F):
  """
  INPUT:
      F -- a finite subset of R

  OUTPUT:
      a finite subset of R

  Sel is fixed to identity: Sel(P) = P
  """
  G = F
  Ftildeplus[0] = F
  d = 0
  P = set([Pair(f,g) for f, g in G with f != g])
  while P != set():
    d += 1
    Pd = Sel( P ) # identity
    P = P.difference( Pd )
    Ld = XLLeft( Pd ).union( XLRight(Pd) )
    Ftildeplus[d] = reduction(Ld , G)
    for h in Ftildeplus[d]:
      P = P.union( set([ Pair(h, g) for g in G ]) )
      G = G.add( h )
  return G
```

In the original description [CKPS00] XL is not presented as an incremental algorithm in the way it is presented here. Version three of the variants presented at page 57 was chosen. Please note that XL is exponential in the parameter d and thus the runtime is dominated by the last iteration of Algorithm 13.

Algorithm 13 contains several redundancies to show the similarities between F_4 and XL. Also the procedure `symbolic_preprocessing` may be omitted in Algorithm 13 as all polynomials potentially generated by this subroutine are already included in L_d due to the "multiply" step of Algorithm 12.

The main theorem of [SKI04] is:

Theorem 4.3.2. *[SKI04] The algorithm XL computes a Gröbner basis G in $k[x_0, \ldots, x_{n-1}]$ such that $F \subseteq G$ and $\langle G \rangle = \langle F \rangle$.*

Proof. The proof is similar to the proof of the correctness of the F_4 algorithm [SKI04] so we refer the reader to [Fau99] or Section 4.1 respectively.

4.4 Specialized Attacks

So far all attacks in this thesis were applicable to any $\mathcal{MQ}$ problem and didn't exploit any special structure. Equation systems as derived from CTC however are highly structured. Every round contains the same equations but with different variables. CTC ideal bases may be viewed as 'iterated' systems of equations. The connections between these iterated systems are the output variables of round $i-1$ which are the input variables of round i and the key schedule.

This structure is exploited in this section. First the "Meet in the Middle" approach by Cid, Murphey, and Robshaw as presented in [CMR05] is described. Next an approach which might be called "Gröbner Surfing" is introduced which is my algorithm. Finally a combination of these two approaches and other possible approaches are discussed.

Whenever an algorithm in this section is timed, it uses Singular's Gröbner basis engine as the underlying implementation.

4.4.1 Meet in the Middle Attack

In [CMR05] the authors state:

> When working with systems with such structure, a promising technique to find the overall solution is, in effect, a meet-in-the-middle approach: rather than attempting to solve the full system of equations for n rounds (we assume that n is even), we can try to solve two subsystems with $n/2$ rounds, by considering the output of round $n/2$ (which is also the input of round $n/2+1$) as variables. By choosing an appropriate monomial ordering we obtain two sets of equations (each covering half of the encryption operation) that relate these variables with the round subkeys. These two systems can then be combined along with some other equations relating the round subkeys. This gives a third smaller system which can be solved to obtain the encryption key." [CMR05]

To summarize, the authors suggest to divide the problem into smaller ones by simply splitting at $N_r/2$. As an example consider a $CTC_{3,1,2}$ instance with plaintext $= 1,0,1$ and encryption key $= 0,1,1$. The matching $\mathcal{MQ}$ problem has 27 variables and 49 equations. Let the term ordering be *lex* for the following calculations.

The equation system is split in two halves *Left* and *Right*:

$$
\begin{aligned}
Left = & 1 + K_{000000} + X_{001000}, K_{000001} + X_{001001}, 1 + K_{000002} + X_{001002}, \\
& 1 + Y_{001000} + X_{001002} + X_{001001} + X_{001000} + X_{001000}X_{001001}, \\
& 1 + Y_{001001} + X_{001001} + X_{001000}X_{001002}, \\
& 1 + Y_{001001} + X_{001001} + X_{001000}Y_{001000}, \\
& Y_{001001} + Y_{001000} + X_{001002} + X_{001000}Y_{001001}, \\
& 1 + Y_{001002} + Y_{001001} + Y_{001000} + X_{001001} + X_{001001}X_{001002} + X_{001000}, \\
& 1 + Y_{001002} + Y_{001001} + Y_{001000} + X_{001001} + X_{001001}Y_{001000} + X_{001000}, \\
& X_{001001}Y_{001001} + X_{001000} + X_{001000}Y_{001002}, \\
& 1 + Y_{001000} + X_{001002} + X_{001001} + X_{001001}Y_{001002} + X_{001000}Y_{001002}, \\
& Y_{001002} + Y_{001000} + X_{001002}Y_{001000} + X_{001000}Y_{001002}, \\
& Y_{001002} + Y_{001000} + X_{001002} + X_{001002}Y_{001001} + X_{001000}, \\
& 1 + Y_{001001} + X_{001002}Y_{001002} + X_{001001} + X_{001000} + X_{001000}Y_{001002}, \\
& Y_{001002} + Y_{001000}Y_{001001} + X_{001000}, \\
& 1 + Y_{001002} + Y_{001001} + Y_{001000}Y_{001002} + X_{001001} + X_{001000}, \\
& Y_{001002} + Y_{001001} + Y_{001001}Y_{001002} + Y_{001000} + X_{001002} + X_{001000}, \\
& Z_{001000} + Y_{001001} + Y_{001000}, Z_{001001} + Y_{001002} + Y_{001001}, \\
& Z_{001002} + Y_{001000}, K_{000001} + K_{001000}, K_{000002} + K_{001001}, K_{000000} + K_{001002}, \\
& Z_{001000} + X_{002000} + K_{001000}, Z_{001001} + X_{002001} + K_{001001}, Z_{001002} + X_{002002} + K_{001002}.
\end{aligned}
$$

$$
\begin{aligned}
Right = & 1 + X_{002000} + X_{002001} + X_{002001}X_{002000} + X_{002002} + Y_{002000}, \\
& 1 + X_{002001} + X_{002002}X_{002000} + Y_{002001}, 1 + X_{002001} + Y_{002000}X_{002000} + Y_{002001}, \\
& X_{002002} + Y_{002000} + Y_{002001} + Y_{002001}X_{002000}, \\
& 1 + X_{002000} + X_{002001} + X_{002002}X_{002001} + Y_{002000} + Y_{002001} + Y_{002002}, \\
& 1 + X_{002000} + X_{002001} + Y_{002000} + Y_{002000}X_{002001} + Y_{002001} + Y_{002002}, \\
& X_{002000} + Y_{002001}X_{002001} + Y_{002002}X_{002000}, \\
& 1 + X_{002001} + X_{002002} + Y_{002000} + Y_{002002}X_{002000} + Y_{002002}X_{002001}, \\
& Y_{002000} + Y_{002000}X_{002002} + Y_{002002} + Y_{002002}X_{002000}, \\
& X_{002000} + X_{002002} + Y_{002000} + Y_{002001}X_{002002} + Y_{002002}, \\
& 1 + X_{002000} + X_{002001} + Y_{002001} + Y_{002002}X_{002000} + Y_{002002}X_{002002}, \\
& X_{002000} + Y_{002001}Y_{002000} + Y_{002002}, \\
& 1 + X_{002000} + X_{002001} + Y_{002001} + Y_{002002} + Y_{002002}Y_{002000}, \\
& X_{002000} + X_{002002} + Y_{002000} + Y_{002001} + Y_{002002} + Y_{002002}Y_{002001}, \\
& Y_{002000} + Y_{002001} + Z_{002000}, Y_{002001} + Y_{002002} + Z_{002001}, \\
& Y_{002000} + Z_{002002}, K_{000002} + K_{002000}, K_{000000} + K_{002001}, K_{000001} + K_{002002}, \\
& Z_{002000} + K_{002000}, Z_{002001} + K_{002001}, 1 + Z_{002002} + K_{002002}.
\end{aligned}
$$

Then an "appropriate monomial ordering" is an ordering which eliminates in the direction of the connection of those two equation systems. For instance a ring for *Left* could be

$$R_{Left} = \mathbb{F}_2[K_{001000}, K_{001001}, K_{001002}, X_{002000}, X_{002001}, X_{002002}, X_{001000}, X_{001001}, X_{001002}, \\ Y_{001000}, Y_{001001}, Y_{001002}, Z_{001000}, Z_{001001}, Z_{001002}, K_{000000}, K_{000001}, K_{000002}]$$

and a ring for *Right* could be

$$R_{Right} = \mathbb{F}_2[K_{002002}, K_{002001}, K_{002000}, Z_{002002}, Z_{002001}, Z_{002000}, Y_{002002}, Y_{002001}, Y_{002000}, \\ X_{002002}, X_{002001}, X_{002000}, K_{000000}, K_{000001}, K_{000002}].$$

A Gröbner basis gb_{Left} for *Left* in R_{Left} would then be

$$\begin{aligned} gb_{Left} =& K_{000002} + K_{000002}^2, K_{000001} + K_{000001}^2, \\ & K_{000000} + K_{000000}^2, 1 + K_{000002} + K_{000000} + K_{000000}K_{000001} + Z_{001002}, \\ & 1 + K_{000002} + K_{000001}K_{000002} + K_{000000}K_{000001} + Z_{001001}, \\ & 1 + K_{000001} + K_{000000}K_{000002} + K_{000000}K_{000001} + Z_{001000}, \\ & 1 + K_{000001} + K_{000001}K_{000002} + K_{000000} + K_{000000}K_{000002} + K_{000000}K_{000001} + Y_{001002}, \\ & K_{000002} + K_{000001} + K_{000000} + K_{000000}K_{000002} + Y_{001001}, \\ & 1 + K_{000002} + K_{000000} + K_{000000}K_{000001} + Y_{001000}, \\ & 1 + K_{000002} + X_{001002}, K_{000001} + X_{001001}, \\ & 1 + K_{000000} + X_{001000}, 1 + K_{000002} + K_{000000}K_{000001} + X_{002002}, \\ & 1 + K_{000001}K_{000002} + K_{000000}K_{000001} + X_{002001}, \\ & 1 + K_{000000}K_{000002} + K_{000000}K_{000001} + X_{002000}, \\ & K_{000000} + K_{001002}, K_{000002} + K_{001001}, \\ & K_{000001} + K_{001000}, \end{aligned}$$

and a Gröbner basis gb_{Right} for *Right* in R_{Right} would be

$$\begin{aligned} gb_{Right} =& K_{000002} + K_{000002}^2, K_{000001} + K_{000001}^2, \\ & K_{000000} + K_{000000}^2, K_{000001}K_{000002} + K_{000000} + X_{002000}, \\ & K_{000002} + K_{000001} + K_{000000} + K_{000000}K_{000001} + X_{002001}, \\ & K_{000002} + K_{000001}K_{000002} + K_{000000} + K_{000000}K_{000002} + K_{000000}K_{000001} + X_{002002}, \\ & 1 + K_{000001} + Y_{002000}, 1 + K_{000002} + K_{000001} + Y_{002001}, \\ & 1 + K_{000002} + K_{000001} + K_{000000} + Y_{002002}, K_{000002} + Z_{002000}, \\ & K_{000000} + Z_{002001}, 1 + K_{000001} + Z_{002002}, \\ & K_{000002} + K_{002000}, K_{000000} + K_{002001}, \\ & K_{000001} + K_{002002}. \end{aligned}$$

Combining these in the ring R and computing the Gröbner basis for the result produces a Gröbner basis for R:

$$\begin{aligned} gb = {} & K_{000002} + K_{000002}^2, K_{000002} + K_{000001}, \\ & 1 + K_{000002} + K_{000000}, 1 + K_{000002} + K_{001002}, \\ & K_{000002} + K_{001001}, K_{000002} + K_{001000}, \\ & K_{000002} + K_{002002}, 1 + K_{000002} + K_{002001}, \\ & K_{000002} + K_{002000}, 1 + K_{000002} + Z_{002002}, \\ & 1 + K_{000002} + Z_{002001}, K_{000002} + Z_{002000}, \\ & Z_{001002}, 1 + Z_{001001}, \\ & 1 + K_{000002} + Z_{001000}, K_{000002} + Y_{002002}, \\ & 1 + Y_{002001}, 1 + K_{000002} + Y_{002000}, \\ & K_{000002} + Y_{001002}, 1 + K_{000002} + Y_{001001}, \\ & Y_{001000}, 1 + K_{000002} + X_{002002}, \\ & 1 + K_{000002} + X_{002001}, 1 + X_{002000}, \\ & 1 + K_{000002} + X_{001002}, K_{000002} + X_{001001}, \\ & K_{000002} + X_{001000}. \end{aligned}$$

This idea was also implemented and tested in [CMR05] and was found to be more effective than computing the Gröbner basis directly for some instances of small scale variants of the AES. Similar results may be obtained for CTC. Timing results for $CTC_{3,1,N_r}$ and term ordering *lex* are shown in Figure 4.4 on page 66 for random CTC instances of the given sizes. Five samples were taken per run.

Cid, Murphey, and Robshaw also state in [CMR05]: "This technique is cryptographically intuitive and is in fact a simple application of Elimination Theory, in which the Groebner bases are computed with respect to the appropriate monomial ordering to eliminate the variables that do not appear in rounds $\frac{n}{2}$ and $\frac{n}{2}+1$. One problem with this approach is that computations using elimination orderings (such as lexicographic) are usually less efficient than those with degree orderings (such as graded reverse lexicographic). Thus, for more complex systems, we might expect that using lexicographic ordering in the two main subsystems would yield only limited benefit when compared with graded reverse lexicographic ordering for the full system. As an alternative, we could simply compute the Groebner bases for the two subsystems (using the most efficient ordering) and combine both results to compute the solution of the full set equations."[CMR05]

This approach was also implemented and benchmarked in [CMR05] and was found to be more effective for some selected instances of small scale variants of the AES. Again, the results are similar for CTC. Timing experiments for $CTC_{3,2,N_r}$ and term ordering *degrevlex* are shown in Figure 4.2 on the next page. However, as B increases this attack technique seems to become less efficient as suggested by the timing experiments shown in Figure 4.3 on the following page for $CTC_{3,3,N_r}$ and term ordering *degrevlex*.

4.4.2 Gröbner Surfing

"These results suggest the applicability of a more general divide-and-conquer approach to this problem, in which some form of (perhaps largely symbolic) pre-computation could be performed and then combined to produce the solution of the full system. This might be a promising direction and more research will assess whether this approach might increase the efficiency of algebraic attacks against the AES and related ciphers."[CMR05]

Motivated by this statement and the timing results of "Meet in the Middle" attacks another specialized approach was implemented for this thesis. This alternative approach is very simple, yet

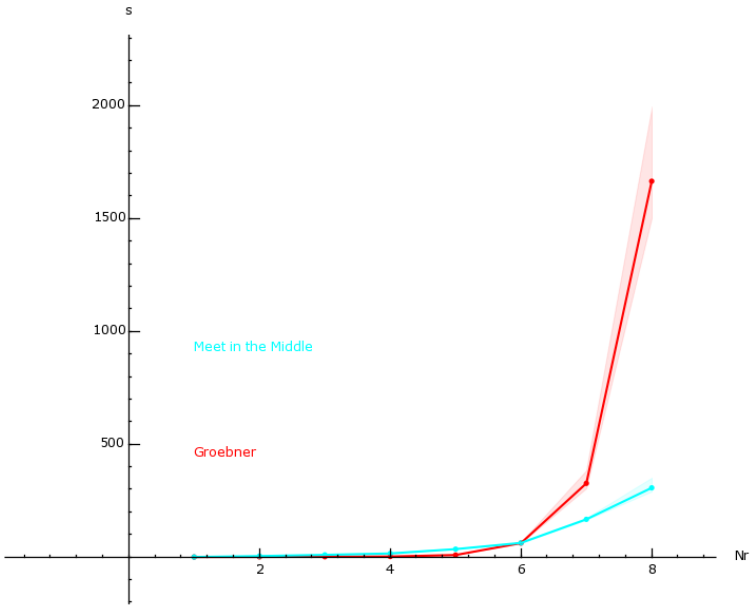

Figure 4.2: Runtimes for B=2 and term ordering *degrevlex*

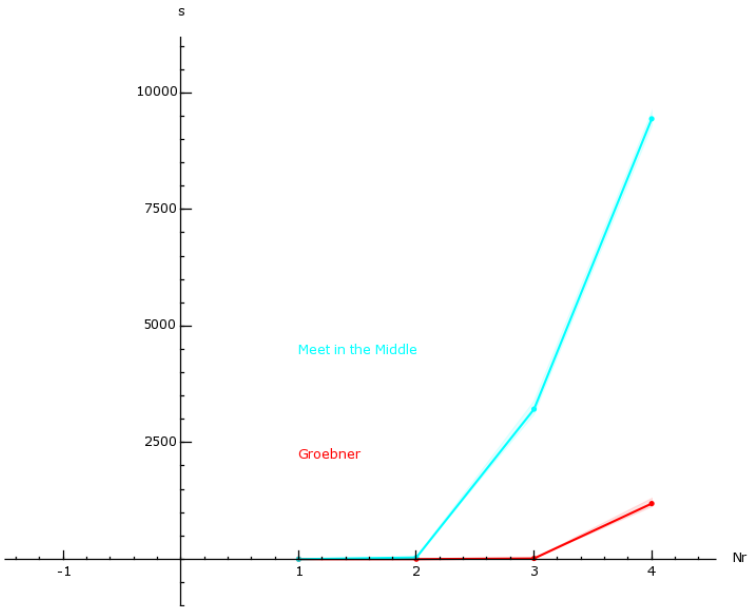

Figure 4.3: Runtimes for B=3 and term ordering *degrevlex*

it is faster for several CTC instances than "Meet in the Middle" and the naïve approach. Instead of computing a reduced Gröbner basis for all rounds rgb_F it computes the reduced Gröbner basis rgb_{i+1} up to round $i+1$ recursively as $rgb_{i+1} = rgb(gb_i + round_{i+1})$ with $rgb_0 = rgb(round_0)$ where $rgb(round_i)$ denotes any algorithm returning a reduced Gröbner basis for a given finite set of polynomials $round_i$. It is easy to see that $rgb_F = rgb_{N_r}$ if N_r denotes the number of rounds as the algorithm may be viewed as a specialized selection strategy for Gröbner basis algorithms and their correctness does not depend on the selection strategy.

Algorithm 14. Gröbner Surfing

```
def groebner_surf(F):
    """
    Computes a Groebner basis for a given finite set of polynomials
    divided into rounds.

    INPUT:
        F -- MQ problem with distinguishable rounds
    OUTPUT:
        a Groebner basis for the ideal spanned by F
    """
    R = F.ring()
    gb = []
    for i in range(len(F.round)):
        gb = R.ideal(gb + F.round[i]).groebner_basis()
    return gb
```

Lexicographical Monomial Ordering

First, we apply "Gröbner Surfing" to compute a *lexicographical* Gröbner basis. To make this approach work faster than a straight forward Gröbner basis calculation a similar strategy like in "Meet in the Middle" is required: The variables need to be ordered in such a way that elimination works in the right direction. A variable ordering satisfying this condition is the reverse of variable ordering described in Section 3.6. As an example consider a two round CTC with $B = 1$. Then a ring with a fast variable ordering for this approach could be:

$$\begin{aligned}\mathbb{F}_2[&K_{000000}, K_{000001}, K_{000002}, X_{001000}, X_{001001}, X_{001002}, Y_{001000},\\ &Y_{001001}, Y_{001002}, Z_{001000}, Z_{001001}, Z_{001002}, K_{001000}, K_{001001}, K_{001002},\\ &X_{002000}, X_{002001}, X_{002002}, Y_{002000}, Y_{002001}, Y_{002002}, Z_{002000}, Z_{002001},\\ &Z_{002002}, K_{002000}, K_{002001}, K_{002002}]\end{aligned}$$

Additionally, "Gröbner Surfing" may be used as a technique to compute the two Gröbner bases gb_{Left} and gb_{Right} in the "Meet in the Middle" approach, this may be called "Meet in the Middle Surfing".

Figure 4.4 on the next page shows the time it takes "Meet in the Middle", a naïve Gröbner basis computation, "Gröbner Surfing", and "Meet in the Middle Surfing" to compute a Gröbner basis for $CTC_{3,1,N_r}$ ideal bases. Five samples where taken per run. At least for these exotic CTC instances "Gröbner Surfing" and "Meet in the Middle Surfing" asymptotically outperform both the straight forward approach so as "Meet in the Middle".

Other Monomial Orderings

For the following discussion the following definition is needed:

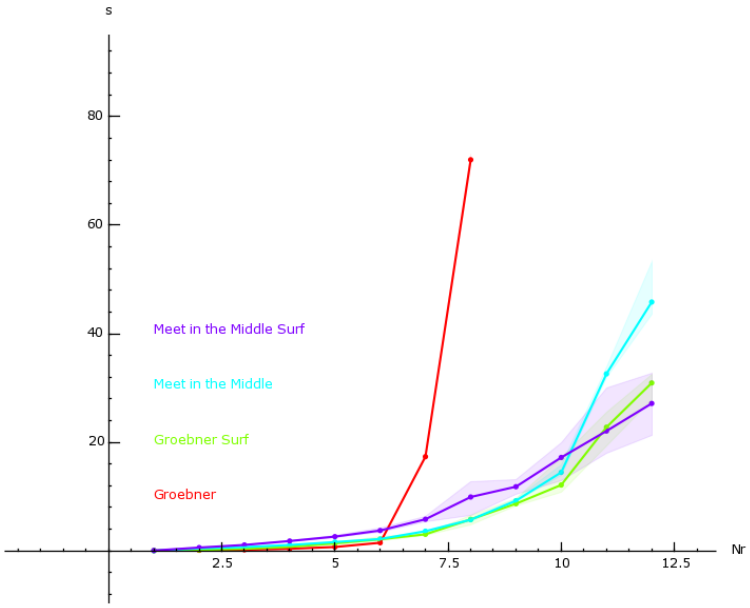

Figure 4.4: Runtimes for B=1 and term ordering *lex*

Definition 4.4.1 (Block Ordering)**.** *Let k be a field. Let $x = (x_0, \ldots, x_{n-1})$ and $y = (y_0, \ldots, y_{m-1})$ be two ordered sets of variables, $<_1$ a monomial ordering on $k[x]$ and $<_2$ a monomial ordering on $k[y]$. The product ordering (or block ordering) $< := (<_1, <_2)$ on $k[x, y]$ is the following:*

$$x^a y^b < x^A y^B \Leftrightarrow x^a <_1 x^A \text{ or } (x^a = x^A \text{ and } y^b <_2 y^B).$$

After noticing the results with "Gröbner Surfing" and *lex* Gröbner bases, Ralf-Phillip Weinmann suggested [Wei06] to use the "Gröbner Surfing" algorithm with a *block ordering* for the monomials. His idea is to use a fast graded monomial ordering inside the blocks and split the system of equations in blocks along the rounds of the cipher.

Using this approach "Gröbner Surfing" was able to compute a reduced Gröbner basis faster than a naïve Gröbner basis calculation for *degrevlex* using the Buchberger algorithm directly. Experiments suggest that a variable ordering as described in Section 3.6 is the best choice for this approach (This is `variable_order=1` in the provided implementation). This might be due to the fact that in this case the calculation of a *degrevlex* Gröbner basis inside the blocks is particularly easy. However, this fact is subject to further investigation.

Figures 4.5 on the facing page and 4.6 on the next page show the results of timing experiments which compare Singular's Buchberger algorithm (`std`) for *degrevlex* with "Gröbner Surfing" for the block ordering described above. Five samples were taken per run.

Please note, that running `std` for *degrevlex* is significantly faster than using a *product* or *block ordering* when using a straight forward `std`.

These benchmarks suggest that "Gröbner Surfing" combined with the appropriate block ordering computes a reduced Gröbner basis faster than the Buchberger algorithm applied directly.

Discussion

The timing experiments show that very simple specialized algorithms may provide a performance gain for computing Gröbner basis for CTC ideals. Even though the crossover points where the

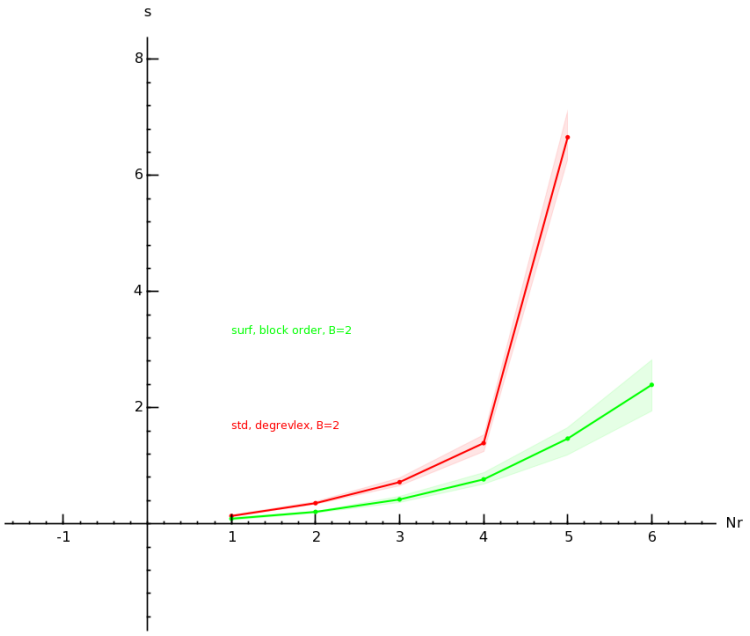

Figure 4.5: Runtimes for $B = 2$

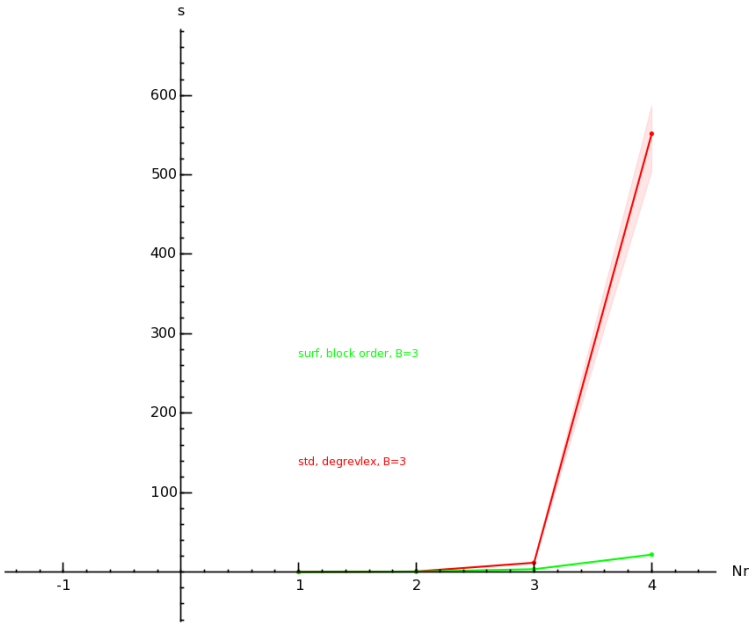

Figure 4.6: Runtimes for $B = 3$

specialized algorithms beat the naïve Gröbner basis calculation are sometimes beyond practical values for N_r, the results motivate further research in this direction. Especially the idea of Ralf-Phillip Weinmann to use *block orderings* in combination with algorithms like "Gröbner Surfing" seems very promising. Possible other approaches include:

- The "Meet in the Middle" approach is trivially parallelized.
- A recursive version of "Meet in the Middle" could compute a Gröbner basis for $n/2$ rounds using the "Meet in the Middle" approach on two $n/4$ blocks of rounds.
- A more fine graded "Gröbner Surfing" algorithm could use the layers (the S-boxes, the linear layer, the key addition layer, and the key schedule layer) instead of rounds as units to iterate over.

Similar improvements might be possible for $CTC_{3,B,1}$.

4.4.3 Using the CTCgb Gröbner Basis

In Section 3.6 a zero-dimensional Gröbner basis – called *CTCgb* – for CTC ideals was created. This section deals with the question how to use this result to cryptanalyse CTC. This section is based on [BPW06].

A first approach could be to convert the *degrevlex* Gröbner basis CTCgb to a lexicographical Gröbner basis using either FGLM [FGLM93], Gröbner Walk [CKM97], or any other Gröbner basis conversion algorithm. However, initial experiments show that this approach is not very efficient, as two rounds and $B = 1$ need approximately 8 s.

```
sage: attach "ctc.py"
sage: ctc=CTC(Nr=2)
sage: R = ctc.ring_factory2()
sage: F = ctc.MQgb_factory(R,p=[1,0,1],k=[1,1,0])
sage: I = Ideal(F.ideal().reduced_basis())
sage: I.is_groebner()
True

sage: time gb1 = I.transformed_basis("fglm")
CPU times: user 1.53 s, sys: 0.16 s, total: 1.70 s
Wall time: 8.14 # includes Singular time
```

Also a theoretical analysis of the runtime of FGLM when applied to CTCgb ideal bases shows that this approach doesn't provide a speed improvement. To perform this analysis, the following definitions need to be established:

Definition 4.4.2. *Let $R = k[x_0, \ldots, x_{n-1}]$. Then the k -space dimension of the ideal $I \subset R$ shall be denoted by $dim(R/I)$.*

Using Lemma 6.51 and Proposition 6.52 from [BW91] the following lemma can be deduced:

Lemma 4.4.1. *[BPW06] Let $\leq$ be a term order on $M(R)$ and G a Gröbner basis of I w.r.t. $\leq$. Then*

$$\begin{aligned} dim(R/I) &= \#\{m \in M(R) : s \nmid m \text{ for all } s \in LM(I)\} \\ &= \#\{m \in M(R) : s \nmid m \text{ for all } s \in LM(G)\} \end{aligned}$$

Applying the lemma to a Gröbner basis with univariate leading monomials yields the following corollary:

Corollary 4.4.2. *[BPW06] Let $G = \{g_0, \ldots, g_{n-1}\}$ be a Gröbner basis for the ideal $I \subset k[x_0, \ldots, x_{n-1}]$ with univariate head terms $x^{d_0}, \ldots, x^{d_{n-1}}$. Then $dim(R/I) = d_0 \cdot \cdots \cdot d_{n-1}$.*

Using this result a complexity bound for FGLM may expressed as follows:

Theorem 4.4.3. *[BPW06] Let k be a finite field and $R = k[x_0, \ldots, x_{n-1}]$. Furthermore, $G_1 \subset R$ is the Gröbner basis relative to a term order $<_1$ of an ideal I, and $D = dim(R/I)$. We can then convert G_1 into a Gröbner basis G_2 relative to a term order $<_2$ in $\mathcal{O}(nD^3)$ field operations.*

As linear polynomials don't contribute to the dimension $dim(R/I)$, it is sufficient to count the quadratic lead monomials in a CTCgb basis. There are Bs quadratic leading monomials (Y) per round. Additionally the last round contributes another Bs quadratic leading monomials. Consequently there are $Bs \cdot N_r + Bs$ quadratic leading monomials and $D = dim(R/I)$ is $2^{Bs \cdot (N_r+1)}$. The number of variables in the CTCgb basis is given by $n = 4BsN_r + Bs$. The complexity bound of the Gröbner basis conversion from the CTCgb ideal basis to another Gröbner basis is therefore given as $\mathcal{O}\big((4BsN_r + Bs) \cdot 2^{Bs(N_r+1)}\big)$. This is clearly worse than exhaustive key search.

It is unknown how to estimate complexity of the "Gröbner Walk" algorithm but experiments have shown that its runtime is worse than FGLM for CTCgb ideals. A test run was interrupted after 30 minutes where Gröbner Walk was used to convert a CTCgb basis to a *lex* basis. FGLM took 8.59 seconds for the same task.

Another approach is to use the fact that Gröbner bases allow to solve the ideal membership problem. For instance it could be tested if a linear polynomial of the form

$$k_i + C, \; C \in k$$

- with C being a key variable guess - lies in the ideal. A first problem with this approach is, that the CTCgb polynomial system has solutions over the closure of the ground field k, which means that one had to test for a polynomial

$$g = p\Pi(k_i + C_j)^{t_j}, t_j \in \mathbb{N}_0, C_j \in k$$

instead, where the C_j denote candidate values for the key variable and p is a product of irreducible non-linear polynomials. Moreover the dimension of the ideal again plays an important role here: it is an upper bound on the number of solutions of the corresponding polynomial system in the closure of the field. Hence the degree of g is expected to be very large.

Consequently, there is no known technique to exploit the fact that CTCgb is a zero-dimensional Gröbner basis for the CTC.

Chapter 5

Implementation Specific Notes

This section briefly describes how to use the provided implementation. As the SAGE computer algebra system is used for this thesis, the reader is referred to the SAGE documentation for a deeper introduction. SAGE provides the *SAGE Installation Guide* at http://sage.scipy.org/sage/doc/html/inst/index.html, the *SAGE Tutorial* at http://sage.scipy.org/sage/doc/html/tut/index.html, the *SAGE Reference Manual* at http://sage.scipy.org/sage/doc/html/ref/index.html, and the *SAGE Programming Manual* at http://sage.scipy.org/sage/doc/html/prog/index.html. Please note, that the full source code of this thesis is provided in Appendix A to ensure that it is distributed with this thesis.

This implementation has been tested to work with SAGE version 1.5.0.2. As SAGE is a fast moving project and API stability is not guaranteed until version 2.0, it might be possible that some aspects of this implementation will not work with later versions. Consequently, it is recommended to use the SAGE version provided on the CD-R.

The preferred way to install SAGE is by compiling it from source. For this, unpack `sage-1.5.0.2.tar`, enter the created directory, and type `make`. This should build SAGE and most of its dependencies automatically under Linux and Mac OSX. The build takes between 1 and 2 hours depending on the system it is running on.

After SAGE is built SAGE's `libcf` bindings need to be enabled. The file `all.py` in

```
$SAGE_ROOT/devel/sage/sage/libs
```

needs to be edited. The line

```
import sage.libs.cf.cf as cf
```

needs to be uncommented. Afterwards, `sage -br` should be called once to rebuild parts of SAGE and run it.

After the build is finished the `thesis.tar` archive may be unpacked wherever the user wishes to. Enter the just created directory and call `/PATH_TO_SAGE/sage`. This should bring up a SAGE prompt.

To create CTC ideals the ctc implementation needs to be loaded first. This is done either by loading or attaching it. Attaching a SAGE source file to SAGE means that it gets automatically reloaded if it changed on disk. To construct a random CTC ideal with $B = 2$, $N_r = 3$ the following commands must be executed at the SAGE prompt.

```
sage: attach 'ctc.py'
sage: F,s = ctc_MQ(B=2,Nr=3)
```

For information on a given function, method, or class the user may type:

```
sage: ctc_MQ?
Type:           function
Base Class:     <type 'function'>
String Form:    <function ctc_MQ at 0xaf896a04>
Namespace:      Interactive
File:           /home/martin/Uni-Bremen/ctc/code/ctc.py
Definition:     ctc_MQ(Nr=1, B=1, subst=0, term_order='degrevlex', qring=False, \
  variable_order=0, mqgb=False)
Docstring:
    Returns a CTC MQ problem with random plaintext and key (if those
    are not provided) for the given configuration.

    INPUT:
        Nr -- number of rounds (default: 1)
        B  -- number of 3-bit blocks (default: 1)
        subst -- how to substitute variables (default: 0)
                 0 - no substitution
                 1 - linear equations are used for substitution
                 2 - all equations are used for substitution
        term_order -- term ordering of the ring (default: degrevlex)
        qring -- use quotient ring implementation (default: False)
        variable_order -- controls the ordering of the variables (default: 0)
                          0 -- ctc.ring_factory is called
                          1 -- ctc.ring_factory2 is called
                          2 -- ctc.ring_factory2(reverse=True) is called
        mqgb -- construct a Groebner basis for ctc ideals
        plain -- plaintext
        key -- key
```

Also the source code may be inspected using `??`:

```
sage: is_PolynomialRing??
Type:           function
Base Class:     <type 'function'>
String Form:    <function is_PolynomialRing at 0xb0a7379c>
Namespace:      Interactive
File:           /opt/sage/local/lib/python2.5/site-packages/sage/rings/polynomial_ring.py
Definition:     is_PolynomialRing(x)
Source:
def is_PolynomialRing(x):
    return isinstance(x, PolynomialRing_generic)
```

As this interactive help system is in place, only the relevant classes and files will be named and the reader is referred to their docstrings for further help.

CTC is implemented in `ctc.py` as the class `CTC`. F_4 is implemented in `f4.py` as the class `F4`, DR in `dr.py` as the class `DR`, and XL in the files `xl.py` and `xl_pyx.spyx` as the class `XL`. The quotient ring polynomials are implemented in `polyf2.spyx` as the classes `MPolynomialGF2` and `MPolynomialRingGF2`.

The `.spyx` file format needs a bit of explanation. SAGE allows the user to provide scripts which get compiled to machine binaries before being executed. This allows the user to place time critical code in a script which may be as fast as native C code if implemented correctly. This has been done in `polyf2.spyx` and `xl_pyx.spyx`. However, these extension modules cannot be imported by other scripts as easily as non-compiled scripts. Thus the following

```
sage: attach 'ctc.py'
sage: F,s = ctc_MQ(qring=True)
```

will throw the following exception:

```
<type 'exceptions.NameError'>: global name 'MPolynomialRingGF2' is not defined
```

To avoid this the `polyf2.spyx` needs to be loaded manually.

```
sage: attach 'ctc.py'
sage: attach 'polyf2.spyx' #load manually.
sage: F,s = ctc_MQ(qring=True)
```

The same has to be done for `xl_pyx.spyx`.

Further examples for the implementation are given in the respective chapters of this thesis.

Chapter 6

Conclusions and Future Work

In the introduction of this thesis, two goals were given:

- A presentation of a variety of algebraic attack algorithms including mathematical background, examples, and full source code.
- Experiments with toy instances of CTC to aid a better understanding of the cipher.

The mathematical background necessary to understand algebraic attacks is mostly given in Chapter 2. Further details are presented in the respective sections for each algebraic attack algorithm (Sections 4.1, 4.2, 4.3, and 4.4) or ideal basis construction (Sections 3.3 and 3.6). Together with the full source code listing in Appendix A, this provides a self contained introduction to algebraic attacks against block ciphers.

Furthermore, this thesis contains an improved version of DR [TF05] which guarantees that the algorithm terminates (Section 4.2).

It is not surprising that no algorithm was found during the course of this thesis which breaks CTC. However, some small progress was made. Several specialized attacks were mounted against toy instances of CTC some of which were significantly ahead of naïve Gröbner basis algorithms (Section 4.4.1). Also new ideas were presented which improve on existing "divide and conquer" strategies (Section 4.4.2) and a zero-dimensional Gröbner basis for CTC ideals was constructed (Section 3.6) and analyzed (Section 4.4.3).

To perform experiments with toy instances of CTC, the cipher and the attacks had to be implemented first. Also much work was devoted to improve the SAGE computer algebra system which was chosen as the environment to perform experiments. Most of these improvements are already part of the upstream version of SAGE. Finally, the F_4 [Fau99] implementation provided in this thesis is faster than Singular – the fastest open-source Gröbner basis engine – for some instances of CTC.

As the CTC cipher has been specifically designed to scale down enough to function as a toy cipher, it is my hope that the provided work will also aid others with that task. For this, however, more work needs to be done.

A critical point this thesis failed to deliver is the lack of reliable data to compare benchmarks against. As the performance of Gröbner basis algorithms is at the moment estimated using benchmarks, reliable information about the state-of-the-art is required. While the implementations and algorithms of this thesis were benchmarked against Singular's algorithms and implementations, there is no strong evidence that Singular's best option was always selected. Even though benchmarks were performed carefully to try to select Singular's best algorithm, no large scale data is available to prove this assumption. As many factors may influence the runtime of Gröbner basis

calculations like monomial orderings, variable orderings, and algorithms, much data is needed to know which combination is the state of the art. However, these experiments may likely take weeks to perform and thus this task was unfortunately out of the scope of this thesis.

Further future work includes:

Optimized open-source implementations of several attack algorithms need to be provided and several aspects of open-source computer algebra systems need to be improved to provide free and open-source research tools for algebraic attacks on block ciphers. The SAGE computer algebra system provides a good environment for these improvements and parts of the implementation of this thesis will eventually become part of this computer algebra system.

On the theoretical side even more work is needed: In [Cou06] Nicolas Courtois describes his "Fast Algebraic Attack on Block Ciphers" as "an efficient method for computing Gröbner bases well-suited for systems of equations derived from block ciphers". Experiments have shown that very simple approaches may speed up Gröbner basis computations dramatically for specially selected CTC instances. However, it is unknown if these approaches scale up, i.e. if these approaches will result in better attacks on ciphers in more realistic dimensions. Still, this thesis gives reasons to believe that this direction is promissing.

Bibliography

[Bar06] 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.

[BCP97] 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.

[BDC03] A. Biryukov and C. De Canniere. Block ciphers and systems of quadratic equations. In *Proceedings Of Fast Software Encryption 2003*, pages 274–289. Springer, 2003. available at: http://www.cosic.esat.kuleuven.be/publications/article-14.pdf.

[BPW05] Johannes Buchmann, Andrei Pychkine, and Ralf-Philipp Weinmann. Block ciphers sensitive to gröbner basis attacks. Cryptology ePrint Archive, Report 2005/200, 2005. available at: http://eprint.iacr.org/2005/200.

[BPW06] Johannes Buchmann, Andrei Pychkine, and Ralf-Philipp Weinmann. A zero-dimensional gröbner basis for aes-128. In *Proceedings Of Fast Software Encryption 2006, LNCS 4047*, pages 78–88. Springer, 2006.

[Bri05a] Michael Brickenstein. Re: Sparse linear algebra, 2005. available at: http://mathforum.org/kb/plaintext.jspa?messageID=3852276.

[Bri05b] Michael Brickenstein. Slimgb: Gröbner bases with slim polynomials. In *Reports On Computer Algebra 35*. Centre For Computer Algebra, University Of Kaiserslautern, 2005. available at: http://www.mathematik.uni-kl.de/~zca/Reports_on_ca/35/paper_35_full.ps.gz.

[Bri06] Michael Brickenstein. Private communication, 10 2006.

[BW91] Thomas Becker and Volker Weispfenning. *Gröbner Bases - A Computational Approach To Commutative Algebra*. Springer, 1991.

[CKM97] S. Collart, M. Kalkbrener, and D. Mall. Converting bases with the gröbner walk. In *Journal Of Symbolic Computation 24*, pages 465–469. Academic Press, 1997.

[CKPS00] Nicolas Courtois, Alexander Klimov, Jacques Patarin, and Adi Shamir. Efficient algorithms for solving overdefined systems of multivariate polynomial equations. In *Proceedings Of Eurocrypt 2000, LNCS 1807*, pages 392–407. Springer, 2000.

[CL05] Carlos Cid and Gaëtan Leurent. An analysis of the xsl algorithm. In *Proceedings Of The Asiacrypt 2005, LNCS 3788*, pages 333–352. Springer, 2005.

[CLO05] David Cox, John Little, and Donal O'Shea. *Ideals, Varieties, And Algorithms*. Springer, 2005.

[CMR05] Carlos Cid, S. Murphy, and M. Robshaw. Small scale variants of the aes. In *Proceedings Of Fast Software Encryption 2005, LNCS 3557*, pages 145–162. Springer, 2005. available at http://www.isg.rhul.ac.uk/~sean/smallAES-fse05.pdf.

[CMR06] Carlos Cid, Sean Murphy, and Matthew Robshaw. *Algebraic Aspects Of The Advanced Encryption Standard*. Springer, 2006.

[Cou04] Nicolas Courtois. Algebraic attacks over gf(2**k), application to hfe challenge 2 and sflash-v2. In *Public Key Cryptography – PKC 2004*, pages 201–217. Springer, 2004.

[Cou06] Nicolas Courtois. How fast can be algebraic attacks on block ciphers? Cryptology ePrint Archive, Report 2006/168, 2006. available at: http://eprint.iacr.org/2006/168.pdf.

[CP02a] Nicolas Courtois and Josef Pieprzyk. Cryptanalysis of block ciphers with overdefined systems of equations. Cryptology ePrint Archive, Report 2002/044, 2002. available at http://eprint.iacr.org/2002/044.

[CP02b] Nicolas Courtois and Josef Pieprzyk. Cryptanalysis of block ciphers with overdefined systems of equations. In *Proceedings Of Asiacrypt 2002, LNCS 2501*, pages 267–287. Springer, 2002.

[CP03] Nicolas Courtois and Jacques Patarin. About the xl algorithm over gf(2). In *Topics in Cryptology - CT-RSA 2003: The Cryptographers' Track At The RSA Conference 2003; Proceedings*, pages 141–157. Springer, 2003.

[CW87] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. In *Annual ACM Symposium On Theory Of Computing Archive Proceedings Of The Nineteenth Annual ACM Conference On Theory Of Computing*, pages 1–6. ACM Press, 1987.

[DGS06] Jintai Ding, Jason E. Gowe, and Dieter S. Schmidt. Zhuang-zi: A new algorithm for solving multivariate polynomial equations over a finite field. Cryptology ePrint Archive, Report 2006/38, 2006. available at: http://eprint.iacr.org/2006/038.pdf.

[Din06] Jintai Ding. Ttm cryptosystems and the direct attack algorithms, 2006. available at http://math.uwyo.edu/RMMC/2006/coursematerial/wyo-4.pdf.

[DK] Orr Dunkelman and Nathan Keller. Linear cryptanalysis of ctc. Cryptology ePrint Archive, Report 2006/250. available at: http://eprint.iacr.org/2006/250.pdf.

[DR99] Joan Daemen and Vincent Rijmen. Aes proposal: Rijndael, 9 1999. available at http://csrc.nist.gov/CryptoToolkit/aes/rijndael/Rijndael-ammended.pdf.

[DR02] Joan Daemen and Vincent Rijmen. *The Design Of Rijndael: AES - The Advanced Encryption Standard*. Springer, 2002.

[Fau99] Jean-Charles Faugère. A new efficient algorithm for computing gröbner basis (f4), 1999. available at http://modular.ucsd.edu/129-05/refs/faugere_f4.pdf.

[Fau02] Jean-Charles Faugère. A new efficient algorithm for computing gröbner bases without reduction to zero (f5). In *Proceedings Of ISSAC*, pages 75–83. ACM Press, 2002.

[Fau06] Jean-Charles Faugère. Website of jean-charles faugère. http://fgbrs.lip6.fr/jcf/, 10 2006.

[Fel05] Adam Thomas Feldmann. A survey of attacks on multivariate cryptosystems. Master's thesis, 2005. available at http://etd.uwaterloo.ca/etd/atfeldma2005.pdf.

[FGLM93] Jean-Charles Faugère, P. Gianno, P. Lazard, and T. Mora. Efficient computation of zero-dimensional gröbner bases by change of ordering. In *Journal Of Symbolic Computation 16*, pages 329–344. Academic Press, 1993.

[FIP01] FIPS. Specification for the advanced encryption standard (aes)). Federal Information Processing Standards Publication 197, 2001. available at http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[GM88] R. Gebauer and H. M. Möller. On an installation of buchberger's algorithm. In *Journal Of Symbolic Computation 6 (2 And 3)*, pages 275–286. Academic Press, 1988.

[GPS05] G. M. Greuel, G. Pfister, and H. Schönemann. Singular 3.0. A Computer Algebra System for Polynomial Computations, Centre for Computer Algebra, University of Kaiserslautern, 2005. available at: http://www.singular.uni-kl.de.

[GS] Daniel R. Grayson and Michael E. Stillman. Macaulay 2, a software system for research in algebraic geometry. available at http://www.math.uiuc.edu/Macaulay2/.

[KR00] Martin Kreuzer and Lorenzo Robbiano. *Computational Commutative Algebra 1*. Springer, 2000.

[KS99] Aviad Kipnis and Adi Shamir. Cryptanalysis of the hfe public key cryptosystem. In *Proccedings Of Crypto 1999*, pages 19–30. Springer, 1999.

[KSY94] Deepak Kapur, Tushar Saxena, and Lu Yang. Algebraic and geometric reasoning using dixon resultants. In *ISSAC '94: Proceedings Of The International Symposium On Symbolic And Algebraic Computation*, pages 99–107, New York, NY, USA, 1994. ACM Press.

[MGH+05] Michael B. Monagan, Keith O. Geddes, K. Michael Heal, George Labahn, Stefan M. Vorkoetter, James McCarron, and Paul DeMarco. *Maple-10 Programming Guide*. Maplesoft, 2005.

[Moh01] T. Moh. On the method of "xl" and its inefficiency to ttm. Cryptology ePrint Archive, Report 2001/047, 2001. available at http://eprint.iacr.org/2001/047.ps.

[MR02] S. Murphy and M. Robshaw. Essential algebraic structure within the aes. In *Proceedings Of Crypto 2002, LNCS 2442*, pages 1–16. Springer, 2002. available at http://www.isg.rhul.ac.uk/~mrobshaw/rijndael/aes-crypto.pdf.

[Neu95] J. Neubüser. An invitation to computational group theory. In *Groups '93 Galway/St. Andrews, Vol. 2, Volume 212 Of London Mathematical Society Lecture Note Series*, pages 457–475. Cambridge Univ. Press, 1995.

[Seg04] A. J. M. Segers. Algebraic attacks from a gröbner basis perspective. Master's thesis, 2004. available at http://www.win.tue.nl/~henkvt/images/ReportSegersGB2-11-04.pdf.

[Sha49] C. E. Shannon. Communication theory of secrecy systems. In *Bell System Technical Journal 28*, pages 656–715, 1949.

[Shi] Mitsunari Shigeo. Horatu / ipa-smw. available at: http://www.math.kobe-u.ac.jp/HOME/kimura/Makoto_Sugita.txt.

[SJ05] William Stein and David Joyner. Sage: System for algebra and geometry experimentation. In *Comm. Computer Algebra*, volume 39, pages 61–64. 2005. available at http://modular.ucsd.edu/sage.

[SKI04] M. Sugita, M. Kawazoe, and H. Imai. Relation between xl algorithm and groebner bases algorithms. Cryptology ePrint Archive, Report 2004/112, 2004. available at http://eprint.iacr.org/2004/112.

[TF05] Xijin Tang and Yong Feng. A new efficient algorithm for solving systems of multivariate polynomial equations. Cryptology ePrint Archive, Report 2005/312, 2005. available at http://eprint.iacr.org/2005/312.

[Wei06] Ralf-Philipp Weinmann. Private communication, 12 2006.

[YCC04] Bo-Yin Yang, Jiun-Ming Chen, and Nicolas T. Courtois. On asymptotic security estimates in xl and gröbner bases-related algebraic cryptanalysis. In *Proceedings Of Information And Communications Security; 6th International Conference 2004, LNCS 3269*, pages 401–413. Springer, 2004.

Appendix A

Sourcecode Listing

A.1 Misc

Listing A.1: Assorted Functions

```
#
# AUTHOR: Martin Albrecht <malb@informatik.uni-bremen.de>
#         using MixIn code from http://www.linuxjournal.com/article/4540
#
# misc functionality

import re
from sage.rings.multi_polynomial_ring import polydict,MPolynomialRing_polydict_domain
from sage.rings.polydict import ETuple
from sage.rings.multi_polynomial_element import MPolynomial_polydict
from sage.interfaces.singular import SingularElement
from sage.matrix.constructor import Matrix
from sage.libs.cf import cf
from sage.misc.misc import verbose


def MixIn(pyClass, mixInClass, makeLast=0):
  """
  Mixes mixInClass into pyClass by making it superclass.

  If makeLast=1 mixInClass will be evaluated last, first otherwise
  """
  verbose("Mixing %s into %s."%(mixInClass,pyClass),level=2)

  if mixInClass not in pyClass.__bases__:
    if makeLast:
      pyClass.__bases__ += (mixInClass,)
    else:
      pyClass.__bases__ = (mixInClass,) + pyClass.__bases__

def MixInSAGE():
  """
  Mixes new functions into SAGE
  """
  MixIn(MPolynomialRing_polydict_domain, MPolynomialRing_polydict_domainMixIn)


class MPolynomialRing_polydict_domainMixIn:
  """
  Class to provide optimized methods for F4.

  """
  def _m_lcmfg_div_f(self,lcm,lm):
    """
    returns lcm/lm

    INPUT:
        lcm -- least common multiple of two monomials
        lm  -- leading monomial of f where lcm = LCM(f,g)
    """

    R = self

    one = R.base_ring()(1)

    lcm = lcm.dict().keys()[0]
    lm = lm.dict().keys()[0]

    res = lcm.esub(lm)

    return MPolynomial_polydict(R,polydict.PolyDict({res:one},\
                                                    force_int_exponents=False, \
                                                    force_etuples=False))

  def _m_lcm(self, f, g):
    """
    LCM for monomials.
```

```
    INPUT:
        f -- mpolynomial
        g -- mpolynomial
    """
    R = self

    one = R.base_ring()(1)

    f=f.dict().keys()[0]
    g=g.dict().keys()[0]


    length = len(f)

    res = {}

    nonzero = []

    for i in f.common_nonzero_positions(g):
      res[i] = max([f[i],g[i]])

    res =  R(polydict.PolyDict({ETuple(res,length):one},\
                                force_int_exponents=False,force_etuples=False))
    return res

  def _m_reduce_mod(self, f, G):
    """
    Tries to find a g in G where g.lm() divides f. If found g is
    returned, 0 otherwise.

    INPUT:
        f -- monomial
        G -- list/set of mpolynomials
    """

    for g in G:
      t = g.lm()
      flt = self._m_is_reducible_by(f,t)
      if flt != 0:
        return flt,g
    return 0,0


  def _m_pairwise_prime(self, h, g):
    """
    Returns True if h and g are pairwise prime

    INPUT:
        h -- monomial
        g -- monomial
    """
    return self._m_lcm(h,g)==h*g

  def _m_is_reducible_by(self, fm, tm):
    """
    Returns 0 if tm does not divide fm and the factor otherwise.

    INPUT:
        fm -- monomial
        tm -- monomial

    """

    R = self
    one = R.base_ring()(1)

    fm=fm.dict().keys()[0]

    tm=tm.dict().keys()[0]

    res = {}

    for i in fm.common_nonzero_positions(tm):
      tmp = fm[i] - tm[i]
      if tmp<0:
        return 0
      if tmp!=0:
        res[i]=tmp

    return MPolynomial_polydict( R , polydict.PolyDict( { ETuple(res,len(fm)):one},
                                                         force_int_exponents=False, \
                                                         force_etuples=False ) )

  def addwithcarry(self, tempvector, maxvector, pos):
    """
    Subroutine used by _m_all_divisors.
    """
    if tempvector[pos] < maxvector[pos]:
      tempvector[pos] += 1
    else:
      tempvector[pos] = 0
      tempvector = self.addwithcarry(tempvector, maxvector, pos + 1)
    return tempvector


  def _m_all_divisors(self,t):
    """
    INPUT:
      t -- \in T a term

    OUTPUT:
      a finite subset of T

    ALGORITHM: addwithcarry idea by Toon Segers
    """
```

```
    if not t.is_monomial():
      raise ArithmeticError, "Only monomials are supported"

    R = self
    one = self.base_ring()(1)
    M = set()

    maxvector = list(t.dict().keys()[0])

    tempvector =[0,]*len(maxvector)

    pos = 0

    while tempvector != maxvector:# and pos < len(maxvector):
      tempvector = self.addwithcarry(list(tempvector) , maxvector, pos)
      M.add(R(polydict.PolyDict({ETuple(tempvector):one}, \
                                force_int_exponents=False,force_etuples=False)))
    return M

def subst_poly(self, mapping, kcache=None):
  """
  evaluates a polynomial using libCF.

  INPUT:
    self    -- MPolynomial to fix
    mapping -- fix mapping

  OUTPUT:
    fixed MPolynomial

  """

  if self.is_constant():
    return self

  if isinstance(mapping,SingularElement):
    ret = mapping(self._singular_())
    return ret.sage_poly(self.parent(),kcache)

  v = cf.setBaseDomain(self.parent().base_ring())
  f=cf.CF(self,v)
  if isinstance(mapping,dict):
    mapping = tuple([(cf.CF(var,v),cf.CF(val,v)) for var,val in mapping.iteritems()])
  return f(mapping)._sage_(self.parent())

def flatten(l):
  """
  Flattens a given list,tuple,set with
  optional nested lists,tuples,sets.

  INPUT:
    l -- input list with lists as elements

  OUTPUT:
    generator of a flattened list

  EXAMPLE:
    sage: list(flatten([1,2,3,[4,[5,6],7]]))
      [1, 2, 3, 4, 5, 6, 7]
  """
  for elem in l:
    if type(elem) in [list,tuple,set]:
      for subelem in flatten(elem):
        yield subelem
    else:
      yield elem

def m_profile(cmd):
  """
  Profiles the cmd string using hotshot. The output is written to
  pythongrind.prof.
  """
  import hotshot
  filename = "pythongrind.prof"
  prof = hotshot.Profile(filename,lineevents=1)
  prof.run(cmd)
  prof.close()

def smtosm(As,base):
  """
  Singular Matrix to SAGE Matrix over base

  INPUT:
      As -- Singular matrix
      base -- base ring
  """
  A = Matrix(base,int(As.nrows()),int(As.ncols()))
  for x in range(int(As.nrows())):
    for y in range(int(As.ncols())):
      A[x,y]=As[x+1,y+1].sage_poly(base)

  return A

def density(A):
  """
  Returns the density of the matrix.  That is: If you choose an index
  at random with the probability as returned by this function you'll
  hit a non zero element.
  """
  count = ZZ(0)
  for x in range(A.nrows()):
    for y in range(A.ncols()):
      if  A[x,y]!=0:
        count+=1
  return Reals()(count/(A.nrows()*A.ncols()))

def s_pol(f,g):
```

```
    """
    S-Polynomial of f and g.

    ALGORITHM: Using Singular.
    """
    R = f.parent()
    if not R is g.parent():
        raise TypeError

    lcm = f.lm().lcm(g.lm())
    return R(lcm/f.lt()) * f - R(lcm/g.lt()) * g
```

Listing A.2: Polynomials over GF(2)

```
r"""

Implements multivariate polynomials over GF(2) in the quotient ring with the field
equations: $F_2[x_0,\dots,x_{n-1}]/\langle x_0, \dots, \x_{n-1} \rangle$.

AUTHOR: Martin Albrecht <malb@informatik.uni-bremen.de>

    * idea about polynomial representation taken from ipa-smw (by
      MITSUNARI Shigeo and 'dsk')

    * listhead implementation based on a C implementation by Till
      Backhaus and me (Martin Albrecht)

TODO:
    * reasonable benchmarks
    * speedup: switch from listhead to AVL tree to avoid Python overhead
    * cdef more ?
    * avoid PyObject * <-> object conversion
    * lm() for degrevlex much more expensive than lm() for lex, add degree field to monomials
"""

#
# typedefs
#

ctypedef unsigned int uint
ctypedef unsigned long ulong
ctypedef ulong monomial

#
# C includes
#

cdef extern from "polyf2.h":
    ulong count32(ulong)
    ulong count64(ulong)


cdef extern from "string.h":
    void *memcpy(void *dest, void *src, unsigned int n)

cdef extern from "Python.h":
    ctypedef struct PyTupleObject:
        void *ob_item   # we don't use this, but we can't use 'pass' here

    ctypedef struct PyListObject:
        void *ob_item   # we don't use this, but we can't use 'pass' here

    ctypedef struct PyTypeObject:
        PyTupleObject *tp_mro

    ctypedef struct PyObject:
        PyTypeObject *ob_type

    cdef PyObject * PyCObject_FromVoidPtr (void* cobj, void (*destr)(void *))
    cdef void * PyCObject_AsVoidPtr (PyObject* self)
    cdef PyObject * PyBuffer_FromMemory(  void *ptr, int size)

    cdef PyObject* PyDict_New(  )
    cdef PyObject* PyDict_GetItem(  PyObject *p, PyObject *key)
    cdef PyObject* PyTuple_GetItem(  PyObject *p, int pos)
    cdef int PyTuple_Size(  object p)
    cdef int PyDict_DelItem(  PyObject *p, PyObject *key)
    cdef int PyDict_SetItem(  PyObject *p, PyObject *key, PyObject *val)
    cdef int PyDict_Contains(  PyObject *p, PyObject *key)
    void Py_INCREF(object o)
    void Py_DECREF(object o)

#
# Python includes
#

import re

from sage.misc.functional import ceil
from sage.rings.finite_field import GF
from sage.rings.multi_polynomial_ring import MPolynomialRing,TermOrder
from sage.interfaces.singular import singular as singular_default
from sage.interfaces.singular import SingularElement
from sage.rings.integer_ring import ZZ
from sage.rings.ring cimport CommutativeRing
from sage.rings.multi_polynomial_ideal import MPolynomialIdeal

from sage.misc.misc import cputime

#
# Module Level Variables
#

cdef uint bits_per_word
```

```
cdef uint _bytes_per_word
cdef ulong oneL

oneL = 1
_bytes_per_word =  sizeof(unsigned long)
bits_per_word   = _bytes_per_word * 8

def get_wordlens():
    return _bytes_per_word,bits_per_word

#
# List Implementation
#

cdef struct listhead:
    monomial *monomial #element
    listhead *tail # next list element

cdef void push_element(listhead **list, monomial *elem):
    """
    adds an element to the front of a list
    """

    cdef listhead *tmp
    tmp = <listhead*>PyMem_Malloc(sizeof(listhead))
    if tmp == NULL:
        raise RuntimeError
    tmp.tail = list[0]
    tmp.monomial = elem
    list[0] = tmp

cdef monomial *pop_element(listhead **list):
    """
    returns the first element and removes it from the list
    """

    cdef listhead *tmp
    cdef monomial *ret
    tmp = list[0]

    if tmp == NULL:
        return NULL

    ret = tmp.monomial
    list[0] = (list[0]).tail
    PyMem_Free(tmp)
    return ret

cdef int remove_element(listhead **list, monomial *elem):
    """
    removes an element
    """

    cdef listhead *rem
    if list[0]:
        if list[0].monomial == elem :
            rem = list[0]
            list[0] = list[0].tail
            PyMem_Free( rem )
            return 0
        return remove_element( &(list[0].tail), elem)
    return -1

cdef listhead *create_listhead( monomial *elem ):
    """
    creates a list which contains only elem
    """
    cdef listhead *element
    element = <listhead*>PyMem_Malloc(sizeof(listhead))
    element.tail = NULL
    element.monomial = elem
    return element

cdef void free_list( listhead **list):
    """
    frees list structures and first class members
    """

    if list and list[0]:
        free_list( &((list[0]).tail) )
        PyMem_Free((list[0]).monomial)
        PyMem_Free(list[0])
        list[0]=NULL


#
# Monomial Arithmetic
#

cdef class MPolynomialGF2

cdef PyObject *monomial_to_python( monomial *elem, uint bytelen):
    """
    Returns Python object for the monomial elem of bytelength len
    """
    return PyBuffer_FromMemory(  elem, bytelen )

cdef monomial *monomial_multiply(monomial *left, monomial *right, uint wordlen):
    """
    Multiplication of two monomials of word length len

    INPUT:
        left  --
        right --
        len   -- left is left[0..len-1] and right is right[0..len-1]

    OUTPUT:
```

```
        left * right
    """
    cdef monomial *res
    cdef int i

    if left==NULL or right==NULL:
        return NULL

    res = <monomial*>PyMem_Malloc( wordlen * _bytes_per_word )

    # We may identify multiplication with OR:
    #
    #  x *  y = xy |  x *  x =  x
    # ------------------------------
    # 10 | 01 = 11 | 10 | 10 = 10
    for i from 0 <= i < wordlen:
        res[i]=(left[i] | right[i])
    return res

cdef monomial *monomial_division(monomial *left, monomial *right, uint wordlen):
    """
    Division of left by right, both of word length len

    INPUT:
        left - monomial
        right - monomial
        wordlen - lengths of both left and right in words

    OUTPUT:
        monomial = left/right

    WARNING: It is not checked if left is divisible by right. If it
        is not divisible by right the result has no meaning.
    """
    cdef monomial *res
    cdef int i
    res = <monomial*>PyMem_Malloc( wordlen * _bytes_per_word )

    # We may identify division with XOR:
    #
    #  xy /  y =  x | xy /  x =  y
    # ------------------------------
    # 11 ^  01 = 10 | 11 ^ 10 = 01

    for i from 0 <= i < wordlen:
        res[i]=(left[i] ^ right[i])
    return res

cdef uint is_divisible_by(monomial *left, monomial *right, uint wordlen):
    """
    Checks whether left is divisible by right, both of word length wordlen

    INPUT:
        left -  monomial
        right - monomial
        wordlen -

    OUTPUT: 1 or 0

    WARNING: Zeros are not dealt with

    TODO: maybe we can spare some loop cycles by merging this with division
    """
    cdef int i

    for i from 0 <= i < wordlen:
        if (left[i] ^ right[i])&(~left[i]):
            return 0
    return 1

cdef uint monomial_equals(monomial *left, monomial *right, uint wordlen):
    """
    Returns 0 if left!=right, 1 otherwise
    """
    cdef int i

    for i from 0 <= i < wordlen:
        if left[i] != right[i]:
            return 0
    return 1

cdef int monomial_compare_lex(monomial *left, monomial *right, uint len):
    """
    Compares left and right wirth respect to lex term order.

    WARNING: This function requires that the variables are stored from left to right:
    Let x > y then x ~= ...10... and y ~= ...01... .
    """
    cdef int i

    for i from 0 <= i < len:
        if left[i] < right[i]:
            return -1
        elif left[i] > right[i]:
            return 1
    return 0

cdef int monomial_compare_revlex(monomial *left, monomial *right, uint len):
    """
    Compares left and right wirth respect to revlex term order.

    WARNING: This function requires that the variables are stored from
    right to left: Let x > y then x ~= ...01... and ~= ...10... .
    """
    cdef int i

    for i from 0 <= i < len:
```

```
        if left[i] > right[i]:
            return -1
        elif left[i] < right[i]:
            return 1
    return 0

cdef int monomial_compare_deglex(monomial *left, monomial *right, uint len):
    """
    Compares left and right wirth respect to deglex term order.
    """
    cdef int i
    cdef uint ld,rd

    ld = monomial_degree(left,len)
    rd = monomial_degree(right,len)

    if ld<rd:
        return -1
    elif ld>rd:
        return 1

    for i from 0 <= i < len:
        if left[i] < right[i]:
            return -1
        elif left[i] > right[i]:
            return 1
    return 0

cdef int monomial_compare_degrevlex(monomial *left, monomial *right, uint len):
    """
    Compares left and right wirth respect to degrevlex term order

    This function requires that the variables are stored reversed
    already in a word.

    """
    cdef int i
    cdef uint ld,rd

    ld = monomial_degree(left,len)
    rd = monomial_degree(right,len)

    if ld<rd:
        return -1
    elif ld>rd:
        return 1

    for i from 0 <= i < len:
        if left[i] > right[i]:
            return -1
        elif left[i] < right[i]:
            return 1
    return 0

cdef monomial *monomial_copy(monomial *src, uint len):
    """
    Copies a monomial of wordlength len.
    """
    cdef monomial *dst
    dst = <monomial*>PyMem_Malloc( len * _bytes_per_word )

    memcpy( dst, src, len * _bytes_per_word )

    return dst


cdef uint monomial_degree(monomial *left, uint len):
    """
    bit counting in a monomial of worlength len.
    """
    cdef unsigned long v
    cdef unsigned long c
    cdef int i

    if bits_per_word == 32:
        c = 0
        for i from 0 <= i < len:
            v = left[i]
            count32(v)
            c = c + v

        return <uint>c

    elif bits_per_word == 64:
        c = 0
        for i from 0 <= i < len:
            v = left[i]
            count64(v)
            c = c + v

        return <uint>c



cdef uint monomial_pairwise_prime(monomial *left, monomial *right, uint wordlen):
    """
    Returns 1 if left and right are pairwise prime, 0 otherwise.
    """
    cdef int i

    for i from 0 <= i <wordlen:
        if (left[i] & right[i]):
            return 0
    return 1

cdef uint monomial_hash(monomial *m, uint wordlen):
```

```
    """
    Hash function similar to Python's tuple hash.
    """
    cdef uint *_m
    cdef uint value
    cdef uint roundconst
    cdef uint roundvar
    cdef int i

    _m = <uint*>m

    if bits_per_word == 64:
        wordlen = 2 * wordlen

    roundconst = 1000003
    value = 0x345678

    for i from 0 <= i < wordlen:

        #integer hash
        roundvar = _m[i]
        if roundvar == -1: roundvar == -2

        value = (roundconst * value) ^ roundvar

    value = value ^ wordlen

    if value == -1:
        value = -2

    return <uint>value #this could be a problem


cdef uint monomial_hasvar_lex(monomial *m, uint i, uint wordlen):
    """
    Returns 1 if the variable given by the index i is in the monomial
    of wordlength wordlen.
    """
    return m[i/bits_per_word] & (oneL<<(bits_per_word - (i % bits_per_word) - 1))

cdef uint monomial_hasvar_revlex(monomial *m, uint i, uint wordlen):
    """
    Returns 1 if the variable given by the index i is in the monomial
    of wordlength wordlen.
    """
    return m[wordlen - i/bits_per_word - 1] & (oneL<<(i %bits_per_word))

cdef class MPolynomialRingGF2(CommutativeRing):
    """
    $F_2[x_0 ... x_{n-1}]/\langle x_0^2+x_0, ... x_{n-1}^2+x_{n-} \rangle$
    """
    cdef int _nvars      # number of variables
    cdef int _wordlen    # len(MPolynomialGF2._value) in words
    cdef int _bytelen    # len(MPolynomialGF2._value) in bytes
    cdef object _names   # names (list)
    cdef object _order   # TermOrder
    cdef int _lex # direction variables are stored in
    cdef MPolynomialGF2 _one  # Do not recreate _one when needed
    cdef MPolynomialGF2 _zero # Do not recreate _zero when needed
    cdef int (*monomial_compare)(monomial*,monomial*,uint)
    cdef uint (*monomial_hasvar)(monomial*,uint,uint)
    cdef object __singular
    cdef object _base

    def __init__(MPolynomialRingGF2 self, n,names=None, order="degrevlex"):
        """

        Creates a new multivariate polynomial ring over GF(2) modulo
        the field ideal.

        INPUT:
            n -- number of variables
            names -- names assigned to those variables (default:None)
            order -- term order (default:degrevlex)

        """
        cdef monomial *one
        cdef int i

        self._order=order
        self._nvars = n
        if names is not None:
            if not isinstance(names,(list,tuple)):
                names = tuple(names)
            self._names = names
        else:
            self._names = []
            for i from 0 <= i < n:
                self._names.append("x%d"%i)

        # internal lengths
        self._wordlen = int(ceil(ZZ(n)/bits_per_word))
        self._bytelen = self._wordlen*_bytes_per_word

        one = <monomial*>PyMem_Malloc(_bytes_per_word * self._bytelen)
        for i from 0 <= i < self._wordlen:
            one[i] = 0

        self._one  = make_MPolynomialGF2(self, create_listhead(one) )
        self._zero = make_MPolynomialGF2(self, NULL)
        self._base = GF(2)

        if order=="lex":
            self.monomial_compare = monomial_compare_lex
            self.monomial_hasvar = monomial_hasvar_lex
            self._lex = 1
```

```
        elif order=="deglex":
            self.monomial_compare = monomial_compare_deglex
            self.monomial_hasvar = monomial_hasvar_lex
            self._lex = 1

        elif order=="revlex":
            self.monomial_compare = monomial_compare_revlex
            self.monomial_hasvar = monomial_hasvar_revlex
            self._lex = 0

        elif order=="degrevlex":
            self.monomial_compare = monomial_compare_degrevlex
            self.monomial_hasvar = monomial_hasvar_revlex
            self._lex = 0

        else:
            raise TypeError,"term order unknown"

    def gens(MPolynomialRingGF2 self):
        """
        List of variables
        """
        return map(self.gen,range(self._nvars))

    def assign_names(MPolynomialRingGF2 self, names):
        """
        """
        if isinstance(names,(list,tuple)):
            if self._nvars == len(names):
                self._names = list(names)
            else:
                raise TypeError, "length do not match"
        else:
            raise TypeError, "must be list or tuple"

    def gen(MPolynomialRingGF2 self, int i=0):
        """
        Returns the variable with index i

        INPUT:
            i -- index
        """
        cdef monomial _gen
        cdef monomial *val
        cdef int j,j2

        if i>=self._nvars:
            raise AttributeError

        val = <monomial*>PyMem_Malloc(self._bytelen)

        if self._lex:
            for j from 0 <= j < self._wordlen:
                if j==i/bits_per_word:
                    # fill from the left
                    _gen = oneL<< ( bits_per_word - ( i % bits_per_word) -1 )
                    val[j] = _gen
                else:
                    val[j] = 0
        else:
            for j from 0 <= j < self._wordlen:
                j2 = self._wordlen - j - 1
                if j==i/bits_per_word:
                    # fill from the right
                    _gen = oneL<< ( i % bits_per_word)
                    val[j2] = _gen
                else:
                    val[j2] = 0


        return make_MPolynomialGF2( self, create_listhead(val) )

    def base_ring(MPolynomialRingGF2 self):
        """
        Returns GF(2).
        """
        return self._base

    def __repr__(MPolynomialRingGF2 self):
        ideal_str = []
        for var in self._names:
            ideal_str.append("%s + %s^2"%(var,var))
        s = "Quotient of Polynomial Ring in %s over Finite Field of size 2 by the ideal (%s)"
        return s%((", ".join(self._names)),", ".join(ideal_str))

    def __call__(MPolynomialRingGF2 self,inp):
        """

        Call accepts Singular elements, MPolynomialGF2 elements,
        integers, and strings.

        """
        cdef int i,vi,bi,wi
        cdef listhead *monomials
        cdef monomial *ct
        monomials = NULL

        if isinstance(inp, MPolynomialGF2):
            return inp
        if isinstance(inp, SingularElement):
            #inp.reduce(inp.parent().current_ring().ideal())
            inp = re.sub(r'\^([0-9]*)','',str(inp))
        if isinstance(inp, str):
            var_dict = dict(zip(self._names,range(len(self._names))))
            inp = inp.replace(" ","")
            if inp=="0":
                return self._zero
```

```
            inp = inp.split("+")
            for i from 0 <= i < len(inp):
                t = inp[i].split("*")
                if t[0] == "1":
                    push_element(&monomials, monomial_copy(self._one._monomials.monomial, self._wordlen))
                else:
                    ct = monomial_copy(self._one._monomials.monomial, self._wordlen)
                    if self._lex:
                        for v in t:
                            vi = var_dict[v]
                            wi = vi/bits_per_word
                            bi = bits_per_word - (vi % bits_per_word) -1

                            ct[wi] = ct[wi] | (oneL<<bi)
                    else:
                        for v in t:
                            vi = var_dict[v]
                            wi = self._wordlen - vi/bits_per_word - 1
                            bi = vi % bits_per_word

                            ct[wi] = ct[wi] | (oneL<<bi)

                    push_element(&monomials, ct)

            return make_MPolynomialGF2(self,monomials)
        if int(inp)%2:
            return self._one
        else:
            return self._zero

    def ideal(MPolynomialRingGF2 self, gens):
        """
        Returns an ideal for the multivariate polynomial list gens.
        """
        gens = map(self,gens)
        return MPolynomialIdeal(self,gens)


    def _singular_(MPolynomialRingGF2 self, singular=singular_default):
        """
        Returns singular ring matching self
        """
        if self.__singular is not None:
            R = self.__singular
            if not (R.parent() is singular):
                return self._singular_init_(singular)
            try:
                R._check_valid()
            except ValueError:
                return self._singular_init_(singular)
            return R
        else:
            return self._singular_init_(singular)

    def _singular_init_(MPolynomialRingGF2 self, singular=singular_default):
        """
        Creates a singular ring matching self
        """
        r = singular.ring(2,tuple(self.gens()),order=self.term_order().singular_str())
        ideal_str = []
        for var in self._names:
            ideal_str.append("%s^2 + %s"%(var,var))
        ideal = singular.ideal(ideal_str)
        self.__singular = singular("%s"%ideal.name(),"qring")
        return self.__singular

    def ngens(MPolynomialRingGF2 self):
        """
        Number of variables.

        """
        return int(self._nvars)

    def term_order(MPolynomialRingGF2 self):
        """
        Term order.

        """
        return TermOrder(self._order)

    # these are special methods used by F4. These should not be called
    # outside their safe margins as described in their docstrings as
    # strange things may happen. These methods are supposed to be fast
    # not safe.

    def _m_lcmfg_div_f(MPolynomialRingGF2 self, MPolynomialGF2 lcm, MPolynomialGF2 f):
        """
        Returns g if lcm == LCM(f,g). Both input parameters must be
        monomials or the behavior is undefined.

        INPUT:
            lcm -- LCM(f,g)
            f   -- monomial

        OUTPUT:
            g - monomial

        """
        cdef monomial *l
        l = monomial_division( lcm._monomials.monomial, \
                               f._monomials.monomial, \
                               self._wordlen )
        return make_MPolynomialGF2(self,create_listhead(l))

    def _m_lcm(MPolynomialRingGF2 self, MPolynomialGF2 f, MPolynomialGF2 g):
        """
        LCM (== f*g) for monomials only.
```

```
        INPUT:
            f -- monomial
            g -- monomial

        OUTPUT:
            f*g
        """
        cdef monomial *m
        m =  monomial_multiply( f._monomials.monomial, g._monomials.monomial, self._wordlen )
        return make_MPolynomialGF2(self,create_listhead(m))

    def _m_reduce_mod(MPolynomialRingGF2 self,MPolynomialGF2 f, G):
        """
        Finds g in G where g.lm() divides f. If found (f/g.lm(),g)
        is returned, (self(0),self(0)) otherwise. f must be a
        monomial.

        INPUT:
            f -- monomial
            G -- iterable object containing MPolynomialGF2 elements

        OUTPUT:
            (f/g.lm(),g) or (self(0),self(0))
        """
        cdef monomial *r
        cdef monomial *gm
        cdef uint flt
        cdef int i
        cdef PyObject *cG
        cdef MPolynomialGF2 t
        cdef monomial *fm
        cdef uint wl

        fm = f._monomials.monomial
        wl = self._wordlen

        cG = <PyObject*>G

        if not PyObject_TypeCheck(G,tuple):
            raise TypeError,"tuple required"


        for i from 0 <= i < PyTuple_Size(G):
            t = (<object>PyTuple_GetItem(cG,i)).lm()
            #if not isinstance(t,MPolynomialGF2) or t.is_zero():
            #    raise TypeError, "non-zero MPolynomialGF2 required"
            gm = t._monomials.monomial
            flt = is_divisible_by(fm, gm, wl )
            if flt:
                r =  monomial_division(fm, gm, wl )
                return make_MPolynomialGF2(self,create_listhead(r)),G[i]
        return self._zero,self._zero

    def _m_pairwise_prime(MPolynomialRingGF2 self, MPolynomialGF2 f, MPolynomialGF2 g):
        """
        Return True if the monomial f is pairwise prime with the
        monomial g. False otherwise.

        INPUT:
            f -- monomial
            g -- monomial
        """
        return bool(monomial_pairwise_prime(f._monomials.monomial, \
                                            g._monomials.monomial, \
                                            self._wordlen))

    def _m_is_reducible_by(MPolynomialRingGF2 self, MPolynomialGF2 f, MPolynomialGF2 g):
        """
        Returns True if the monomial f is reducible by g. False otherwise.

        INPUT:
            f -- monomial
            g -- monomial

        """
        if not is_divisible_by(f._monomials.monomial, g._monomials.monomial, self._wordlen):
            return 0
        else:
            return make_MPolynomialGF2(self,
                                       create_listhead(monomial_division(f._monomials.monomial,
                                                                         g._monomials.monomial,
                                                                         self._wordlen )))


    cdef monomial * addwithcarry(MPolynomialRingGF2 self, \
                                 monomial *tempvector, \
                                 monomial *maxvector, \
                                 uint i):
        cdef uint wi,bi

        if self._lex:
            wi = i/bits_per_word
            bi = bits_per_word - (i % bits_per_word) -1
        else:
            wi = self._wordlen - i/bits_per_word - 1
            bi = i % bits_per_word

        if (tempvector[wi] & (oneL<<bi)) < (maxvector[wi] & (oneL<<bi)):
            tempvector[wi] = tempvector[wi] | (oneL<<bi)
        else:
            tempvector[wi] = tempvector[wi] & ~(oneL<<bi)
            tempvector =  self.addwithcarry(tempvector, maxvector, i+1)
        return tempvector


    def _m_all_divisors(MPolynomialRingGF2 self, MPolynomialGF2 t):
```

```
        """
        Returns all monomials that divide t.

        INPUT:
            t -- a monomial

        OUTPUT:
            all monomials that divide t.

        ALGORITHM: addwithcarry idea by Toon Segers
        """

        cdef monomial *maxvector
        cdef monomial *tempvector

        if t._monomials == NULL:
            return [self._zero]

        if t._monomials.tail:
            raise ArithmeticError, "Only monomials are supported"

        M = set()

        maxvector = t._monomials.monomial

        tempvector = monomial_copy(self._one._monomials.monomial, self._wordlen)

        cdef uint pos
        pos = 0

        while monomial_compare_lex(tempvector,maxvector,self._wordlen)!=0 and pos < self._nvars:
            tempvector = self.addwithcarry(tempvector , maxvector, pos)
            M.add(make_MPolynomialGF2(self,create_listhead(tempvector)))
            tempvector = monomial_copy(tempvector,self._wordlen)

        PyMem_Free(tempvector) # last is unused
        return M

    def _m_has_variable(MPolynomialRingGF2 self,  MPolynomialGF2 t, int i):
        """
        Returns True if the monomial t has the variable given by the index i.

        INPUT:
            t -- monomial
            i -- index
        """
        if 0>i or i>=self.ngens():
            raise AttributeError, "i must be >= 0 and < ngens"
        if self._lex:
            return bool(self.monomial_hasvar(t._monomials.monomial, i, self._wordlen))
        else:
            return bool(self.monomial_hasvar(t._monomials.monomial, i, self._wordlen))

cdef make_MPolynomialGF2(MPolynomialRingGF2 parent, listhead *lh):
    cdef MPolynomialGF2 p
    p = MPolynomialGF2(parent)
    p._monomials = lh
    return p

cdef class MPolynomialGF2:
    """
    Multivariate Polynomial in F_2[x_0 .... x_n]/(x_0^2+x_0,...,x_n^2+x_n)
    """
    cdef listhead *_monomials
    cdef MPolynomialRingGF2 _parent
    cdef MPolynomialGF2 _lm

    def __init__(MPolynomialGF2 self,parent):
        """
        Constructs 0 in parent.
        """
        self._parent = parent
        self._monomials = NULL

    def __dealloc__(MPolynomialGF2 self):
        free_list(&self._monomials)

    def __repr__(MPolynomialGF2 self):
        cdef int i, j, lex
        cdef listhead *m

        if self._monomials == NULL:
            return "0"
        s = ""
        m = self._monomials


        lex = self._parent._lex

        while m!=NULL: # monomials loop

            tmp = ""

            if lex:
                for i from 0 <= i < self._parent._wordlen: # word loop
                    for j from 0 <= j < bits_per_word: # bit loop
                        if m.monomial[i]==0:
                            continue
                        if (m.monomial[i] & (oneL<<(bits_per_word - j - 1))):
                            tmp=tmp+self._parent._names[i*bits_per_word+j]+"*"
            else:
                for i from 0 <= i < self._parent._wordlen: # word loop
                    for j from 0 <= j < bits_per_word: # bit loop
                        if m.monomial[i]==0:
                            continue
                        if (m.monomial[i] & (oneL<<j)):
                            tmp=tmp + \
```

```
                                        self._parent._names[(self._parent._wordlen -i -1)*bits_per_word+j]+"*"
            if tmp=="":
                tmp = "1*"
            s=s + tmp[:-1]+" + "
            m = m.tail
        if s==" + ":
            return "1"
        return s[:-3]

    def __mul__(MPolynomialGF2 self, MPolynomialGF2 other):
        cdef listhead *res, *self_iter, *other_iter
        cdef monomial *tmp, *m
        cdef int found, length
        cdef PyObject *_val, *_key, *res_dict

        res = NULL
        res_dict = PyDict_New()

        length = self._parent._bytelen

        self_iter = self._monomials
        while self_iter:
            other_iter = other._monomials
            while other_iter:
                m =  monomial_multiply( self_iter.monomial, other_iter.monomial, length )

                # search double entries (this is very expensive)
                _key = monomial_to_python(m, length)
                _val = PyDict_GetItem( res_dict , _key )

                if _val:
                    #remove
                    tmp = <monomial*>PyCObject_AsVoidPtr(<PyObject*>_val)
                    PyDict_DelItem( res_dict , _key )
                    # optimize this from O(n) to O(1)
                    Py_DECREF(<object>_key)
                    remove_element( &res, tmp )
                    PyMem_Free( tmp )
                    PyMem_Free( m )
                else:
                    #add
                    _val = PyCObject_FromVoidPtr(m,NULL)
                    PyDict_SetItem( res_dict , _key, _val )
                    Py_DECREF(<object>_key)
                    Py_DECREF(<object>_val)
                    push_element( &res, m )
                other_iter = other_iter.tail
            self_iter = self_iter.tail

        Py_DECREF(<object>res_dict)
        return make_MPolynomialGF2(self._parent, res)


    def __pow__(MPolynomialGF2 self,int exp,ignored):
        if exp==0:
            return self._parent._one
        else:
            return self

    def __add__(MPolynomialGF2 self, MPolynomialGF2 other):
        cdef listhead *res, *res_iter, *self_iter, *other_iter
        cdef monomial *tmp
        cdef PyObject *_key, *_val, *res_dict
        cdef int length

        length = self._parent._bytelen

        res_dict = PyDict_New()

        res = NULL
        self_iter  = self._monomials

        # copy self
        while self_iter:
            tmp = <monomial*>PyMem_Malloc( length )
            memcpy(tmp,self_iter.monomial, length )
            push_element(&res,tmp)
            _key = monomial_to_python(tmp, length)
            _val = PyCObject_FromVoidPtr(tmp,NULL)
            PyDict_SetItem( res_dict , _key, _val)
            Py_DECREF(<object>_key)
            Py_DECREF(<object>_val)
            self_iter = self_iter.tail

        # add those elements of other which are not in self, and
        # remove those elements which are in self and other
        other_iter = other._monomials
        while other_iter:
            _key = monomial_to_python(other_iter.monomial, length)
            _val = PyDict_GetItem( res_dict , _key )

            if _val:
                #remove
                tmp = <monomial*>PyCObject_AsVoidPtr(_val)
                remove_element(&res,tmp)
                #PyDict_DelItem(res_dict , _key)
                PyMem_Free(tmp)

            else:
                #add
                tmp = <monomial*>PyMem_Malloc(self._parent._bytelen)
                memcpy(tmp,other_iter.monomial,self._parent._bytelen)
                push_element(&res,tmp)
            Py_DECREF(<object>_key)

            other_iter = other_iter.tail
```

```
        Py_DECREF(<object>res_dict)
        return make_MPolynomialGF2(self._parent,res)

    def __sub__(MPolynomialGF2 self,MPolynomialGF2 other):
        # __sub__ == __add__ in GF(2)
        return self.__add__(other)

    def __div__(MPolynomialGF2 self, MPolynomialGF2 other):
        """
        Only division by a monomial is implemented
        """
        cdef monomial *l
        cdef listhead *self_iter,*res

        if other._monomials == NULL:
            raise ArithmeticError

        if self._monomials == NULL:
            return self

        if other._monomials.tail:
            raise NotImplementedError, "poly/poly not supported yet"

        self_iter = self._monomials
        res = NULL

        while self_iter:
            if not is_divisible_by(self_iter.monomial, \
                                   other._monomials.monomial, \
                                   self._parent._wordlen):
                free_list(&res)
                return self._parent._zero

            l = monomial_division( self_iter.monomial, \
                                   other._monomials.monomial, \
                                   self._parent._wordlen )
            push_element(&res,l)
            self_iter = self_iter.tail

        return make_MPolynomialGF2(self._parent,res)

    def lt(MPolynomialGF2 self):
        """
        Leading term of self.
        """
        return self.lm()

    def lc(MPolynomialGF2 self):
        """
        Leading coefficient of self.
        """
        if self.is_zero():
            return self._parent.base_ring()(0)
        else:
            return self._parent.base_ring()(1)

    def lm(MPolynomialGF2 self):
        """
        Leading monomial of self.
        """
        cdef listhead *self_iter
        cdef monomial *m, *tmp
        cdef int _cmp
        cdef int len
        cdef int (*monomial_compare)(monomial *,monomial *,uint)

        if self._monomials==NULL:
            return self

        if self._lm is not None:
            return self._lm

        len = self._parent._wordlen
        m = self._monomials.monomial
        self_iter = self._monomials.tail

        monomial_compare = self._parent.monomial_compare

        while self_iter:
            _cmp = monomial_compare(m,self_iter.monomial,len)
            if _cmp == -1:
                m = self_iter.monomial
            self_iter = self_iter.tail

        tmp = <monomial*>PyMem_Malloc(self._parent._bytelen)
        memcpy(tmp, m, self._parent._bytelen)

        self._lm = make_MPolynomialGF2(self._parent,create_listhead(tmp))

        return self._lm

    cdef int _cmp_(MPolynomialGF2 self, MPolynomialGF2 other):
        # this is very (!) inefficient
        left =  iter(sorted(self.monomials(),reverse=True))
        right = iter(sorted(other.monomials(),reverse=True))

        for m in left:
            try:
                n = right.next()
            except StopIteration:
                return 1 # left has terms, right doesn't, so left beats right
            ret = self._parent.monomial_compare((<MPolynomialGF2>m)._monomials.monomial, \
                                                (<MPolynomialGF2>n)._monomials.monomial, \
                                                self._parent._wordlen)
            if ret!=0:
                return ret # we have a difference
            #try next pair
```

```
        try:
            right.next()
            return -1 # right has terms, left doesn't
        except StopIteration:
            return 0 # right not has terms, left doesn't, they equal

    def __richcmp__(MPolynomialGF2 self, MPolynomialGF2 other, int op):
        cdef int res

        if not PyObject_TypeCheck(self, MPolynomialGF2) or not PyObject_TypeCheck(other,MPolynomialGF2):
            raise TypeError

        # handle ZERO first
        if self._monomials == NULL:
            if other._monomials == NULL:
                if op == 2 or op == 1 or op ==5: # == <= >=
                    return True
                else:
                    return False  # != < >
            else:
                if op == 3 or op == 0 or op ==1: # != < <=
                    return True
                else: # == > >=
                    return False
        if other._monomials == NULL:
            if op == 3 or op == 4 or op ==5: # != > >=
                return True
            else: # == < <=
                return False

        # now handle monomials
        if not self._monomials.tail and not other._monomials.tail:
            res = self._parent.monomial_compare(self._monomials.monomial, \
                                                other._monomials.monomial, \
                                                self._parent._wordlen)

            if op == 0: #<
                return bool(res<0)
            elif op == 2: #==
                return bool(res==0)
            elif op == 4: #>
                return bool(res>0)
            elif op == 1: #<=
                return bool(res<=0)
            elif op == 3: #!=
                return bool(res!=0)
            elif op == 5: #>=
                return bool(res>=0)

        # finally handle polynomials
        res = self._cmp_(other)
        if op == 0: #<
            return bool(res<0)
        elif op == 2: #==
            return bool(res==0)
        elif op == 4: #>
            return bool(res>0)
        elif op == 1: #<=
            return bool(res<=0)
        elif op == 3: #!=
            return bool(res!=0)
        elif op == 5: #>=
            return bool(res>=0)

    def bit_repr(MPolynomialGF2 self):
        """
        For debugging purposes: Returns a tuple for the monomial which
        represents the bits in in the internal monomial
        representation. This tuple is wordlen_bits *
        self.parent()._wordlen long.
        """
        cdef int i,j

        if not self._monomials:
            return tuple([0]*self._parent._wordlen * bits_per_word )
        if self._monomials.tail:
            raise NotImplementedError

        ret = []

        for i from 0 <= i < self._parent._wordlen:
            for j from 0 <= j < bits_per_word:
                if self._monomials.monomial[i] & (oneL << (bits_per_word - j  -1)):
                    ret.append(1)
                else:
                    ret.append(0)
        return tuple(ret)

    def exp_tuple(MPolynomialGF2 self):
        """
        Returns the exponent tuple for the monomial self. The tuple is
        self.parent().nvars() long.
        """
        cdef int i,wi,bi

        if not self._monomials:
            return tuple([0]*self._parent._nvars )
        if self._monomials.tail:
            raise NotImplementedError

        ret = []

        if self._parent._lex:
            for i from 0 <= i < self._parent._nvars:
                wi = i/bits_per_word
                bi = _bits_per_word - (i % _bits_per_word) -1
```

```
                if self._monomials.monomial[wi] & (oneL << bi):
                    ret.append(1)
                else:
                    ret.append(0)
        else:
            for i from 0 <= i < self._parent._nvars:
                wi = self._parent._wordlen - i/_bits_per_word - 1
                bi = i % _bits_per_word
                if self._monomials.monomial[wi] & (oneL << bi):
                    ret.append(1)
                else:
                    ret.append(0)

        return tuple(ret)


    def total_degree(MPolynomialGF2 self):
        """
        Returns the total degree of self:

        if
          self = m_0 + ... + m_n,
        then
          max(m_0,...,m_n)

        """
        cdef listhead *self_iter
        cdef int deg, _tdeg
        cdef int len

        len = self._parent._wordlen
        deg = 0


        self_iter = self._monomials
        while self_iter:
            _tdeg = monomial_degree(self_iter.monomial , len)
            if _tdeg > deg:
                deg = _tdeg
            self_iter = self_iter.tail

        return deg

    def is_zero(MPolynomialGF2 self):
        """
        return self == 0
        """
        return self._monomials == NULL


    def monomials(MPolynomialGF2 self):
        """
        List of monomials in self.
        """
        cdef listhead *self_iter
        cdef monomial *tmp
        cdef int len

        ret = list()
        len = self._parent._wordlen

        self_iter = self._monomials

        while self_iter:
            tmp = monomial_copy(self_iter.monomial, len)
            ret.append( make_MPolynomialGF2( self._parent, create_listhead(tmp)) )
            self_iter = self_iter.tail

        return ret

    def parent(MPolynomialGF2 self):
        """
        The ring self lives in.
        """
        return self._parent

    def _singular_(MPolynomialGF2 self, singular=singular_default):
        """
        Singular representation of self.
        """
        return singular(str(self))

    def __hash__(MPolynomialGF2 self):
        cdef listhead *self_iter
        cdef uint _hash
        #cdef uint roundconst
        cdef uint i

        #roundconst = 1000003
        _hash = 0x345678
        i = 0

        self_iter = self._monomials
        while self_iter:
            _hash =  _hash ^ monomial_hash(self_iter.monomial,self._parent._wordlen)
            i = i + 1
            self_iter = self_iter.tail

        _hash = _hash ^ i

        if _hash == -1:
            return -2
        return _hash

    def is_monomial(MPolynomialGF2 self):
        """
        Returns True is self is a monomial, False it it is a polynomial.
```

```
        """
        if self._monomials == NULL or self._monomials.tail == NULL:
            return True
        else:
            return False

    def variables(MPolynomialGF2 self):
        """
        Returns a list of varibales in this polynomial

        EXAMPLE:
            sage: R.<s,a,g,e> = MPolynomialRingGF2(4)
            sage: (s*a+g*e).variables()
            [s, a, g, e]
        """
        cdef monomial *tmp
        cdef listhead *self_iter
        cdef int i
        tmp = monomial_copy(self._parent._one._monomials.monomial, self._parent._wordlen)
        var_list = []

        self_iter = self._monomials

        while self_iter:
            for i from 0 <= i < self._parent._wordlen:
                tmp[i] = tmp[i] | self_iter.monomial[i]
            self_iter = self_iter.tail

        for i from 0 <= i < self._parent._nvars:
            if self._parent.monomial_hasvar(tmp,i,self._parent._wordlen):
                var_list.append(self._parent.gen(i))
        PyMem_Free(tmp)
        return var_list

    def append_monomial(MPolynomialGF2 self, MPolynomialGF2 m):
        """
        Appends a monomial. This is like add if it is known that the
        monomial m is not in self. Used by CoeffMatrix.

        INPUT:
            m -- monomial
        """

        cdef monomial *tmp
        tmp = monomial_copy(m._monomials.monomial, self._parent._wordlen)

        if self._monomials == NULL:
            return make_MPolynomialGF2(self._parent, create_listhead(tmp))
        else:
            push_element(&self._monomials,tmp)
            return self
```

Listing A.3: Polynomials over GF(2) (C)

```
#define count64(b) b = (b & 0x5555555555555555LU) + (b >> 1 & 0x5555555555555555LU); \
                   b = (b & 0x3333333333333333LU) + (b >> 2 & 0x3333333333333333LU); \
                   b = b + (b >> 4) & 0x0F0F0F0F0F0F0F0FLU; \
                   b = b + (b >> 8); \
                   b = b + (b >> 16); \
                   b = b + (b >> 32) & 0x0000007F; \

#define count32(b) b = b - ((b >> 1) & 0x55555555); \
                   b = (b & 0x33333333) + ((b >> 2) & 0x33333333); \
                   b = (((b + (b >> 4) & 0xF0F0F0F) * 0x1010101) >> 24); \
```

Listing A.4: Misc Singular Functions

```
version="20060627";
category="Miscellaneous";
// summary description of the library
info="
LIBRARY:    shared.lib Routines shared by several libs
AUTHOR:     Martin Albrecht, email: malb@informatik.uni-bremen.de

SEE ALSO:   mq.lib dr.lib

KEYWORDS:

PROCEDURES:
";

////////////////////////////////////////////////////////////////////////////
proc unique(l)
"USAGE:    unique(l); l iteratable
RETURN:    ideal; contains only unique elements and those sorted
NOTE:      It is believed that this implementation is very clumpsy and slow.
"
{
  ideal i;

  //make l an ideal
  for(int j=1; j<=size(l); j=j+1) {
    i[j] = l[j];
  }
  //sort it
  i = sort(i)[1];

  list l2 = list();

  //and remove doubles
  poly last = 0;
  for(int j=1 ; j<=size(i); j=j+1) {
```

```
    if(i[j]!=last) {
      l2 = insert(l2,i[j]);
    }
    last = i[j];
  }

  i = ideal();

  for(int j=1; j<=size(l); j=j+1) {
    i[j] = l2[j];
  }
  return(i);
}
////////////////////////////////////////////////////////////////////////



////////////////////////////////////////////////////////////////////////
proc coeff_matrix(gens, monomials, variables)
"
USAGE:   coeff_matrix(gens,monomials,variables); gens, monomials : ideal, variables : list
RETURN:  A,v; so that gens = A*v and A does not contain elements in variables
NOTE:    Returns the coeffient matrix for given set of gens (polynomials)
         and monomials occuring in those gens. The returned matrix may
         contain paramerters which do not occure in the product variables.
"
{
  ideal i1 = gens;
  ideal i2 = monomials;

  list l = compress(transpose(coeffs(i1,i2,variables))),i2;

  return(l);

}
////////////////////////////////////////////////////////////////////////


////////////////////////////////////////////////////////////////////////
proc cut(intvec in, int cutsize)
// Reduces a intvec to size cutsize
{
  intvec out;
  for(int i=1; i<=cutsize ; i=i+1) {
    out[i]=in[i];
  }
  return(out);
}
////////////////////////////////////////////////////////////////////////


////////////////////////////////////////////////////////////////////////
proc range(int length)
// Returns an intvec of length length whichs values are equivalent to
// their indices.  This is a rough equivalent to the python range()
// function.

{
  intvec ret;
  for(int i=1; i<=length; i=i+1) {
    ret[i]=i;
  }
  return(ret);
}
////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////

proc exclude_column(matrix M, int ex)
"USAGE:   exclude_column(M,ex); M matrix, ex int
RETURN:   submatrix with column ex excluded
"
{
  intvec cols;
  int j=1;

  for(int i=1 ; i<=ncols(M) ; i=i+1) {
    if(i!=ex) {
      cols[j]=j;
      j=j+1;
    }
  }
  return(submat(M,range(nrows(M)),cols));
}
////////////////////////////////////////////////////////////////////////


////////////////////////////////////////////////////////////////////////
proc ideal2vector(ideal i)
"USAGE:   ideal2vector(i) i ideal
RETURN:   m; vector where m[j,1] = i[j]
"
{
  matrix m[size(i)][1];

  for(int j=1 ; j<=size(i) ; j=j+1) {
    m[j,1] = i[j];
  }
  return(m);
}
////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////
proc list2ideal(list l)
"USAGE:   list2ideal(l) l list
RETURN:   i; ideal where i[j] = l[j]
"
```

```
{
  ideal i;

  for(int j=1 ; j<=size(l) ; j=j+1) {
    i[j] = l[j];
  }
  return(i);
}
////////////////////////////////////////////////////////////////////////////


////////////////////////////////////////////////////////////////////////////
proc findelem(l, elem)
"
NOTE:    This is the dumpest possible algorithm to do this. However Singular's list
         implementation is very basic so some work would be necessary to do something
         like binary search.
"
{
  for(int i=1; i<=size(l); i=i+1) {
    if(l[i]==elem) {
      return(1);
    }
  }
  return(0);
}

proc reductor(f, l)
"
"
{
  for(int i=1; i<=size(l); i=i+1) {
    poly t = l[i];
    poly flt = f/lead(t);
    if(flt!=0) {
      list ret = flt,l[i];
      return(ret);
    }
  }
  list ret = 0,0;
  return(ret);
}

////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
proc toF2(ideal i)
"""
 USAGE: Converts parameter i (ideal over GF(2^3)) to ideal over GF(2)

 AUTHOR: Michael Brickenstein <brickenstein@googlemail.com>

 INPUT:
      i -- ideal in F_{2^n}[x_0,...,x_n-1]

 OUTPUT:
      ideal in F_{2}[x_0,...,x_n+1]

 EXAMPLE:
      ring r=(8,a),(w,x,y,z),lp;
      ideal i=w2-ax,x-a,y+x,y-a,z-y-1;
      minpoly;
      def res_ring=toF2(i);
      setring res_ring;
      result;

 WARNING:
      until ringlist is fixed, we assume, that the param is called a
"""
{

  def old_ring=basering;
  int nvars_old=nvars(old_ring);

  string myminpoly_str=string(minpoly);
  string i_str=string(i);

  execute("ring helper=2,("+parstr(old_ring)+","+varstr(old_ring)+"),lp;");
  execute("poly myminpoly="+myminpoly_str+";");
  execute("ideal i="+ i_str+";");

  //now have minpoly and i in var format
  int r=deg(myminpoly);

  ring intermediate_ring=2,x(1..(nvars_old*r+1)),dp;

  poly im_of_param=var(1);
  int index, sum_index;
  ideal map_ideal;
  int var_index=1;
  poly sum;

  // map e.g. x -> a^2*x_2 + a*x_1 + a*x_0, where x_0,..,x_2 represent
  // the bits of x
  map_ideal[1]=var(1);//the param
  for(index=1;index<=nvars_old;index++){
    sum=0;

    for(sum_index=0;sum_index<r;sum_index++){
        var_index++;
        sum=sum+im_of_param^sum_index*var(var_index);
    }
    map_ideal[index+1]=sum;
  }

  map m=helper,map_ideal;
  ideal i=m(i);
```

```
  //print(i);
  //print("");

  poly myminpoly=m(myminpoly);
  i=reduce(i,std(myminpoly));

  //print(i);
  //print("");

  ideal result;
  matrix cm;

  // split e.g. a^2*x0+a*x1+x2 to x0,x1,x2
  for(index=1;index<=size(i);index++) {
    cm=coef(i[index],var(1));
    result=result+ideal(submat(cm,2,1..ncols(cm)));
  }

  //print(result);
  //print("");

  //eliminate parameter, change order to lp
  ring result_ring=2,(x(1..(nvars_old*r))),lp;
  ideal map_ideal;
  for(index=1;index<=nvars_old*r;index++){
  map_ideal[index+1]=var(index);
  }
  map m=intermediate_ring, map_ideal;
  ideal result=m(result);
  export(result);
  return(result_ring);
}
```

A.2 MQ

Listing A.5: Multivariate Polynomial Equation System

```
#!/usr/bin/env sage-python
#
# -*- Mode: Python -*-
# # vi:si:et:sw=4:sts=4:ts=4

"""
 AUTHOR: Martin Albrecht <malb@informatik.uni-bremen.de>

 This class represents a Multivariate Polynomial System which is ought
 to be attacked by instances of AlgebraicAttack like XL,F4,DR
"""

# system
import sys, pdb, os, copy
from random import random


# sage
from sage.structure.sage_object import SageObject
from sage.rings.multi_polynomial_ring import *
from sage.rings.polydict import *
from sage.libs.cf import cf
from sage.interfaces.singular import singular
from sage.misc.misc import verbose
from sage.rings.multi_polynomial_element import *
import sage.libs.ntl.all as ntl
from sage.misc.misc import mul

from coeff_matrix import *
from misc import *

class MQ(SageObject):
    """
    Attack me if you can.
    """

    def __init__(self,ring=None,rounds=None):
        """
        Constructs a MQ Problem for a given ring and list of
        polynomials in this ring.

        INPUT:
            ring   -- base ring
            rounds -- this is an iterable object of iterable objects
                      which contain polynomials in the base_ring. So
                      e.g., a list of lists will do. rounds[0] contains
                      all polynomials of the 0-th round, rounds[i] contains
                      all polynomials of the i-th round, etc.
        """

        gens = flatten(rounds)

        if ring != None:
            self._ring=ring
        else:
            self._ring,gens = self.minimal_ring(gens)
        self._base_ring = self._ring.base_ring()
        self._gens=list(gens)
        self.terminate = lambda x: False # solve all!
        self.round=rounds
```

```
    def _repr_(self):
        """
        String representation
        """
        return "Multivariate polynomial equation system with %d variables and %d polynomials (gens)."\
               %(self.nvariables(),self.ngens())
    def __iter__(self):
        """
        Iterating over the system means iterating over its generators
        """
        for f in self._gens:
            yield f

    def __len__(self):
        """
        Length of this system is number of generators
        """
        try:
            return self.__len
        except AttributeError:
            self.__len = len(list(self._gens))
            return self.__len

    def __getattr__(self,attrib):
        try:
            return getattr(self._gens,attrib)
        except AttributeError:
            raise AttributeError

    def __getitem__(self,key):
        """
        """
        return self._gens[key]

    def __contains__(self,element):
        """
        """
        return element in self._gens

    def __add__(self,elem):
        """
        """
        return self.parent()(self._ring, self._gens+list(elem))

    def __setitem__(self,key,value):
        """
        """
        self._gens[key] = value

    def __list__(self):
        """
        """
        return self._gens

    def copy(self):
        """
        Shallow copy of self.
        """
        return copy.copy(self)

    def parent(self):
        """
        """
        return type(self)

    # Queries: monomials, terms, variables, polynomials, linear polynomials

    def _monomials_(self,system=None):
        """
        """
        if not system:
            system = self

        s = set([m for f in system for m in f.dict().keys() ])
        return list(s)

    def monomials(self,system=None):
        """
        Returns a list of all monomials occuring in this polynomial
        system.
        """

        #return [ MPolynomial_polydict(self._ring,
        #                              PolyDict( {m:int(1)},force_int_exponents=False,
        #                                        force_etuples=False) )
        #          for m in self._monomials_(system) ]
        return list(set([m for f in self for m in f.monomials() ]))

    def nmonomials(self):
        """
        """
        return len(self._monomials_())

    def variables(self, gens=None):
        if gens==None:
            gens = self._gens

        return set([m  for f in gens for m in f.variables() ])

    def nvariables(self):
        """
        Number of variables in the ring
        """
        return len(self.variables())

    def ngens(self):
```

```
        """
        Number of generators (polynomials)
        """
        try:
            return self.__len
        except AttributeError:
            return len(self)

    def _terms_(self):
        """
        """
        s = set()
        for f in self._gens:
            map( s.add, f.element().dict().iteritems())
        return list(s)

    def terms(self):
        """
        Returns a list of all terms occuring in this polynomial
        system.
        """
        return [self._ring(PolyDict({m:c}, force_etuples=False, force_int_exponents=False)) \
                for m,c in self._terms_() ]

    def nterms(self):
        """
        Number of terms
        """
        return len(self._terms_())


    def lingens(self):
        return [e for e in self._gens if e.total_degree() < 2]

    def nlingens(self):
        return len(self.lingens())

    def gens(self):
        return list(self._gens)

    # coefficient matrix

    def coeff_matrix(self,param=None, T=None):
        """
        Returns the coefficient matrix A so as the monomial vector v
        corresponding to this multivariate polynomial equation
        system. So

         x=A*v

        where x is a vector containing all equations of this system.

        INPUT:
            param -- one variable of this system's ring may be treated
                     as a parameter instead of a variable
            T     -- term order to use

        OUTPUT:
            (A,v) where (A*v) is a vector containing all euqations
            of this polynomial system.
        """

        if T==None:
            T = self._ring.term_order()

        if not isinstance(T,list):
            T=TermOrder(T)

        if param:
            return self.coeff_matrix_w_param(param, T)
        else:
            return self.coeff_matrix_wo_param(T)

    def coeff_matrix_wo_param(self, T='lex'):
        """
        See MQ.matrix()
        """

        r = self._ring

        m = self.monomials()
        m = sorted(m,reverse=True)
        if isinstance(r,MPolynomialRing_polydict_domain):
            m_fast = [ f.dict().keys()[0] for f in m]
        else: #MPolynomialRingGF2
            m_fast = m

        #construct dictionary for fast lookups
        v = dict( map(lambda x,y:(x,y), m_fast , range(0,len(m_fast)) ) )

        A = CoeffMatrix_modint( r.base_ring() ,
                                len( self._gens ) ,
                                len(v) )
        if isinstance(r,MPolynomialRing_polydict_domain):
            for x in range( 0 , len(self._gens) ):
                poly = self._gens[x]
                poly_dict = poly.dict()
                for y,val in poly_dict.iteritems():
                    A[ x , v[y] ] = int(val)
        else:
            for x in range( 0 , len(self._gens) ):
                poly = self._gens[x]
                for y in poly.monomials():
                    A[ x , v[y] ] = 1

        return ( A , MonomialVector(m) )
```

```
    def coeff_matrix_wo_param_singular(self, T='lex'):
        """
        See MQ.matrix()
        """

        r = self._ring
        varstr = str(mul(r.gens()))

        m = sorted(set([m for f in self for m,c in f.coef(varstr)]))
        #sort list so correct variables are eliminated
        m = self.poly_sort(m, T)

        m_fast = m

        #construct dictionary for fast lookups
        v = dict( map(lambda x,y:(x,y), m_fast , range(0,len(m_fast)) ) )

        A = CoeffMatrix_modint( r.base_ring() ,
                                len( self._gens ) ,
                                len(v) )

        for x in range( 0 , len(self._gens) ):
            poly = self._gens[x]
            for mon,coef in poly.coef(varstr):
                A[ x , v[mon] ] = int(c)

        return ( A , singular.matrix(len(m),1,m) )

    def coeff_matrix_w_param(self,param, T='lex'):
        """
        See MQ.coeff_matrix()

        """

        r = self._ring
        m = self.monomials()

        v = cf.setBaseDomain(r.base_ring())
        repl = ((cf.CF(param,v),cf.CF(r(1))),)
        kcache = {}
        m = list(set([(cf.CF(f,v,kcache=kcache)(repl))._sage_(r,kcache=kcache) for f in m ]))

        param_idx = param._variable_indices_()[0]

        exps = [int(0)]*r.ngens()

        #sort list so correct variables are eliminated
        m = self.poly_sort(m,T)

        m_fast = [ f.element().dict().keys()[0] for f in m]

        #construct dictionary for fast indices lookup
        v = dict( map(lambda x,y:(x,y), m_fast , range(0,len(m_fast)) ) )

        A=Matrix( r , len( self._gens ) , len(v) , sparse=True )

        for x in range( 0 , len(self._gens) ):
            poly = self._gens[x]
            poly_dict = poly.element().dict()
            for y in poly_dict:
                if y[param_idx]!=0:

                    #remember exponent
                    exp = int(y[param_idx])

                    #remove parameter for lookup
                    _y = list(y)
                    _y[param_idx]=int(0)

                    exps[param_idx]=exp
                    A[ x , v[ETuple(_y)] ] += r(PolyDict({ETuple(exps):poly_dict[y]},\
                                                         force_etuples=False,force_int_exponents=False))

                else:
                    A[ x , v[y] ] += r(poly_dict[y])

        return ( A , MonomialVector( m ) )

    def poly_sort(self,m,T=None):
        """
        Sorts monomial list

        If the term order is 'lex' sorting is much faster than any other
        term order, as it is the nativ Python sorting algorithm.

        INPUT:
            m -- monomial list
            T -- term order
        """


        if isinstance(T,list) or isinstance(T,tuple):
            if len(m) != len(T):
                raise ArithmeticError, "Monomial order length does not match monomial list length"
            if set(m).difference(T):
                raise ArithmeticError, "Monomial order list does not match monomial list"
            return T

        if T==TermOrder("lex"):
            m.sort()
            m.reverse()
            return m

        r = m[0].parent()
        if T==None:
            T=r.term_order()
```

```
        singular.lib("general.lib"); #included for 'sort'
        self._ring._singular_().set_ring()
        singular_ideal = singular(str(m)[1:-1],type="ideal")
        s_list = singular_ideal.sort("\""+T.singular_str()+"\"")[1]
        m = [ poly.sage_poly(r) for poly in s_list ]
        m.reverse()
        return m

    def ideal(self):
        return self._ring.ideal(self._gens)

    def substitute(self,substitute):
        """
        susbtitutes the generators with substitute
        """
        kcache = {}
        m = substitute
        v = cf.setBaseDomain(self._ring.base_ring())
        if isinstance(m,dict):
            m = tuple([(cf.CF(var,v),cf.CF(val,v)) for var,val in m.iteritems()])

        gens2 = []
        for f in self._gens:
            poly=cf.CF(f,v)
            poly2 = poly(m)
            while poly2 != poly:
                poly  = poly2
                poly2 = poly(m)
            gens2.append(poly2._sage_(self._ring,kcache))

        self._gens = gens2


    def gen(self,i,Av=None):
        """
        Either returns the i-th polynomial from A*v counting from the
        bottom or -- if Av is not provided -- the i-th polynomial in
        the polynomial list of this system.

        INPUT:
           Av -- (coefficent matrix,monomial vector) tuple
           i  -- offset from bottom of polynomial
        """
        if Av==None:
            return self._gens[int(i)]
        else:
            A,v = Av

        poly_dict = dict()
        zero = self._base_ring(0)
        for col in range(A.ncols()):
            if A[A.nrows()-i-1, col] != zero:
                poly_dict[ v[col].element().dict().keys()[0] ] = A[A.nrows()-i-1, col]

        return self._ring(PolyDict(poly_dict, force_etuples=False, force_int_exponents=False))

    def minimal_ring(self,gens,T='lex'):
        """
        Given an iterable object which contains polynomials this
        method returns a minimal ring (only containing those variables
        which form the polynomials) and the polynomials coerced to
        that ring.

        """
        R = gens[0].parent()
        k = R.base_ring()

        o_vars = R.gens()
        n_vars = list(self.variables(gens))

        #preserve ordering
        n_vars = [str(v) for v in o_vars if v in n_vars]

        R2 = MPolynomialRing( k , len(n_vars), n_vars, T)
        gens = [ R2(str(f)) for f in gens ]
        return R2,gens

    def ring(self):
        return self._ring


class MQVariety:
    def __init__(self,F,A,v,flag=0):
        """
        Solves a multivariate polynomial system which is solvable by
        solving the univariate polynomials contained in this system.

        INPUT:
            F                -- this polynomial list will be used to check whether an intermediate
                                solution eventually solves the original polynomial system
            A                -- coefficient matrix of the system to be solved
            v                -- matching monomial vector to A, so that F = A*v
            offset           -- don't start at the bottom but at this offset
            partial_solution -- solution so far
            flag             -- if set, ignore zeros as root

        OUTPUT:
            Either the solution to the orig_polynomials polynomial
            list or None. The format of the solution ist the same as
            described in \\class{XL}::attack method.


        """

        self.univariate_filter = lambda h: h.is_univariate() and \
                                 not h.is_constant() and \
```

```
                                h.variable(0) == g.variable(0)

        self.F = F
        self.A = A
        self.v = v
        self.next_n = 0
        self.flag = flag
        self.max_roots = F._ring.base_ring().order()-1
        self._ring = F._ring
        self.base = self._ring.base_ring()

    def solve( self , offset=None, partial_solution=None):
        """
        Tries to actually solve the system F
        """

        F = self.F

        if hasattr(F,"terminate"):
            self.terminate = F.terminate

        A = self.A
        v = self.v

        if partial_solution==None:
            partial_solution = {}

        if offset==None:
            offset = 0

        if len(partial_solution) > 0:
            if self.terminate(partial_solution) or len(partial_solution) == F._ring.ngens():
                return partial_solution #this is it

            sm = list(partial_solution.iteritems())

        for pol in range(offset,self.A.nrows()):
            verbose("pol: %d"%pol,level=3)

            f = self.A.polynomial(v,pol)

            if len(partial_solution) > 0: #try to solve, what's solvable
                g = subst_poly(f,sm)
            else:
                g = f

            if not isinstance(g,MPolynomial) or g.is_constant():
                if g==0:
                    continue
                else:
                    return #won't work at all!

            if not g.is_univariate():
                continue #(?) proceed if non found

            if g == g.variable(0)** F._ring.base_ring().order() - g.variable(0):
                continue # ignore field equations

            verbose("found univariate %s"%(g),level=3)
            sys.stdout.flush()
            roots = [ root[0] for root in g.univariate_polynomial().roots() ]

            if roots == [] :
                # if one equation becomes unsolvable with a given
                # intermediate solution this solution will not work at
                # all, so:
                return

            for root in roots:
                if self.flag==1 and root==0:
                    continue

                local_solution = partial_solution.copy()
                local_solution[g.variable(0)] = g.parent()(root)

                self.next_n = v.find_next_n(local_solution,self.next_n)
                pol    = A.find_next_m( pol, self.next_n) #skip useless calculations
                ret = self.solve( pol, local_solution ) # and try it

                if ret!=None and ret!={}:
                    return ret #pass through solutions

        return partial_solution

    def solve_univariate(self,partial_solution=None,offset=0):
        """
        """
        if partial_solution==None:
            partial_solution = {}
        pols = self.A*self.v
        uni_pols = [ f for f in pols if f.is_univariate() ]
        for poly in uni_pols:
            verbose("found univariate %s"%(poly),level=3)
            if poly!=0:
                roots = poly.univariate_polynomial().roots()
                if len(roots)>self.max_roots or len(roots)==0:
                    continue
                else:
                    partial_solution[poly.variable(0)]=roots[0][0]
        if partial_solution != {}:
            return partial_solution

    def univariate_polynomials(self,ignored=None,ignored2=0):
        """
        """
        pols = self.A*self.v
        return [ f for f in pols if f.is_univariate() ]
```

```
##
## Functional
##


def coeff_matrix_w_param(F,param, T=None):
    """
    Experimental coeff_matrix_w_param
    """
    singular._start()
    singular.LIB("linalg.lib")
    singular.LIB("control.lib")
    singular.LIB(os.getcwd()+"/mq.lib")

    nvars = int(F._ring.ngens()-1)
    ngens = F.ngens()

    r = F._ring
    r._singular_()

    pols    = singular(str(F._gens)[1:-1],type="ideal")

    vars = singular(str(F._ring.gens()[:-1])[1:-1],type="ideal")

    A,v = singular.coeff_matrix_w_param(pols.name(), vars.name(), str(param))
    return A._sage_(r),v


def solves_problem(polynomials, solution):
    """
    Checks wether a given solution solves the system of multivariate
    polynomials.
    """

    if len(polynomials)==0:
        return True

    ring = polynomials[0].parent()

    for f in polynomials:
        if not subst_poly(f,solution) == 0:
            return False
    return True

def random_problem(k,m,n,hom=False,prob=1.0,term_order="lex"):
    """
    INPUT:
        k           -- base field
        m           -- number of equations
        n           -- number of variables
        hom         -- if True all non constant terms are quadratic
        prob        -- controls the density of the system, if 1.0 (default)
                       every equation contains all possible monomials, the
                       lower the value the lower the chance that a monomial
                       will be included
        term_order -- string which represents the term order of the ring to
                       be created, must be understood by MPolynomialRing
    OUTPUT:
        Returns a tuple where the first element is a MQ and
        the second one is the solution to this system.

    """
    r = MPolynomialRing(k,n,'x',term_order)

    useNTL = False
    usePoly = False
    deg  = 2

    if k.characteristic()==2 and k.order()!=2:
        useNTL = True
        ntl.GF2E_modulus(k)

    solution = {}
    cache = {}
    from sage.rings.polynomial_ring import PolynomialRing_field
    if isinstance(k,PolynomialRing_field):
        for e in r.gens():
            solution[e] = k.random_element(deg)
        usePoly = True
    else:
        for e in r.gens():
            solution[e] = k.random_element()
    if hom:
        ngens = range(r.ngens())
    else:
        ngens = range(r.ngens()+1)

    system = []
    i=0
    while i < m:
        d = {}
        for var1 in ngens:
            for var2 in ngens:
                if random() <= prob:
                    exponents = [int(0),]*r.ngens()
                    try:
                        exponents[var1]+=int(1)
                        exponents[var2]+=int(1)
                    except IndexError:
                        pass
                    if useNTL:
                        d[tuple(exponents)]=ntl.GF2E_random()._sage_(k,cache)
                    elif usePoly:
                        d[tuple(exponents)]= k.random_element(deg)
                    else:
```

```
                        d[tuple(exponents)]= k.random_element()
        f = r(PolyDict(d))
        f = f - subst_poly(f,solution)
        if not f.is_constant():
            system.append(f)
            i+=1

    sys = MQ(r,[system])
    return (sys, solution)
```

Listing A.6: Singular Functions for MQ

```
// Singular-library.
// AUHTOR: Martin Albrecht <malb@informatik.uni-bremen.de>

LIB "shared.lib"
LIB "matrix.lib";

proc coeff_matrix_w_param(pols, vars, poly param)
{

  list monomials;

  matrix tmp;

  poly parameter_mask = 1;

  // construct mask of monomials in x_i without parameter
  for(int i=1 ; i<=size(vars); i=i+1) {
    if(vars[i]!=param) {
      parameter_mask = parameter_mask * vars[i];
    }
  }
  // extract all monomials in x_i (without parameter)
  for(int i=1 ; i<=size(pols); i++) {
    tmp = coef(pols[i],parameter_mask);
    tmp = subst(tmp,param,1);
    for(int j=1; j<=ncols(tmp); j++) {
      monomials = insert(monomials, tmp[1,j]);
    }
  }
  // return coeff matrix
  return(coeff_matrix(pols,unique(monomials), parameter_mask ));
}

//
// AUTHOR: Michael Brickenstein <brickenstein@googlemail.com>
//
// INPUT:
//      i -- ideal in F_{2^n}[x_0,...,x_n-1]
//
// OUTPUT:
//      ideal in F_{2}[x_0,...,x_n+1]
//
// EXAMPLE:
//      ring r=(8,a),(w,x,y,z),lp;
//      ideal i=w2-ax,x-a,y+x,y-a,z-y-1;
//      minpoly;
//      def res_ring=toF2(i);
//      setring res_ring;
//      result;
//
// WARNING:
//      until ringlist is fixed, we assume, that the param is called a
proc toF2(ideal i)
{

  def old_ring=basering;
  int nvars_old=nvars(old_ring);

  string myminpoly_str=string(minpoly);
  string i_str=string(i);
  execute("ring helper=2,("+parstr(old_ring)+","+varstr(old_ring)+"),lp;");

  execute("poly myminpoly="+myminpoly_str+";");

  execute("ideal i="+ i_str+";");

  //now have minpoly and i in var format
  int r=deg(myminpoly);

  ring intermediate_ring=2,x(1..(nvars_old*r+1)),dp;
  poly im_of_param=var(1);
  int index, sum_index;
  ideal map_ideal;
  int var_index=1;
  poly sum;
  map_ideal[1]=var(1);//the param
  for(index=1;index<=nvars_old;index++){
    sum=0;

    for(sum_index=0;sum_index<r;sum_index++){
        var_index++;
        sum=sum+im_of_param^sum_index*var(var_index);
    }
    map_ideal[index+1]=sum;
  }

  //print(map_ideal);
  map m=helper,map_ideal;
  ideal i=m(i);
  poly myminpoly=m(myminpoly);
  //myminpoly;
```

```
  i=reduce(i,std(myminpoly));
  ideal result;
  matrix cm;
  for(index=1;index<=size(i);index++){
    cm=coef(i[index],var(1));
    result=result+ideal(submat(cm,2,1..ncols(cm)));
  }

  //eliminate parameter, change order to lp
  ring result_ring=2,(x(1..(nvars_old*r))),lp;
  ideal map_ideal;
  for(index=1;index<=nvars_old*r;index++){
  map_ideal[index+1]=var(index);
  }
  map m=intermediate_ring, map_ideal;
  ideal result=m(result);
  export(result);
  return(result_ring);
}
```

A.3 CTC

Listing A.7: CTC

```
"""

Implementation of the Courtois Toy Cipher

AUTHOR: Martin Albrecht <malb@informatik.uni-bremen.de>

"""

#import sage.libs.linbox.all as linbox
from sage.rings.integer_ring import ZZ
from sage.rings.finite_field_givaro import FiniteField_givaro
from sage.rings.multi_polynomial_ring import MPolynomialRing

from mq import *
from misc import *
#from polyf2_0 import *

class CTC:
    """

    Any CTC object may construct several CTC ideals for a given B and
    Nr combination.

    EXAMPLE:
        sage: ctc = CTC(B=1,Nr=1,qring=False)
        sage: R = ctc.ring_factory()

    """
    def __init__(self,B=1,Nr=1, qring=False):
        """
        """
        self.s = 3
        self.B = B
        self.Bs = int(self.B*self.s)
        self.Nr = Nr
        self.k = GF(2)
        self.qring = qring

    def ring_factory(self,pc=False,term_order="degrevlex"):
        """

        Constructs a ring which holds all variables of CTC
        instance. The variable ordering is as follows:

           X_0,i < ... < X_n,i < Y_0,i < ... < Y_n,i < Z_0,i < ... < Z_n,i < K_n,i < ... < K_0,i

        INPUT:
            pc      -- if True the plaintext and the ciphertext are viewed as variables
                       not as constants. (default: False)

        CREATES ATTRIBUTES:
            self.X -- 2D array of X_{ij} variables
            self.Y -- 2D array of Y_{ij} variables
            self.Y -- 2D array of X_{ij} variables
            self.K -- 2D array of K_{ij} variables
            self.ring -- PolynomialRing

        """
        Bs = self.Bs
        ns = 3 * self.Nr * self.Bs # #(X_ij,Y_ij,Z_ij)
        nk =  (self.Nr+1)*self.Bs  # #K_ij
        Nr = self.Nr


        self.X,self.Y,self.Z,self.K = [],[],[],[]

        if not pc:
            self.Z += [[]] #we may also denote the plaintext by Z_0
        self.X += [[]]
        self.Y += [[]]

        n_X = self.Nr
        n_Y = self.Nr
```

```
        n_Z = self.Nr
        n_K = (self.Nr+1)
        if pc:
            n_Z += 1
            n_X += 1
        names =  []

        offset = 0

        start = lambda i: int((i+offset  )*Bs)
        end   = lambda i: int((i+offset+1)*Bs)

        for i in range(n_X):
                self.X += [range(start(i),end(i))]
                names += ["X%03d%03d"%(i+1,j) for j in range(Bs)] #X_i starts at 1

        if not pc:
            self.X += []

        offset = n_X

        for i in range(n_Y):
            self.Y += [range(start(i),end(i))]
            names += ["Y%03d%03d"%(i+1,j) for j in range(Bs)] #Y_i starts at 1

        offset = n_X+n_Y

        for i in range(n_Z):
            self.Z += [range(start(i),end(i))]
            if pc:
                names += ["Z%03d%03d"%(i,j) for j in range(Bs)]
            else:
                names += ["Z%03d%03d"%(i+1,j) for j in range(Bs)]

        offset = n_X+n_Y+n_Z

        for i in reversed(range(n_K)): #K000 eliminated last
            self.K += [range(start(i),end(i))]
            names += ["K%03d%03d"%(i,j) for j in range(Bs)]

        if pc:
            if self.qring:
                self.ring = MPolynomialRingGF2(ns + nk + 2*Bs, names, order=term_order)
            else:
                self.ring = MPolynomialRing(self.k, names, ns + nk + 2*Bs ,order=term_order)
        else:
            if self.qring:
                self.ring = MPolynomialRingGF2(ns+nk , names, order=term_order)
            else:
                self.ring = MPolynomialRing(self.k, ns+nk, names, order=term_order)

        # immutable -> idx -> var
        for nr in range(Nr+1):
            self.K[nr] = tuple(map(self.ring.gen,self.K[nr]))
            self.X[nr] = tuple(map(self.ring.gen,self.X[nr]))
            self.Y[nr] = tuple(map(self.ring.gen,self.Y[nr]))
            self.Z[nr] = tuple(map(self.ring.gen,self.Z[nr]))

        return self.ring

    def ring_factory2(self,reverse=False, term_order="degrevlex"):
        """
        Constructs a ring which holds all variables of CTC
        instance. The variable ordering is as follows:

           K_n,i > Z_n,i > Y_n,i > X_n,i > ... > K_1,i > Z_1,i > Y_1,i > X_1,i > K_0,i

        where {X,Y,Z,K}_n,i+1 > {X,Y,Z,K}_n,i.

        If reverse is True this variable ordering is reversed.

        INPUT:
            term_order -- (default: 'degrevlex')

        CREATES ATTRIBUTES:
            self.X -- 2D array of X_{ij} variables
            self.Y -- 2D array of Y_{ij} variables
            self.Y -- 2D array of X_{ij} variables
            self.K -- 2D array of K_{ij} variables
            self.ring -- PolynomialRing

        """
        self.ring_factory(term_order=term_order)
        Nr = self.Nr
        Bs = self.Bs

        var_order = []

        for nr in reversed(range(1,self.Nr+1)):
            var_order += list(reversed(self.K[nr]))
            var_order += list(reversed(self.Z[nr]))
            var_order += list(reversed(self.Y[nr]))
            var_order += list(reversed(self.X[nr]))
        var_order += list(reversed(self.K[0]))

        if reverse:
            var_order = list(reversed(var_order))

        R = MPolynomialRing(self.k,len(var_order),[str(e) for e in var_order], order=term_order)

        self.K[0] = tuple([ R(str(self.K[0][i])) for i in range(Bs) ])
        for nr in range(1,Nr+1):
            self.K[nr] = tuple([ R(str(self.K[nr][i])) for i in range(Bs) ])
            self.X[nr] = tuple([ R(str(self.X[nr][i])) for i in range(Bs) ])
            self.Y[nr] = tuple([ R(str(self.Y[nr][i])) for i in range(Bs) ])
            self.Z[nr] = tuple([ R(str(self.Z[nr][i])) for i in range(Bs) ])
```

```
        self.ring = R
        return self.ring

    def MQ_factory(self,R=None,p=None,k=None):
        """
        Returns an instance of MQ in R.
        """

        if R is None:
            if p is None and k is None:
                R = self.ring_factory(pc=True)
            else:
                R = self.ring_factory()

        if p: # we have a plaintext
            self.Z[0]    = tuple(p)

        if p and k: # we have both plaintext and ciphertext
            self.X += [self.encrypt(p,k)]

        Bs = self.Bs
        s = self.s
        self.ring = R

        # s-boxes
        sbox = [tuple()] + [ sum([ self.Sbox_factory(self.X[i][j:j+s],self.Y[i][j:j+s])
                                for j in range(0,self.Bs,s) ],[])
                            for i in range(1,self.Nr+1) ]
        # diffusion layer
        lin =  [tuple()] + [ self.D___factory(self.Z[i],self.Y[i]) \
                                for i in range(1,self.Nr+1) ]

        # key addition equations
        add = [ self.Add_factory(self.X[int(i+1)],self.Z[int(i)],self.K[int(i)]) \
                for i in range(self.Nr+1) ]

        # key schedule equations
        key = [tuple()] + [ self.Key_factory(self.K[0],self.K[i],i) \
                               for i in range(1,self.Nr+1) ]

        add[0] = tuple(add[0])
        ctc_rounds = [add[0]]

        for i in range(1,self.Nr+1):
            rnd = []
            rnd += sbox[i]
            rnd += lin[i]
            rnd += key[i]
            rnd += add[i]
            ctc_rounds.append(tuple(rnd))
            sbox[i] = tuple(sbox[i])
            lin[i] = tuple(lin[i])
            add[i] = tuple(add[i])
            key[i] = tuple(key[i])

        F = MQ(R, tuple(ctc_rounds))
        F.sbox = tuple(sbox)
        F.lin  = tuple(lin)
        F.add  = tuple(add)
        F.key  = tuple(key)
        F.terminate = ctc_terminate

        return F

    def field_equations(self):

        return [ var**2 + var for var in self.ring.gens()]

    def Sbox_factory(self,x,y):
        """
        """
        x1,x2,x3 = x[0],x[1],x[2]
        y1,y2,y3 = y[0],y[1],y[2]

        one = self.ring(1)

        l = [ x1*x2 + y1 + x3 + x2 + x1 + one,
              x1*x3 + y2 + x2 + one,
              x1*y1 + y2 + x2 + one,
              x1*y2 + y2 + y1 + x3,
              x2*x3 + y3 + y2 + y1 + x2 + x1 + one,
              x2*y1 + y3 + y2 + y1 + x2 + x1 + one,
              x2*y2 + x1*y3 + x1,
              x2*y3 + x1*y3 + y1 + x3 + x2 + one,
              x3*y1 + x1*y3 + y3 + y1,
              x3*y2 + y3 + y1 + x3 + x1,
              x3*y3 + x1*y3 + y2 + x2 + x1 + one,
              y1*y2 + y3 + x1,
              y1*y3 + y3 + y2 + x2 + x1 + one,
              y2*y3 + y3 + y2 + y1 + x3 + x1
              ]

        return l

    def D___factory(self,z,y):
        """
        The diffusion part D of the cipher is defined as follows:

         Z_{i,(257 mod Bs)} = Y_{i,0} for all i = 1 ... Nr
         Z_{i,j*1987+257 mod Bs} = Y_{i,j} + Y_{i,j+137 mod Bs} for j != 0 and all i

        """
        Bs = self.Bs

        l = [ z[ (j*1987+257) % Bs ] + y[ j ]  + y[ (j+137) % Bs ] for j in range(1,Bs)]
        l += [ z[ 257 % Bs ] + y[ 0 ] ]
        return l
```

```
    def Add_factory(self,x,z,k):
        """
        With all these notations, the linear equations from the key
        schedule are as follows:

          X_{i+1,j} = Z_{i,j} + K_{i,j} for all i = 0 ... Nr

        """
        return [ self.ring(x[j]) + self.ring(z[j]) + self.ring(k[j]) for j in range(self.Bs)]

    def Key_factory(self,k0,k,i):
        """
        There is no S-Boxes in the key schedule and the derived key in
        round i, K_i is obtained from he secret key K_0, by a very
        simple permutation of wires:

         K_{i,j} = K_{0,(j+i mod Bs)}

        """

        Bs = self.Bs

        return [ k[j] + k0[ (j+i) % Bs] for j in range(Bs) ]

    def subst_factory(self, only_linear=True):
        subst = {}

        Bs = self.Bs
        Nr = int(self.Nr)

        if not only_linear:
            for i in range(1,self.Nr+1):
                for j in range(0,Bs,3):
                    X1 = self.X[i][j+0]
                    X2 = self.X[i][j+1]
                    X3 = self.X[i][j+2]
                    Y1 = self.Y[i][j+0]
                    Y2 = self.Y[i][j+1]
                    Y3 = self.Y[i][j+2]
                    subst[Y1] =X1*X2 + X3 + X2 + X1 + 1
                    subst[Y2] =X1*X3 + X2 + 1
                    subst[Y3] =X2*X3 + subst[Y2] + subst[Y1] + X2 + X1 + 1

        # replace K_{i,j} by K_{0,j}
        for i in range(1,self.Nr+1):
            for j in range(Bs):
                subst[self.K[i][j]] = self.K[0][ (j+i) % Bs ]

        # replace Z_{i,j} by Y_{i,j}
        for i in range(1,self.Nr+1):
             subst[self.Z[i][257 % Bs]] =  self.Y[i][0]
             for j in range(1,Bs):
                 subst[self.Z[i][(j*1987+257) % Bs]]  = self.Y[i][j] + self.Y[i][(j+137) % Bs ]

        # replace X_{i+1,j} by Z_{i,j} + K_{i,j}
        for i in range(0,self.Nr):
             for j in range(Bs):
                 subst[self.X[i+1][j]] = self.Z[i][j] + self.K[i][j]

        return subst

    def MQgb_factory(self,R=None,p=None,k=None):
        """

        """

        if R is None:
            if p is None and k is None:
                R = self.ring_factory(pc=True)
            else:
                R = self.ring_factory()

        F = self.MQ_factory(R,p,k)


        sbox = []
        lin = []
        key = []
        add = []

        Bs = self.Bs
        for nr in range(self.Nr+1):
            if nr!=0:
                sbox.append([])
                for i in range(len(F.sbox[nr])/14):
                    f1 = F.sbox[nr][14*i+3]  - R("Y%03d%03d"%(nr,3*i+0)) + R("Y%03d%03d^2"%(nr,3*i+0))
                    f2 = F.sbox[nr][14*i+1]  - R("Y%03d%03d"%(nr,3*i+1)) + R("Y%03d%03d^2"%(nr,3*i+1))
                    f3 = F.sbox[nr][14*i+11] - R("Y%03d%03d"%(nr,3*i+2)) + R("Y%03d%03d^2"%(nr,3*i+2))
                    sbox[-1] += [f1,f2,f3]
                sbox[-1] = tuple(sbox[-1])
            else:
                sbox.append(tuple())
            lin.append(F.lin[nr])
            key.append(F.key[nr])
            if nr!=self.Nr:
                add.append(F.add[nr])
            else:
                #K_{i,j} = K_{0,(j+i mod Bs)}
                add.append(tuple([ F.add[nr][j] - R("K%03d%03d"%(nr,j)) + R("K000%03d^2"%((nr+j)%Bs)) \
                                   for j in range(len(F.add[nr])) ]))

        ctc_rounds = [add[0]]

        for i in range(1,self.Nr+1):
            rnd = []
            rnd += sbox[i]
```

```
            rnd += lin[i]
            rnd += key[i]
            rnd += add[i]
            ctc_rounds.append(rnd)

        F = MQ(R, tuple(ctc_rounds))
        F.sbox = sbox
        F.lin  = lin
        F.add  = add
        F.key  = key
        F.terminate = ctc_terminate

        return F

    #
    # encryption methods
    #

    def encrypt(self,p,k):
        """
        """

        def Kfactory(i):
            K = [0]*self.Bs
            for j in range(self.Bs):
                K[j] = k[int((j+i))%self.Bs]

            return K

        Z = [ self.k(e) for e in p ]
        for i in range(self.Nr):
            K = Kfactory(i)
            X = self.add( Z, K )
            Y = self.sbox( X )
            Z = self.d(Y)

        K = Kfactory(self.Nr)
        X = [ Z[j] + K[j] for j in range(self.Bs) ]

        return X

    def add(self,x,y):
        return [ x[j] + y[j] for j in range(self.Bs) ]

    def sbox(self,X):
        k = self.k
        sbox = dict(zip(range(8),[7,6,0,4,2,5,1,3]))
        s = self.s

        def single_sbox(v):
            v = list(ZZ(sbox[4*int(v[2])+2*int(v[1])+int(v[0])]).binary())
            if(len(v)<3):
                v = [0]*(3-len(v))+v
            return [k(v[2]),k(v[1]),k(v[0])]

        return sum([ single_sbox(X[j:j+s]) for j in range(0,self.Bs,s) ],[])

    def d(self,y):
        Bs = self.Bs

        z = [0]*self.Bs

        z[ 257 % Bs ] = y[ 0 ]

        for j in range(1,Bs):
            z[ (j*1987+257) % Bs ] = y[ j ]  + y[ (j+137) % Bs ]

        return z

def ctc_terminate(solution):
    for elem in solution:
        if not str(elem).startswith('K000'):
            return True
    return False


def ctc_MQ(Nr=1,B=1,subst=0, term_order="degrevlex",qring=False, \
           variable_order=0, mqgb=False, plain=None, key=None):
    """
    Returns a CTC MQ problem with random plaintext and key (if those
    are not provided) for the given configuration.

    INPUT:
        Nr -- number of rounds (default: 1)
        B  -- number of 3-bit blocks (default: 1)
        subst -- how to substitute variables (default: 0)
                 0 - no substitution
                 1 - linear equations are used for substitution
                 2 - all equations are used for substitution
        term_order -- term ordering of the ring (default: degrevlex)
        qring -- use quotient ring implementation (default: False)
        variable_order -- controls the ordering of the variables (default: 0)
                          0 -- ctc.ring_factory is called
                          1 -- ctc.ring_factory2 is called
                          2 -- ctc.ring_factory2(reverse=True) is called
        mqgb -- construct a Groebner basis for ctc ideals
        plain -- plaintext
        key -- key
    """
    ctc=CTC(Nr=Nr,B=B,qring=qring)

    if variable_order==1:
        R = ctc.ring_factory2(term_order=term_order,reverse=False)
    elif variable_order==2:
        R = ctc.ring_factory2(term_order=term_order,reverse=True)
    else:
        R = ctc.ring_factory(term_order=term_order)
```

```
    # random solution
    k = ctc.ring.base_ring()
    if plain is None:
        plain = [k.random_element() for _ in range(B*3)]
    else:
        plain = [k(e) for e in plain]
    if key is None:
        key = [k.random_element() for _ in range(B*3)]
    else:
        key = [k(e) for e in key]

    if mqgb:
        F = ctc.MQgb_factory(R, plain, key)
    else:
        F = ctc.MQ_factory(R, plain, key)

    if subst!=0:
        if subst==1:
            s = ctc.subst_factory(True)
        elif subst==2:
            s = ctc.subst_factory(only_linear=False)
        F.substitute(s)
        r,g = F.minimal_ring(F._gens, term_order)
        g = [e for e in g if e!=0]
        F = MQ(r,g)
        F.terminate = ctc_terminate
    return F, dict(zip([ctc.ring("K000%03d"%i) for i in range(B*3)], key))

def strip_sboxes(F):
    sbox = [tuple()]
    for i in range(1, len(F.sbox)):
        ret = []
        for j in range(len(F.sbox[i])/14):
            ret.append(F.sbox[i][j*14+0])
            ret.append(F.sbox[i][j*14+1])
            ret.append(F.sbox[i][j*14+4])
        sbox.append(tuple(ret))

    rounds = []
    for i in range(len(sbox)):
        rounds.append(tuple( list(sbox[i])+list(F.lin[i])+list(F.key[i])+list(F.add[i])))

    F.round = tuple(rounds)
    F.sbox = sbox
    F._gens = list(flatten(rounds))
    return F


def block_order(B=1,Nr=1):
    """
    Constructs a blockorder/product order string for singular
    """
    if Nr==1:
        return "dp"

    bo = []
    for nr in range(Nr-1):
        bo.append("dp(" + str(B*3*4) + ")")
    return "(dp("+str(B*3*4+B*3)+")," + ",".join(bo) + ")"

def block_order_rev(B=1,Nr=1):
    """
    Constructs a blockorder/product order string for singular
    """
    if Nr==1:
        return "dp"

    bo = []
    for nr in range(Nr-1):
        bo.append("dp(" + str(B*3*4) + ")")
    return "(" +  ",".join(bo) + ",dp("+str(B*3*4+B*3)+"))"
```

A.4 F4

Listing A.8: F4

```
#!/usr/bin/env sage-python
#
# -*- Mode: Python -*-
# # vi:si:et:sw=4:sts=4:ts=4
#

"""
 F4

 AUTHOR: Martin Albrecht <malb@informatik.uni-bremen.de>

"""

from mq import *
from algebraicattack import *
from sage.rings.all import *
import sage.misc.misc as misc
from sage.rings.ideal import is_Ideal

class F4_orig(AlgebraicAttack):
    """
    Original F4 as described by Faugere
```

```
    """
    def __init__(self):
        pass

    def example_Faugere(self,R=None):

        if R == None:
            R = PolynomialRing(GF(31991),4,'abcd',order="degrevlex")
        x0,x1,x2,x3 = R.gens()

        f1 = x0*x1*x2*x3 - R(1)
        f2 = x0*x1*x2 + x0*x1*x3 + x0*x2*x3 + x1*x2*x3
        f3 = x0*x1 + x1*x2 + x0*x3 + x2*x3
        f4 = x0 + x1 + x2 + x3

        F = [f1,f2,f3,f4]

        return MQ(R,F)

    def __call__(self,F):
        return self.groebner(F)

    def groebner(self,F,sel=None):
        if is_Ideal(F):
            F = MQ(F._ring(),F.gens())

        self.ring = F[0].parent()
        #self.ring._singular_()
        self.rr_bases = []
        G = list(F)
        F0p = F
        d = 0
        P = set([self.pair(f,g) for f in G for g in G if f<g ] )

        if sel==None:
            sel = self.normal_strategy

        while P != set():
            d = d+1
            Pd = sel(P)
            P = P.difference(Pd)
            Ld = set(self.left(Pd)).union(set(self.right(Pd)))
            Fdp = self.reduction(Ld,G)
            for h in Fdp:
                P = P.union(set([self.pair(h,g) for g in G ]))
                G.append(h)
            sys.stdout.flush()
        return G

    def reduction(self,L,G):
        F = self.symbolic_preprocessing(L,G)
        Ft = self.row_echelon(F)
        LMF = LM(F)
        Ftp = set([f for f in Ft if f.lm() not in LMF])
        return list(Ftp)

    def symbolic_preprocessing(self,L,G):
        """
        """
        G = G
        F = set([t*f for (t,f) in L ])
        Done = LM(F)
        M = set([m  for f in F for m in f.monomials()])
        R = self.ring
        while M != Done:
            m = M.difference(Done).pop()
            Done.add(m)
            t,g = self.ring._m_reduce_mod(m,G)
            if t!=R(0): F.add(t*g)
            M = set([m  for f in F for m in f.monomials()])
        return F

    def pair(self,f,g):
        lcm = self.ring._m_lcm(f.lm(),g.lm())
        # it seems better speed-wise to calculate those on the fly
        #tf = LCMdLM(lcm,f.lm())
        #tg = LCMdLM(lcm,g.lm())
        return (lcm,f,g)

    def left(self,p):
        if isinstance(p,(list,set,tuple)):
            s = set()
            for f in p:
                s.add((self.ring._m_lcmfg_div_f(f[0],f[1].lm()),f[1]))
            return s
        else:
            return (self.ring._m_lcmfg_div_f(p[0],p[1].lm()),p[1])

    def right(self,p):
        if isinstance(p,(list,set,tuple)):
            s = set()
            for f in p:
                s.add((self.ring._m_lcmfg_div_f(f[0],f[2].lm()),f[2]))
            return s
        else:
            return (self.ring._m_lcmfg_div_f(p[0],p[2].lm()),p[2])

    def row_echelon(self, F):
        """
        """

        F2 = MQ(self.ring,F)
        A,v=F2.coeff_matrix()
        A.echelonize()
        F = A*v
        return F
```

```
    # Strategies

    def normal_strategy(self,P):
        """
        The normal selection strategy

        INPUT:
            P -- a list of critical pairs

        OUTPUT:
            a sublist of P

        """
        d = min(set([ lcm.total_degree() for (lcm,fi,fj) in P ]))
        return set([ (lcm,fi,fj) for (lcm,fi,fj) in P if lcm.total_degree()==d])

    def update_GM(self, G, P, h):
        """
        Gebauer Moeller Installation as written by Toon Segern

        INPUT:
            G -- an intermediate Groebner basis
            P -- a list of critical pairs
            h -- a polynomial

        OUTPUT:
            an intermediate Groebner basis, a list of critical pairs

        WARNING: untested
        """
        # Initialization
        R = self.ring
        LCM = lambda x,y: R._m_lcm(x,y)
        hlm = h.lm()


        # Rule B_h(i,j)
        D = set()
        for p in P:
            lcmp0p1,p0,p1 = p
            if LCM(p0.lm(), p1.lm()) != LCM( LCM( p0.lm(),p1.lm()), hlm) \
                    or LCM(p0.lm(),  hlm) == LCM(p0.lm(),p1.lm()) \
                    or LCM(p0.lm(),  p1.lm()) == LCM(p1.lm(), hlm) \
                    or LCM(p0.lm(),  hlm) == LCM(p1.lm(), hlm):
                D.add(p)

        # Create the set P1
        P1 = set([ self.pair(g,h) for g in G])

        # Rule M(i,h)
        for g in G:
            res =  exists(P1, \
                          lambda (lcmp0p1,p0,p1): R._m_is_reducible_by( LCM(g.lm(),hlm) , \
                                                                        LCM(p0.lm(),p1.lm()) ) \
                          and LCM(g.lm(),hlm) != LCM(p0.lm(),p1.lm()) )
            if res[0]:
                P1.remove(self.pair(g,h))

        # Modified criterion F
        # Create the set of all LCM monomials corresponding to the pairs in D1
        tauset = set([lcmp0p1 for lcmp0p1,p0,p1 in P1])
        P1new = set()

        while len(tauset):
            # Treat every subset of pairs of P1 with LCM equal to tau separately
            tau = tauset.pop()
            subsetP1tau = set([(lcmp0p1,p0,p1) for (lcmp0p1,p0,p1) in P1 if lcmp0p1 == tau])
            res = exists(subsetP1tau, lambda (lcmp0p1,p0,p1): p0.lm()*p1.lm() == lcmp0p1)
            if res[0]:
                subsetP1tau = set([res[1]])
            else:
                subsetP1tau = set([list(subsetP1tau)[0]])
            P1new = P1new.union( subsetP1tau )

        P1new = set([ (lcmp0p1,p0,p1) for (lcmp0p1,p0,p1) in P1new if not R._m_pairwise_prime(p0.lm(), \
                                                                                               p1.lm())])
        G.append(h)
        return G, D.union( P1new )

    def update_buchbergerGF2(self, G, B, h):
        """
        Buchberger Criterion

        INPUT:
            G -- an intermediate Groebner basis
            B -- a list of critical pairs
            h -- a polynomial

        OUTPUT:
            an intermediate Groebner basis, a list of critical pairs

        WARNING: untested
        """
        R = self.ring

        hlm = h.lm()
        B_new = set()

        r1 = []
        r2 = []

        for gen in hlm.variables():
            r1.append(h*gen); r2.append(h)

        G = G + Reduce(r1,r2)
```

```
        for g in G:
            if R._m_pairwise_prime(g.lm(),hlm):
                continue

                        # Buchberger criterion2
            lcm = R._m_lcm(g.lm(),h)
            for j in G:
                if j==g:
                    break
                if R._m_is_reducible_by(lcm,j.lm()):
                    break
                if j != g:
                    continue
            B_new.add(self.pair(h,g))

        G.append(h)

        return G,B.union(B_new)

    def update_pairsGF2(self,G,B,h):
        """
        Following Becker, 'Groebner Bases', Springer 1993 as suggested
        by Faugere in his F4 paper. Also works in the quotient ring.

        INPUT:
            G -- an intermediate Groebner basis
            B -- a list of critical pairs
            h -- a polynomial

        OUTPUT:
            an intermediate Groebner basis, a list of critical pairs

        """

        R = self.ring

        hlm = h.lm()

        G_new = list()


        # if G is a set then C only contains unique elements
        C = list([self.pair(h,g) for g in G]) # 1.86
        D = list() # only adding elements of C, thus unique


        # Criterion F & M # 3.7
        while C!=list():
            (lcmhg1,h,g1) = C.pop()

            # will be removed in next loop
            if R._m_pairwise_prime(hlm,g1.lm()):
                D.append((lcmhg1,h,g1))
                continue

            found = 0
            for c in C:
                if R._m_is_reducible_by( lcmhg1, c[0] ):
                    found=1; break
            if found: continue

            found = 0
            for d in D:
                if R._m_is_reducible_by( lcmhg1, d[0] ):
                    found=1; break
            if found: continue

            D.append((lcmhg1,h,g1))


        E = list() #only adding elements of D, thus unique


        # Buchberger criterion 1 # 0.71
        for (lcmhg,h,g) in D:
            # if LM(h) and LM(g) are not disjoint
            if not R._m_pairwise_prime(hlm,g.lm()):
                E.append((lcmhg,h,g))


        B_new = set()

        # Criterion B_k # 2.52
        for (lcmg1g2,g1,g2) in B:
            if not self.ring._m_is_reducible_by( lcmg1g2, hlm ) or \
                   self.ring._m_lcm(g1.lm(), hlm) == lcmg1g2 or \
                   self.ring._m_lcm( hlm,g2.lm()) == lcmg1g2 :
                B_new.add((lcmg1g2,g1,g2))

        B_new = B_new.union(E)



        # j, {Si} prove this! 3.29
        r=[]
        for gen in hlm.variables():
                        # consider F -- always true
                        # consider M
            for f in G:
                if R._m_is_reducible_by(f.lm(),hlm):
                    r.append( (gen,f) )
                    break
        r.append((R(1),h))
        G_new = self.reduction(r,[],[],True)[0]


        for g in G: # 1.05
```

```
            if not R._m_is_reducible_by(g.lm(),hlm):
                G_new.append(g)
        G_new.append(h)

        return G_new,B_new

    def update_pairs(self,G,B,h):
        """
        Following Becker, 'Groebner Bases', Springer 1993 as suggested
        by Faugere in his F4 paper.

        INPUT:
            G -- an intermediate Groebner basis
            B -- a list of critical pairs
            h -- a polynomial

        OUTPUT:
            an intermediate Groebner basis, a list of critical pairs

        """

        R = self.ring

        # if G is a set then C only contains unique elements
        C = [self.pair(h,g) for g in G]
        D = list() # only adding elements of C, thus unique

        # Criterion M

        while C!=list():
            (lcmhg1,h,g1) = C.pop()

            lcm_divides = lambda lcmhg2: R._m_is_reducible_by( lcmhg1, lcmhg2[0] )

            # if LM(h) and LM(g_1) are disjoint
            if R._m_pairwise_prime(h.lm(),g.lm()) or \
               (\
                    not misc.exists(C, lcm_divides )[0] \
                    and \
                    not misc.exists(D, lcm_divides )[0]\
                ):
                D.append((lcmhg1,h,g1))

        E = list() #only adding elements of D, thus unique

        # Criterion F

        while D != list():
            (lcmhg,h,g) = D.pop()
            # if LM(h) and LM(g) are not disjoint
            if not R._m_pairwise_prime(h.lm(),g.lm()):
                E.append((lcmhg,h,g))

        B_new = set()

        # Criterion B_k

        while B != set():
            lcmg1g2,g1,g2 = B.pop()
            if not self.ring._m_is_reducible_by( lcmg1g2, h.lm() ) or \
                    self.ring._m_lcm(g1.lm(), h.lm()) == lcmg1g2 or \
                    self.ring._m_lcm( h.lm(),g2.lm()) == lcmg1g2 :
                B_new.add((lcmg1g2,g1,g2))

        B_new = B_new.union(E)

        G_new = list()

        while G != list():
            g = G.pop()
            if not R._m_is_reducible_by(g.lm(),h.lm()):
                G_new.append(g)

        G_new.append(h)

        return G_new,B_new

    def update_simple(self,G, P, h):
        """
        Adding all critical pairs

        INPUT:
            G -- an intermediate Groebner basis
            B -- a list of critical pairs
            h -- a polynomial

        OUTPUT:
            an intermediate Groebner basis, a list of critical pairs

        """
        return G+[h],P.union([self.pair(g,h) for g in G])


class F4(F4_orig):
    """
    The improved F4 as described in Faugere's paper.
    """

    def __call__(self,F):
        if isinstance(F.ring(), MPolynomialRing_polydict):
            return self.groebner(F,Update=self.update_pairs)
        else: #this is risky
            return self.groebner(F,Update=self.update_pairsGF2)

    def attack(self, F, Sel=None, Update=None, protocol=False):
        """
        Computes a Groebner basis and tries to calculate the variety
```

```
        afterwards.

        INPUT:
            F      -- a finite subset of R[x]
            Sel    -- selection strategy
            Update -- update pairs to select critical pairs to compute

        OUTPUT:
            a (partial) solution that satisfies F
        """
        gb = self.groebner(F,Sel,Update,protocol)
        gb = MQ(F._ring,gb)
        A,v = gb.coeff_matrix()
        A.echelonize()
        solver = MQVariety(gb,A,v)
        ret = solver.solve()
        return ret

    def groebner(self, F, Sel=None, Update=None, protocol=False):
        """
        INPUT:
            F      -- a finite subset of R[x]
            Sel    -- selection strategy
            Update -- update pairs to select critical pairs to compute

        OUTPUT:
            G -- a Groebner Basis for F
        """
        self.protocol = protocol

        if is_Ideal(F):
            F = MQ(F._ring(),F.gens())

        # pretty looking code
        Left           = self.left
        Right          = self.right
        Reduction      = self.reduction
        first          = self.first
        if     Sel==None: Sel = self.normal_strategy
        if Update==None: Update = self.update_pairs

        self.ring = F[0].parent()
        self.ring._singular_().set_ring()
        self.term_order = self.ring.term_order()

        # We maintain a list of dictionaries which contain f.lm() => f
        # maps for the sets $F_j^~$ to allow O(1) lookups for this code:
        #"$F_j^~$ is the row echelon form of F_j w.r.t. < there exists a
        # (unique) $p \in F_j^~ such that LM(p) = LM(u*f)"
        self.Ftd = [[]]

        F = list(F) #
        Fd = dict()

        G = list()
        P = set()
        d = 0

        _rt = 0
        _ut = 0
        self._rt = 0.0

        _t = cputime()
        while F != list():
            f = first(F)
            F.remove(f)
            G,P = Update(G,P,f)
        verbose("Init time %f"%cputime(_t),level=1)
        sys.stdout.flush()

        while P != set():
            d = d+1
            Pd = Sel(P)
            if self.protocol:
                print "P",sorted(P)
                print "P%d"%d,sorted(Pd)
                print "G",sorted(G)
            P = P.difference(Pd)
            Ld = Left(Pd).union( Right(Pd) )
            if self.protocol:
                print "L%d"%d,sorted(Ld)
            _t = cputime()
            Fdp,Fd[d] = Reduction(Ld,G,Fd)
            _rt += cputime(_t)
            _t = cputime()
            for h in Fdp:
                G,P = Update(G,P,h)
            _ut += cputime(_t)
        verbose("Reduction time %f"%_rt,level=1)
        verbose("Update time %f"%_ut,level=1)
        return G

    def reduction(self,L,G,Fset,no_update=False):
        """
        INPUT:
            L -- a finite subset of M x R[x]
            G -- a finite subset of R[x]
            F -- (F_k)k=1,\dots,(d-1), where F_k is finite subset of R[x]

        OUTPUT:
            F^+,F
        """
        F = self.symbolic_preprocessing(L,G,Fset)
        if self.protocol:
            print " F", sorted(F)

        Ft = self.row_echelon(F)
```

```
        if self.protocol:
            print " Ft", sorted(Ft)

        LMF = LM(F)
        Ftp = list(set([f for f in Ft if f.lm() not in LMF]))

        if self.protocol:
            print " Ftp", sorted(Ftp)
        if not no_update:
            # maintain the f.lm()=>f dictionary
            self.Ftd.append( dict([(f.lm(),f) for f in Ft]) )
        return Ftp,F

    def symbolic_preprocessing(self,L,G,Fset):
        """
        INPUT:
            L -- a finite subset of M x R[x]
            G -- a finite subset of R[x
            F -- (F_k)k=1,\dots,(d-1), where F_k is finite subset of R[x]

        OUTPUT:
            a finite subset of R[x]
        """
        Simplify = self.simplify
        R = self.ring
        Mul = lambda (m,f): m*f


        F = set([Mul(Simplify(m,f,Fset)) for (m,f) in L])
        if self.protocol:
            print "  F",sorted(F)

        Done = LM(F)

        if self.protocol:
            print "  Done",sorted(Done)

        M = set([m for f in F for m in f.monomials()])

        if self.protocol:
            print "  T(F)",sorted(M)

        MdivDone = M.difference(Done)
        zero = R(0)
        G = tuple(G)

        while MdivDone != set():#M != Done
            #m = M.difference(Done).pop()
            m = MdivDone.pop()
            Done.add(m)
            t,g = self.ring._m_reduce_mod(m,G)
            if t!=zero:
                tg = Mul(Simplify(t,g,Fset))
                F.add(tg)
                # M = set([m for f in F for m in f.monomials()])
                for tgm in tg.monomials():
                    M.add(tgm)
                    if tgm not in Done:
                        MdivDone.add(tgm)
        return F

    def simplify(self,t,f,F):
        """
        INPUT:
            t -- \in M a monomial
            f -- \in R[x] a polynomial
            F -- (F_k)k=1,\dots,(d-1), where F_k is finite subset of R[x]

        OUTPUT:
            a non evaluated product, i.e. an element of M x R[x]
        """
        for u in sorted(self.ring._m_all_divisors(t),reverse=True):
            uf = u*f
            for j in F:
                if uf in F[j]:
                    # F~_j is the row echelon form of F_j w.r.t. <
                    # there exists a (unique) p \in F~_j such that LM(p) = LM(u*f)
                    p = self.Ftd[j][uf.lm()]
                    if u!=t:
                        return self.simplify(self.ring._m_lcmfg_div_f(t,u),p,F) #t/u
                    else:
                        return (self.ring(1),p)
        return (t,f)


    def first(self,G):
        """
        Returns the largest element of G.

        INPUT:
            G -- a finite subset of G

        OUTPUT:
            a polynomial \in G

        """
        mg = G[0]
        mm = mg.lm()
        for g in G:
            if g.lm() > mm:
                mm = g.lm()
                mg = g
        return mg

def LM(F):
    """
    """
```

```
    if isinstance(F,(list,set,tuple)):
        return set([f.lm() for f in F])
    else:
        return F.lm()

def Reduce(f,g):
    if f==[] or g==[]:
        return []
    R = f[0].parent()
    res = str(singular(f,"ideal").reduce(singular(g,"ideal")).string()).split(",")
    res = [R(res[i]) for i in range(len(res)) if res[i]!='0']
    return res
```

A.5 DR

Listing A.9: DR SAGE part

```
#!/usr/bin/env sage-python
#
# -*- Mode: Python -*-
# vi:si:et:sw=4:sts=4:ts=4
#

"""
 Dixon Resultants [DR]

 AUTHOR: Martin Albrecht <malb@informatik.uni-bremen.de>

 Dixon/RSC code is based on a Maple implementation of the
 KYS-Dixon matrix algorithm by Arthur D. Chtcherba <cherba@cs.panam.edu>

--
 [DR] X. Tang and Y. Feng; A New Efficient Algorithm for Solving Systems of
 Multivariate Polynomial Equations; 2005

"""


from mq import *
from misc import *
try:
    del range
except:
    pass

def singular_setup():
    singular._start()
    singular.LIB("linalg.lib")
    singular.LIB("control.lib")
    singular.LIB("rootsur.lib")
    singular.LIB(os.getcwd()+"/dr.lib")

class RSCError(Exception):
    pass

class DR:
    """
    This class implements the DR algorithm as described in [DR].

    Changes to the upstream version:
      * this version only features a partial solver
      * this version is guaranteed to terminate
    """

    def example_Tang_et_al(self):
        """
        Example as in [DR]
        """
        r = MPolynomialRing(GF(127),5,"x",order="lex")
        x1,x2,x3,x4,x5 = r.gens()

        l_1 = 9*x1 + 37*x3 + 17*x1*x2 + 120*x2*x3 + 18*x3*x5 + 58*x4**2 + 87
        l_2 = 46*x1 + 43*x3 + 117*x5 + 43*x1*x2 + 93*x1*x3 + 61*x3*x4 + 48
        l_3 = 32*x1*x2 + 54*x1*x4 + 56*x2*x3 + 93*x3*x5 + 60*x5**2 + 45
        l_4 = 124*x1 + 93*x1*x3 + 78*x1*x4 + 45*x1*x5 + 39*x2*x3 + 38*x2*x4 + 46
        l_5 = 27*x2 + 95*x2*x5 + 85*x3**2 + 74*x3*x4 + 46*x3*x5 + 77*x4*x5 + 66

        m = MQ(r,[l_1,l_2,l_3,l_4,l_5])

        return m

    def example_type_a(self,n,k=GF(127),order="lex"):
        """
        Type A examples as in [DR]
        """
        k = k
        r = MPolynomialRing(k,n,"x",order=order)
        x = r.gen
        l = []
        solution = {}
        for i in range(n):
            solution[x(i)]=k.random_element()
            l.append( x(i)+x(((i%n)+1)%n) * x((((i+1)%n)+1)%n) )

        for i in range(n):
            l[i] = l[i] - subst_poly(l[i],solution)

        return MQ(r,l),solution
```

```
def attack(self,A, step45=True):
    """
    Computes a solution vector for A

    INPUT:
        A -- an MQ problem
        step45 -- combine step 4 & 5 (default: True)

    OUTPUT:
        partial solutions of which at least one satisfy A.

    """
    singular_setup()

    ### step 1. Taking x_1 \dots x_{n-1} as variables and x_n as
    ### paramter, compute the Dixon matrix of A
    _time,_stime = cputime(),singular.cputime()

    M,v = self.dixon_matrix(A)

    verbose("M: (%s,%s)"%(M.nrows(),M.ncols()),level=1)
    verbose("step 1. dixon matrix time: %ss"%(cputime(_time)+singular.cputime(_stime)),level=2)
    sys.stdout.flush()

    ### step 2. Run subprogram RSC to check RSC Criteria and select
    ### rows and columns needed for contructing KSY Dixon Matrix

    _time,_stime = cputime(),singular.cputime()

    cols,rows = 0,0
    i = 0
    order = A.ring().base_ring().order()
    while (cols,rows) == (0,0) and i < order-1:
        cols,rows = self.rsc(M)
        i+=1

    if (cols,rows) == (0,0):
        #this is very likely over GF(2)
        AA,Av = A.coeff_matrix()
        AA.echelonize()
        solver = MQVariety(A,AA,Av)
        ret = solver.solve(partial_solution={self.parameter:self.p})
        return ret

    verbose("step 2. rsc time: %ss"%(cputime(_time)+singular.cputime(_stime)),level=3)
    sys.stdout.flush()


    ### step 3. Construct the KSY Dixon Matrix

    _time,_stime = cputime(),singular.cputime()
    KSY = singular.submat(M,rows,cols)
    verbose("M': (%s,%s)"%(KSY.nrows(),KSY.ncols()),level=1)
    verbose("step 3. ksy matrix time: %ss"%(cputime(_time)+singular.cputime(_stime)),level=2)
    sys.stdout.flush()

    if step45==True:
        s = []
        _time,_stime = cputime(),singular.cputime()
        for p in A._ring.base_ring():
            if KSY.subst(str(self.parameter),str(p)).det()==0:
                s.append(p)
        verbose("step 4&5. det + roots time: %ss"%(cputime(_time)+singular.cputime(_stime)),level=2)
        sys.stdout.flush()

    else:
        ### step 4. Compute the determinant of the KSY Dixon Matrix

        _time,_stime = cputime(),singular.cputime()
        det =  KSY.det().sage_poly(A._ring)
        verbose("step 4. det(ksy) time: %ss"%(cputime(_time)+singular.cputime(_stime)),level=2)
        sys.stdout.flush()

        ### step 5. Solve the equation gotten in step 4 over
        ### GF(q). There may be several roots, the set of these roots is
        ### called s;

        _time,_stime = cputime(),singular.cputime()
        s = det.univariate_polynomial().roots()
        s = [ e[0] for e in s ] # strip multiplicity
        verbose("step 5. roots time: %ss"%(cputime(_time)+singular.cputime(_stime)),level=2)
        sys.stdout.flush()

    ### step 6. For each root of x_n, substitute it to the KSY Dixon
    ### Matrix gotten in step 3, then solve the linear equation to
    ### find the values of all the other monomials, in particular
    ### for all the other variables x_i.

    # convert v to something multiplicable with M
    v = singular.ideal2vector(v)

    _time,_stime = cputime(),singular.cputime()
    w = []

    #return M,v

    for root in s:
        Y = M.subst(str(self.parameter),str(root))
        Y = Y * v
        Y = smtosm(Y,self.ring)
        F = MQ(self.ring,Y.list())
        AA,Av = F.coeff_matrix()
        AA.echelonize()
        solver = MQVariety(F,AA,Av)
        ret = solver.solve_univariate(partial_solution={self.parameter:root})
```

```
            if ret!=None and len(ret)>1:
                w.append(ret)
        verbose("step 6. variable recovery: %ss"%(cputime(_time)+singular.cputime(_stime)),level=2)
        sys.stdout.flush()

        # Better checks are needed:
        #for solution in w:
        #    if solves_problem(A,solution):
        #        return solution

        return w

        ### step 7. If the algorithm fails to find a common solution of
        ### A in step 6, let s = {0,p (used in subprogram RSC)}, run
        ### step 6.

        # TODO: dead code
        s = [0,self.p]


    def rsc(self,M):
        """
        INPUT:
            M -- Dixon Matrix

        OUTPUT:
            cols,rows -- columns and rows needed to construct the KSY matrix
        """
        # step 1. Substitute a random value p in GF(q) for x_n in the
        # Dixon Matrix;

        self.p = self.ring.base_ring().random_element()
        verbose("random p=%s"%str(self.p),level=2)
        sys.stdout.flush()

        # steps 2. - 5. implemented in pure Singular (see dr.lib)
        cols,rows = singular.rsc(M,str(self.parameter),singular(str(self.p)))

        return cols,rows

    def dixon_polynomial(self,F):
        """
        INPUT:
            F -- MQ problem

        OUTPUT:
            dixon polynomial as Singular coef() matrix
        """

        return self._dixon_(F,flag=0)

    def dixon_matrix(self,F):
        """
        INPUT:
            F -- MQ problem

        OUTPUT:
            dixom matrix and monomial vector as Singular matrices
            and ideals resp.
        """
        return self._dixon_(F,flag=1)

    def _dixon_(self,F,flag=0):

        if not F.ngens() == F._ring.ngens():
            raise TypeError,"ngens needs to equal nvariables but got system with %d and %d"\
                  %(F.ngens(),F._ring.ngens())

        nvars = int(F._ring.ngens()-1)
        ngens = F.ngens()


        org_names = [str(e) for e in F._ring.gens()]
        _r_ngens = F._ring.ngens()+nvars
        _r_names = org_names + ["a%d"%i for i in range(_r_ngens-F._ring.ngens())]

        r = MPolynomialRing(F._base_ring, F._ring.ngens()+nvars, _r_names, order="lex" )

        r._singular_()

        F2_1    = singular(str(F._gens)[1:-1],type="list")

        gens = list(r.gens())[:ngens]
        gens_1 = singular(str(gens[:-1])[1:-1],type="list")

        a = r.gens()[F._ring.ngens():]
        if len(a)>1:
            a_1     = singular(str(a)[1:-1],type="list")
        else:
            a_1     = singular(str(a)[1:-2],type="list")

        self.ring = r


        dixon_polynomial,a = singular.dixon_polynomial(ngens,
                                                       F2_1.name(),
                                                       gens_1.name(),
                                                       a_1.name())

        dp = dixon_polynomial.coef(a)

        if flag==0 :
            return dp
        else:
            self.parameter = F._ring.gen(F._ring.ngens()-1)
            p,m = singular.dixon_matrix_helper(ngens,
                                               F2_1.name(),
```

```
                                                    gens_1.name(),
                                                    a_1.name(),
                                                    str(self.parameter),
                                                    dp)

            F2 = MQ(r,[f[1].sage_poly(r) for f in p])
            A,v = F2.coeff_matrix(self.parameter)
            As = singular.matrix(A.nrows(),A.ncols())
            setstr = ""
            for (i,j) in A.nonzero_positions():
                setstr += "%s[%s,%s]=%s; "%(As.name(),i+1,j+1,str(A[i,j]))
            singular.eval(setstr)
            return As,m
```

Listing A.10: DR Singular part

```
version="20060627";
category="Utilities";
// summary description of the library
info="
LIBRARY:    shared.lib Routines shared by several libs
AUTHOR:     Martin Albrecht, email: malb@informatik.uni-bremen.de

NOTE:       The dixon_polynomial code is partly a port of the maple Dixon.mpl
            implementation by Arthur D. Chtcherba <cherba@cs.panam.edu> (C)
            2003

SEE ALSO:   mq.lib dr.lib

KEYWORDS:   Dixon Resultants

PROCEDURES: dixom_polynomial(ngens,pols, vars,repl_vars)
            dixon_matrix(ngens, pols, vars, repl_vars, param)
            rsc(M, variable, value)
";

LIB "shared.lib"

proc dixon_polynomial(int ngens, list pols, list vars, list repl_vars)
{
  matrix cm[ngens][ngens];
  poly dixon_polynomial = 0;
  poly dv = 0;

  int i = 1;
  int j = 1;

  for(i=1 ; i<=ngens ; i=i+1) {
    cm[1,i] = pols[i];
  }

  for(i=2 ; i<=ngens ; i=i+1) {
    for (j=1 ; j<=ngens ; j=j+1) {
      cm[i,j] = subst(cm[i-1,j],vars[i-1],repl_vars[i-1]);
    }
  }

  // Now, divide the determinant by the product of (vars[i-1] -
  // new_vars[i-1]).  This is accomplished by subtracting ith row from
  // the (i+1)^th, and dividing the result by (vars[i-1] -
  // new_vars[i-1]).
  for(i=ngens; i>=2; i=i-1) {
    dv = (vars[i-1] - repl_vars[i-1]);
    for(j=1 ; j<=ngens ; j=j+1) {
      cm[i, j] = (cm[i, j] - cm[i-1, j])/dv;
    }
  }

  poly dixon_polynomial = det(cm);

  poly a = 1;

  for(i=1; i<=size(repl_vars)  ; i=i+1) {
    a =  a*repl_vars[i];
  }
  list l = dixon_polynomial,a;
  return(l);
  // strange thing: if I do this Singular eats all RAM,
  // if I do the same from SAGE (i.e. not in a Singular function)
  // it works.
  //def ret = coef(dixon_polynomial,a);
  //return(ret);
}

proc dixon_matrix_helper(int ngens, list pols, list vars, list repl_vars, poly param, matrix dp)
"""
This doesn't actually return the Dixon matrix but a matrix which is used to construct the
Dixon matrix.
"""
{
  //matrix dp = dixon_polynomial(ngens,pols,vars,repl_vars);

  ideal monomials;
  int mon_idx = 1;

  matrix tmp;

  poly parameter_mask = 1;

  // construct mask of monomials in x_i without parameter
  for(int i=1 ; i<=size(vars); i=i+1) {
    if(vars[i]!=param) {
      parameter_mask = parameter_mask * vars[i];
    }
```

```
  }

  // extract all monomials in x_i (without parameter)
  module polynomials;
  for(int i=1 ; i<=ncols(dp); i++) {
    polynomials[i]=dp[2,i];
    tmp = coef(polynomials[i][1],parameter_mask);
    tmp = subst(tmp,param,1);
    for(int j=1; j<=ncols(tmp); j++) {
      monomials[mon_idx] = tmp[1,j];
      mon_idx++;
    }
    monomials = simplify(monomials,4);
    mon_idx = size(monomials)+1;
  }

  //monomials = reverse(sort(simplify(monomials,4))[1]);
  //monomials = unique(monomials);
  //list l = polynomials, parameter_mask;
  list l = polynomials, unique(simplify(monomials,4));
  return(l);

  //return(coeff_matrix(polynomials, unique(monomials), parameter_mask ));

}

proc rsc(M, variable, value)
// Checks the RSC criteria for a given matrix $M$ by treating
// $variable$ as a parameter which is substitute by the value
// $value$. This function returns the columns and rows needed to
// construct the KSY matrix for M.
//
// INPUT:
//     M        -- matrix over polynomial ring $R$ over $k$
//     variable -- variable in the ring $R$
//     value    -- element of base field $k$
//
// OUTPUT:
//     cols, rows
{
  //step 1. Substitute a random value p in GF(q) for x_n in the Dixon
  //Matrix;
  matrix Mdash = subst(M,variable,value);

  // step 2. Perfom gauss elimination on the matrix gotten in
  // step 1, assume the result is M' and the rank of M' is r;

  int r = colrank(Mdash);
  int s1 = nrows(M);
  int s2 = ncols(M);

  // step 3. If M' is a square and full rank matrix then return
  // all the rows and columns in M';

  if(s1 == s2 & s1 == r) {
    list ret = range(s1),range(s2);
    return(ret);
  }

  // step 4. Fo each column m of the matrix M' construct a
  // submatrix M_s of M' of dimension s1 \times (s2-1) by
  // deleting m;

  int found = 0;

  for(int i=1 ; i<=s2 ; i=i+1) {
    matrix M_s = exclude_column(M,i);
    // if rank of M_s < r then break this loop;
    if(colrank(M_s) < r) {
      found = 1;
      break;
    }
  }

  if(found==0) {
    list ret = 0,0;
    return(ret);
  }

  // step 5. If step 4 finds a submatrix M_s, whose rank is less
  // than r then choose the columns needed for constructing a r
  // \times r submatrix of M' whose rank is r; Transpose M' and
  // perfom gauss elimination, then choose the rows needed for
  // constructing a r \times r submatrix M' whose rank is r;
  // return the rows and columns;

  intvec rows = cut(bareiss(M)[2],r);
  intvec cols = cut(bareiss(transpose(M))[2],r);
  list ret = cols,rows;
  return(ret);

}
```

A.6 XL

Listing A.11: XL

```
#!/usr/bin/env sage-python
#
# -*- Mode: Python -*-
# # vi:si:et:sw=4:sts=4:ts=4
#

"""
 AUTHOR: Martin Albrecht <malb@informatik.uni-bremen.de>



"""
# system
import pdb

# sage
#from sage.matrix.matrix_modn_sparse import Matrix_modn
from sage.misc.misc import verbose,walltime

# local includes
from algebraicattack import *
from mq import *
from misc import *
#from xl_pyx_0 import *

class XL(AlgebraicAttack):
    """
    This class implements the XL algorithm as described in [XL] so as
    some more or less well known examples to test the algorithm
    against.

    Please refer to \\class{XL}::attack for a description of
    differences between this implementation and the definition in
    [XL].

    ---
    [XL] N. Courtois et al.; Efficient Algorithms for Solving
    Overdefined Systems of Multivariate Polynomial Equations; 2000
    """

    def __init__(self):
        """
        """
        MixInSAGE() #make sure everything is setup
        self._examples = [self.example_Courtois_et_al]


    def example_Courtois_et_al(self):
        """
        Creates a MQ equal to the 'Toy Example of XL' found in [XL] on
        page 9

        However the base field of the 'Toy Example' is not defined in
        [XL] so we chose GF(127) here which seems to be the finite
        field of choice in [XL]

        Furthermore support for parameters is not implemented (yet) so
        we assign some values to the parameters a,b,m, and n
        """

        r = MPolynomialRing(GF(127), 2, 'x', order='lex');
        (x1,x2) = r.gens();

        a = 13 # some arbitrary values
        b = -29 #
        m = -47
        n = -20

        polynomials = [ x1**2 + m * x1*x2 - a,
                        x2**2 + n * x1*x2 - b ];

        F = MQ(r,polynomials)
        return F

    def attack(self,F,D=0,step_size=1,flavor=2):
        """
        Tries to solve the given polynomial system with XL as
        described in [XL]

        It is however implemented incremental that is it attacks the
        result of the D-round with D=D+1 in round D+1.

        Furthermore it checks all possible solutions to ensure no
        valid results are omitted. However it does not keep track of
        intermediate solutions between the Dth and the (D+1)th round.

        INPUT:
            self      -- a XL class
            F         -- a MQ to attack
            D         -- parameter D as defined in [XL]
            step_size -- it may be desirable not to include e.g. odd monomials, in this
                         case the step width would be set to 2.
            flavor    -- if 0, the algorithm terminates also if no solution could be found
                         if 1, the algorithm increases D by one and continues with the set
                               of equations generated for D, if no solution could be found
                         if 2, the algorithm increases D by one and continues with the
                               original set of equations, if no solution could be found

        OUTPUT:
            The solution to the polynomial equation system if any
```

```
            could be found. The solution is represented through a
            dictionary where the keys are the variables and the values
            their corresponding values. A solution may be checked by
            calling fix(solution) on a multivariate polynomial system:
            This should result in a system containing only 0 as
            polynomial.

        EXAMPLES:
            sage: xl=XL()
            sage: F = xl.example(0)
            sage: solution = xl.attack(F,4,2)
            sage: F.substitute(solution)
            sage: F.gens
            [0, 0]
        """
        time_for_attack = walltime()

        #singular._start()
        self._singular_ring = F._ring._singular_()

        self._ring = F._ring
        self._F = F#.copy()

        totaldeg = min( [f.total_degree() for f in F] )

        if(totaldeg >= D):
            D = totaldeg+1;
            verbose("redefining D to %d"%D, level=1,caller_name="XL attack")

        roots = None
        while True :
            # 1. Multiply
            F = self.equation_factory( F, D, step_size )
            F.terminate = self._F.terminate
            (A,v) = F.coeff_matrix(T="lex")
            verbose("Matrix size         : %s,%s"%(A.nrows(),A.ncols()),level=2,caller_name="XL attack")
            sys.stdout.flush()

            # 2. Linearize
            _time = walltime()

            A.echelonize()
            rank = A.rank()
            verbose("Reduced matrix size: %s,%s"%(rank,A.ncols()),level=2,caller_name="XL attack")
            verbose("multivariate time: %s"%(walltime(_time)),level=1,caller_name="XL attack")
            sys.stdout.flush()

            if flavor==4:
                return A,v

            # 3. Solve
            _time = walltime()
            p = F.gen(A.nrows()-rank,(A,v))

            if not isinstance(p,MPolynomial) or p.is_constant() or p.is_univariate():
                solver = MQVariety(F,A,v)
                #roots = solver.solve_univariate(offset = A.nrows()-rank)
                roots = solver.solve(offset = A.nrows()-rank)

            verbose("univariate time:    %s"%(walltime(_time)),level=1,caller_name="XL attack")
            sys.stdout.flush()
            if roots != None:
                verbose("attack time: %s"%(walltime(time_for_attack)),level=1,caller_name="XL attack")
                verbose("m = %d, n = %d, D = %d"%(len(F),self._ring.ngens(),D),level=1, \
                        caller_name="XL attack")
                return roots
            # 4. Repeat
            if flavor==0:
                return
            elif flavor==1:
                D=D+1
                verbose("redefining D to %d"%D,level=1,caller_name="XL attack")
                sys.stdout.flush()
            elif flavor==2:
                D=D+1
                verbose("redefining D to %d"%D,level=1,caller_name="XL attack")
                sys.stdout.flush()
                F = self._F
            elif flavor==3:
                return MQ(F._ring,A*v)


    def equation_factory(self, F, D, step_size=1):
        """
        Generates all equations of the form $\Pi_{j=1}^k x_{i_j} * l_i
        \in I_D$ as described in [XL].

        INPUT:
            self      -- \\class{XL}
            F         -- the list/system which contains $l_i$
            D         -- total degree up to which equations are generated
            step_size -- intermediate degrees may be skipped

        OUTPUT:
            list of polynomials
        """

        monomials = [ self.monomial_factory( F._ring, d, raw=True )
                      for d in range( 0, D+1-2, step_size) ]

        gens = [ self.monomial_multiply(i,f)
                 for f in F
                 for e in monomials
                 for i in e
                 #for d in range( 0, D+1-f.total_degree(), step_size)
                 #for i in monomials #in self.monomial_factory( f.parent(), d, raw=True )
                ]
```

```
        return MQ(F._ring,gens)

    def monomial_multiply(self,i,f):
        """
        Multiplies a polynomial f by a monomial i. This is done quite
        fast using the internal representation of polynomials.

        INPUT:
            i -- monomial, represented as exponent tuple
            f -- multivariate polynomial
        """

        f_dict  = f.element().dict()
        nexps = range(len(i))
        iexp = i

        res_dict = {}
        for exp in f_dict:
            _exp = exp.eadd(iexp)
            res_dict[ _exp ]=f_dict[ exp ]
        return f.parent()(polydict.PolyDict(res_dict,force_int_exponents=False,force_etuples=False))

    def monomial_factory(self, ring, D, raw=False):
        """
        Returns all monomials of the given degree D in the rings
        generators.

        INPUT:
            self -- \\class{XL}
            ring -- provides ring variables to generate monomials from
            D    -- degree to generate monomials for
            raw -- if true tuples of exponents are returned instead of
                   mpolynomials

        OUTPUT:
            list of polynomials representing the monomials

        """
        size = ring.ngens()

        # see xl_pyx
##          def am(D,size):
##              res = []
##              for d in reversed(range(D+1)):
##                  if size>1:
##                      res += [ (d,)+rest for rest in am( D-d , size-1) ]
##                  else:
##                      return [ (d,) ]
##              return res

        _list = [ ETuple(e) for e in am_pyx(D,size) ]

        if raw==False:
            return [ ring(polydict.PolyDict({elem:int(1)},force_int_exponents=False)) \
                     for elem in _list ]
        else:
            #fastest implementation:
            return _list


    def tdash(self, F, x):
        """
        Returns T' for T and a given variable x
        """

        def apply_field_equations(expvec,order):
            for i in range(len(expvec)):
                if expvec[i]>=order:
                    expvec[i]=expvec[i]-int(order-1)
            return expvec

        T = set(F._terms())
        _T = set()
        r = x.parent()
        x =  x._MPolynomial_polydict__variable_indices()[0] #asserting only one var

        for t in T:
            m,c = list(t[0]),t[1]
            m[x]=m[x]+int(1) # multiply with var x
            apply_field_equations(m,r.base_ring().order()) # reduce with field_equations
            if (tuple(m),c) in T: # check if in T
                _T.add(t) #  add to _T
        return [r(polydict.PolyDict({m:c},force_int_exponents=False)) for m,c in list(_T) ]
```

Listing A.12: XL (Pyrex)

```
# __no_preparse__

def am_pyx(D,size):
    _sig_on
    r = am_pyx2(int(D),int(size))
    _sig_off
    return r


cdef am_pyx2(D,size):
    res = []
    for d2 from 0 <= d2 <= D:
        d = D-d2
        if size>1:
            # TODO: avoid recursive function call
            for rest in am_pyx2( d2 , size-1):
                res = res + [ (d,) + rest ]
```

```
        else:
            return [ (d,) ]
    return res
```

A.7 Specialized Attacks

Listing A.13: Specialized Attacks

```
"""
AUTHOR: Martin Albrecht <malb@informatik.uni-bremen.de>

Attacks that use a Groebner basis algorithm repeatedly to compute a
Groebner basis.

 * Meet-in-the-Middle by Cid, Murphey, Robshaw
 * Groebner Surfing

EXAMPLES:
    sage: F,s = ctc_MQ(Nr=8,B=1,order='lex')
    sage: time gb = mitm(F)
    CPU times: user 4.73 s, sys: 0.30 s, total: 5.04 s
    Wall time: 16.17

    sage: time gb2 = F.ideal().groebner_basis()
    CPU times: user 1.30 s, sys: 0.05 s, total: 1.35 s
    Wall time: 363.94

    sage: gb2 == gb
    True

"""
import thread, sys

from sage.interfaces.singular import singular
from sage.rings.multi_polynomial_ring import *
from sage.misc.misc import verbose, get_verbose

from mq import *

def singular_groebner_function(l):
    """
    INPUT: list of polynomials
    OUTPUT: list of polynomials, cputime
    """
    R = l[0].parent()
    R._singular_().set_ring()

    t = singular.cputime()
    gb = [R(e) for e in singular.ideal(str(l)).groebner()]
    return gb,singular.cputime(t)


def mitm(F, gb_func=singular_groebner_function, force_lex=False, surf=False):
    """
    Implementation of the Meet in the Middle Attack

    'However it is well-known that the equation systems derived from the
    AES are highly structured [...]. In particular, these systems might be
    viewed as 'iterated' systems of equations, with similar blocks of
    multivariate quadratic equations repeated for every round. These
    blocks are connected to each other via the input and output variables,
    as well as the key schedule. When working with systems with such
    structure, a promising technique to find the overall solution is, in
    effect, a meet-in-the-middle approach: rather than attempting to solve
    the full system of equations for $n$ rounds (we assume that $n$ is
    even), we can try to solve two subsystems with $n/2$ rounds, by
    considering the output of round $n/2$ (which is also the input of
    round $n/2 + 1$) as variables. By choosing an appropriate monomial
    ordering we obtain two sets of equations (each covering half of the
    encryption operation) that relate these variables with the round
    subkeys. These two systems can then be combined along with some
    other equations relating the round subkeys. This gives a third smaller
    system which can be solved to obtain the encryption key.

    [...]

    This technique is cryptographically intuitive and is in fact a simple
    application of Elimination Theory, in which the Groebner bases
    are computed with respect to the appropriate monomial ordering to
    eliminate the variables that do not appear in rounds $n$ and $n + 1$. One
    problem with this approach is that computations using elimination
    orderings (such as lexicographic) are usually less efficient than those
    with degree orderings (such as graded reverse lexicographic). Thus,
    for more complex systems, we might expect that using lexicographic
    ordering in the two main subsystems would yield only limited benefit
    when compared with graded reverse lexicographic ordering for the full
    system. As an alternative, we could simply compute the Groebner bases
    for the two subsystems (using the most efficient ordering) and combine
    both results to compute the solution of the full set equations.

    (C. Cid , S. Murphy, and M.J.B. Robshaw, Small Scale Variants of the AES)
    """

    singular.option("redSB")

    if surf:
        gb_func = groebner_surf
```

```
    rounds = F.round

    R = F._ring
    k = R.base_ring()

    if is_MPolynomialRing(R):
        ring_constructor = lambda nvars,vars, order: MPolynomialRing(k,nvars,vars,order=order)
    elif isinstance(R, MPolynomialRingGF2):
        ring_constructor = lambda nvars,vars, order: MPolynomialRingGF2(nvars,vars,order=order)

    if not force_lex:
        term_order = R.term_order()
    else:
        term_order = "lex"

    Bs = len( [f for f in F.variables() if str(f).startswith("K000")] )

    # we assume that F.rounds[0] is an initial key addition and we
    # don't consider it to be a real round. So we add rounds[0] and
    # rounds[1]

    rounds = [list(rounds[0])+list(rounds[1])]+list(rounds[2:])

    Nr = len(rounds)

    # if we only have one round, we don't split
    if Nr < 2:
        if not surf:
            gb, t = gb_func(rounds[0])
        else:
            gb, t = gb_func(MQ(R,[rounds[0]]))
        verbose("gb time: %f"%(t),level=2)
        sys.stdout.flush()
        return gb

    split = Nr/2
    assert(isinstance(split, int)) #always true for python2.4


    ###############
    # Left
    ###############

    #if split>2:
    #    lpols = mitm(MQ(R,rounds[:split]))
    #    return lpols

    lpols, lvars = [],[]

    lvars = []

    if surf:

        for i in reversed(range(split+1)):
            for j in range(Bs):
                if i < split:
                    lvars.append( "K%03d%03d"%(i+1,j) )
                lvars.append( "X%03d%03d"%(i+1,j) )
                if i < split:
                    lvars.append( "Y%03d%03d"%(i+1,j) )
                    lvars.append( "Z%03d%03d"%(i+1,j) )
        lvars += ["K000%03d"%(j) for j in range(Bs)]
        lvars = reversed( lvars )

    else:

        lvars += ["K%03d%03d"%(i+1,j) for i in reversed(range(split)) for j in range(Bs)]
        lvars += ["X%03d%03d"%(i+1,j) for i in reversed(range(split+1)) for j in range(Bs)]
        lvars += ["Y%03d%03d"%(i+1,j) for i in reversed(range(split)) for j in range(Bs)]
        lvars += ["Z%03d%03d"%(i+1,j) for i in reversed(range(split)) for j in range(Bs)]
        lvars += ["K000%03d"%(j) for j in range(Bs)]

    lring = ring_constructor(len(lvars),lvars,order=term_order)

    if not surf:
        for r in range(split):
            lpols += rounds[r] #flatten
        lpols = [lring(str(f)) for f in lpols]
    else:
        for r in range(split):
            lpols += [[lring(str(f)) for f in rounds[r]]]

    lF = MQ(lring,lpols)
    verbose(" Left: Variables: %d Equations: %d Monomials: %d"%(len(lF.variables()),
                                                                len(lF.gens()),
                                                                len(lF.monomials()) ),
            level=3)
    sys.stdout.flush()

    if not surf:
        lgb, lt = gb_func(lpols)
    else:
        lgb, lt = gb_func(lF)
    verbose(" left groebner basis time: %f"%(lt),level=2)
    sys.stdout.flush()
    l = [R(str(f)) for f in lgb]

##      print "left"
##      print latex(lpols)
##      print latex(lring)
##      print latex(lgb)

    ############
    # Right
    ############
```

```
    rpols, rvars = [],[]

    rvars = []

    if surf:

        rvars += ["K000%03d"%(j) for j in range(Bs)]
        for i in range(split,Nr):
            for j in range(Bs):
                rvars.append("Z%03d%03d"%(i+1,j))
                rvars.append("Y%03d%03d"%(i+1,j))
                rvars.append("X%03d%03d"%(i+1,j))
                rvars.append("K%03d%03d"%(i+1,j))

        rvars = reversed( rvars )

    else:
        rvars += ["K%03d%03d"%(i+1,j) for i in range(split,Nr) for j in reversed(range(Bs))]
        rvars += ["Z%03d%03d"%(i+1,j) for i in range(split,Nr) for j in reversed(range(Bs))]
        rvars += ["Y%03d%03d"%(i+1,j) for i in range(split,Nr) for j in reversed(range(Bs))]
        rvars += ["X%03d%03d"%(i+1,j) for i in range(split,Nr) for j in reversed(range(Bs))]
        rvars += ["K000%03d"%(j) for j in range(Bs)]

    rring = ring_constructor(len(rvars),rvars,order=term_order)

    if not surf:
        for r in range(split,Nr):
            rpols += rounds[r]
        rpols = [rring(str(f)) for f in rpols]
    else:
        for r in reversed(range(split,Nr)):
            rpols += [[rring(str(f)) for f in rounds[r]]]

    rF = MQ(rring,rpols)
    verbose("Right: Variables: %d Equations: %d Monomials: %d"%(len(rF.variables()),
                                                                len(rF.gens()),
                                                                len(rF.monomials()) )\
            ,level=3)
    sys.stdout.flush()

    if not surf:
        rgb, rt = gb_func(rpols)
    else:
        rgb, rt = gb_func(rF)
    verbose("right groebner basis time: %f"%(rt),level=2)
    sys.stdout.flush()
    r = [R(str(f)) for f in rgb]

##      print "right"
##      print latex(rpols)
##      print latex(rring)
##      print latex(rgb)

    #################
    # Union
    #################

    uF = MQ(R,r+l) # including doubles
    verbose("Union: Variables: %d Equations: %d Monomials: %d"%(len(uF.variables()),\
                                                                len(uF.gens()),\
                                                                len(uF.monomials()) )\
            ,level=3)
    sys.stdout.flush()
    #gb, ut = gb_func(l+r)
    gb, ut = singular_groebner_function(l+r)

    verbose("union groebner basis time: %f"%(ut),level=2)

    verbose("all groebner basis time: %f"%(lt+rt+ut),level=2)
    sys.stdout.flush()
    return gb


def sing_gb(F):
    """
    """
    t = singular.cputime()
    gb = F.ideal().groebner_basis()
    verbose("all groebner basis time: %f"%(singular.cputime(t)),level=2)
    return gb

def groebner_surf(F):
    """
    """

    singular.option("redSB")
    gb = singular(0,"ideal")
    R = F.ring()
    all_time = 0.0
    for i in range(len(F.round)):
        t = singular.cputime()
        #gb = R.ideal(gb + list(F.round[i])).groebner_basis("singular:std")
        gb = (gb + singular(list(F.round[i]),"ideal")).std()
        t = singular.cputime(t)
        all_time += t
        if get_verbose() > 1:
            print i,t
            sys.stdout.flush()
            gbMQ = MQ(F._ring,[R(e) for e in gb])
            print "Variables: %d Equations: %d Monomials: %d"%(len(gbMQ.variables()),\
                                                               len(gbMQ.gens()),\
                                                               len(gbMQ.monomials()) )
            sys.stdout.flush()
    return [R(e) for e in gb],all_time
```