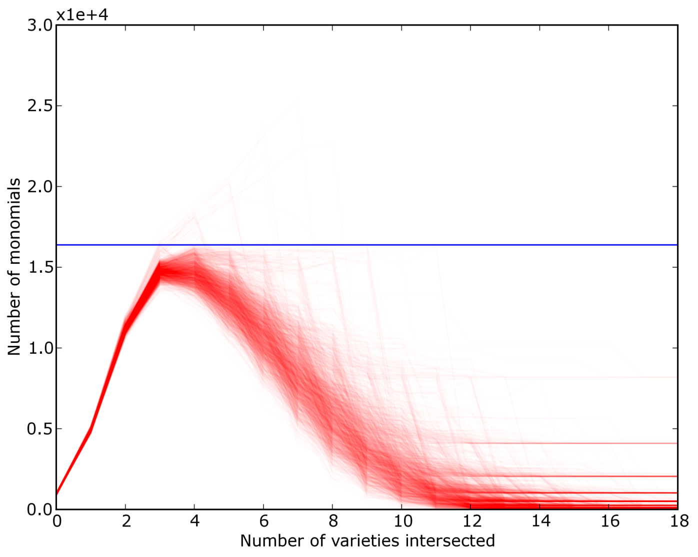
Diplomarbeit

Algebraic attacks specialized for $\mathbb{F}_2$

Thomas Dullien

28. August 2008



Betreuer: Roberto Avanzi

HGI, University of Bochum

The picture on the title shows the number of monomials that occur in intermediate results of the pseudo-euclidian algorithm on random quadratic equation systems in 14 variables.

Contents

Einleitung

Introduction

In his landmark paper, Shannon stated that breaking a good cipher should 'require at least as much work as solving a system of simultaneous equations in a large number of unknowns of a complex type'.

In the years since, most published cryptanalytic successes in symmetric cryptography were obtained by statistical or combinatorial weaknesses in the ciphers that were analyzed. Equation-system solving was not at the forefront, partially because most ciphers were built without a 'clean' algebraic structure in which to work, partially because efficient algorithms for the algebraic solving of equation systems were not available.

Interest in algebraic methods for attacks on cryptosystems was revived from two directions in the late 90's and early 00's: On the one side, cryptographers had attempted to build asymmetric systems from the hardness of the MQ-problem (the problem of solving multivariate quadratic equation systems over an arbitrary field). Patarin [Patarin95] showed how to break a signature scheme originally proposed by Matsumoto and Imai [MatImai88] as early as 1988 by making use of the special structure of the equations in question. Several attempts at using general-purpose equation solving algorithms were made, resulting in the break of HFE [FauJou03] using a new Gröbner-Base algorithm introduced the year before [Faugere02]. This showed that more efficient algorithms for equation system solving could have a direct application in cryptanalysis.

The other direction from which interest in algebraic attacks was reignited was the emergence of Rijndael as the new advanced encryption standard (AES): Rijndael was designed to resist the standard statistical attacks and was modeled to use only the 'normal' field operations of $\mathbb{F}_{2^8}$, yielding a very clean algebraic structure. Soon, several researchers proposed methods that allowed modeling Rijndael as one large system of multivariate polynomial equations over $\mathbb{F}_{2^8}$ or $\mathbb{F}_2$, and began looking into methods for solving such systems [Courtois02]. Some algorithms based on relinearization were proposed, and a lot of speculation about their actual efficiency ensued, until it was finally determined that the relinearization algorithms proposed were in fact inefficient variants of the Buchberger algorithm for computing a Gröbner basis.

Actual practical successes of algebraic cryptanalysis are few and far between: Outside of the successes against multivariate cryptosystems and some degree of success against stream ciphers (where obtaining huge quantities of equations was possible, allowing linear algebra to play a role), one could argue that at the time of this writing, no algebraic method has succeeded in breaking a 'real' block cipher. This is not for a lack of trying: Several research groups are approaching the topic from different angles, yet tangible results have not really materialized.

Due to the lack of tangible results, it was proposed that algorithms that solve algebraic equations in arbitrary finite fields might be too general a setting, and the current direction of research appears to be work on methods specialized to certain base fields (such as $\mathbb{F}_2$, or $\mathbb{F}_{2^8}$).

As of mid-2007, the main approaches to algebraic cryptanalysis appear to be the following:

1. Work on more efficient algorithms for calculating Gröbner Bases in an eliminiation order [Brick06], potentially under specialization to $\mathbb{F}_2$ [BrickDrey07].

2. Specializing the problem to $\mathbb{F}_2$ and attempting to model the problem as 'message passing on a graph', with some further improvements [RadSem06, RadSem07]

3. Specializing the problem to $\mathbb{F}_2$ and attempting to convert the problem to an instance of the well-known 3SAT problem. This instance can then potentially be solved by using heuristic SAT solving algorithms [Bard07, BaCouJef07, McDChaPie07, CouBar07] . The proof for NP-hardness of MQ is based on a reduction to 3SAT - it therefore makes sense to attempt to utilize known algorithms for solving 3SAT to solve MQ.

In the cryptographic setting, one is usually not interested in solutions that lie outside of the base field of the multivariate polynomial ring. This means that the proper environment in which to analyze the problem (and the different approaches) is the *ring of Boolean functions*

$$\mathcal{B}_n := \mathbb{F}_2[x_1, \ldots, x_n]/\langle x_1^2 + x_1, \ldots, x_n^2 + x_n \rangle$$

This thesis will put the focus on the special structural properties of this ring (for example the fact that it happens to be a principal ideal ring). Some surprising results occur:

1. Even though $\mathcal{B}_n$ is not Euclidian, there is a very simple algorithm that performs very much like the Euclidian algorithm: Given a set of generators $g_1, \ldots, g_n$ of an ideal $I \subset \mathcal{B}_n$, it returns a single generator

g with $\langle g \rangle = I$ in just $n+1$ additions and n multiplications in $\mathcal{B}_n$. Geometrically, this algorithm yields an intersection of a set of varieties.

Interestingly, this algorithm can be generalized to arbitrary finite fields, yielding a proof for the following intuitively clear statement:

> Let $K := \mathbb{F}_q[x_1, \dots, x_n]$ be a multivariate polynomial ring over a finite field and $f_1, \dots, f_n \in K$ a set of polynomials. If one is not interested in solutions that lie in the algebraic closure of $\mathbb{F}_q$, one can perform calculations in $K/\langle x_1^q - x_1, \dots, x_n^q - x_n, \rangle$. This ring is a principal ideal ring isomorphic to the ring of subsets of K^n, and the operations in this ring corresponding to the usual set operations $\cup, \cap$ can be computed easily by polynomial arithmetic.

For cryptographic purposes, this algorithm will calculate the solution to an arbitrary system of equations (provided that the solution is unique and lies entirely in the base field) in n polynomial multiplications and $n+1$ polynomial additions. We will see that while this looks great at first glance, it still fails to solve any relevant equation system in reasonable time.

2. The 'message passing on a graph'-approach presented in [RadSem06, RadSem07] is put into an algebraic-geometric context: It can be viewed as an iterative algorithm that projects input varieties to lower-dimensional subspaces on which they are intersected. This eliminates points in the lower-dimensional subspaces that do not occur in all projections of varieties. The resulting variety is 'lifted' to the original spaces again, where it is used to eliminate points that are not common to all varieties. As a corolary of this it becomes evident that *linear* functions are "worst-case" for this algorithm, which leads to the (unproven) conjecture the algorithm performs best on very *non-linear* equation systems.

3. The pseudo-Euclidian algorithm leads to a number of interesting corollaries:

   (a) Iterated multivariate polynomial multiplication over $\mathbb{F}_2$ is NP-hard

   (b) The combinatorial question "How many monomials does the product of given polynomials $f_1, \dots, f_n$ have ?" is of cryptographic relevance: Even an approximate answer to this question will privide information about the Hamming-weight of the solution of the system of equations.

4. A generalized definition of "approximation" for Boolean functions in the context of algebraic cryptanalysis is given. This definition of "ap-

proximation" is furthermore found to generalize Courtois' trick of lowering the degree of a system of equations by multiplying with carefully-chosen polynomials. The concept of algebraic immunity can be reinterpreted as nonexistence of approximations of low degree.

5. An algorithm that attempts to "approximate" a Boolean polynomial by one of lower monomial count is constructed and evaluated. The results can be only described as complete failure, and the algorithm needs to be re-designed if it is to be of any use.

Generally, emphasis has been put on making use of the structure of $\mathcal{B}_n$ and on attempts to understand the geometry behind the algorithms.

Chapter 1

Definitions and Notation

In the following, some essential facts about the ring of Boolean functions will be defined and proved. A bit of time will be spent building "geometric" ways of looking at polynomial arithmetic. Due to special properties of $\mathbb{F}_2$ (and hence $\mathcal{B}_n$), many 'classical' constructions take on a slightly different twist.

Some results in the following do not appear in the literature in the same way, and a bit of notation that is used throughout the thesis will be introduced. A bit of care is advisable when reading this chapter - it is probably the most "strenous" part of this thesis.

1.1 Essentials

Definition 1.1.1 (Ring of Boolean Functions)**.** The *ring of Boolean functions* is the principal object studied throughout this thesis:

$$\mathcal{B}_n := \mathbb{F}_2[x_1, \dots, x_n]/\langle x_1^2 + x_1, \dots, x_n^2 + x_n\rangle$$

Definition 1.1.2 (Variety)**.** The *variety* of a function $f \in \mathcal{B}_n$ is the subset of $\mathbb{F}_2^n$ on which f vanishes:

$$V(\langle f\rangle) := \{v \in \mathbb{F}_2^n | f(v) = 0\}$$

One can interpret this as a mapping $V : \mathcal{B}_n \to \mathfrak{P}(\mathbb{F}_2^n)$, e.g. from the Boolean functions to the powerset of $\mathbb{F}_2^n$. Often, the $\langle\rangle$ symbols will be omitted if no ambiguities are introduced.

Through this mapping, polynomial arithmetic in $\mathcal{B}_n$ can be interpreted as operations on elements in $\mathfrak{P}(\mathbb{F}_2^n)$, e.g. as set-operations. This yields geometric interpretations to the algebraic operations.

Lemma 1.1.3. *Given $f, g \in \mathcal{B}_n$, the product $h := fg$ has as variety the union of the two individual varieties.*

$$V(h) = V(fg) = V(f) \cup V(g)$$

This is analogous to what happens over any base field: Multiplying polynomials is the same as taking the union of the solution sets. The next results are more "special": $\mathbb{F}_2$ starts showing properties not shared by most other base fields.

Lemma 1.1.4. *Given $f \in \mathcal{B}_n$, the $f + 1 \in \mathcal{B}_n$ has as variety exactly the set-theoretic complement of f.*

$$V(f+1) = \overline{V(f)}$$

Now that multiplication and addition of 1 have given a geometric interpretation, the logical next thing to study is polynomial addition. In our case, it admits a particularly simple geometric interpretation:

Lemma 1.1.5. *Given $f, g \in \mathcal{B}_n$, the sum $h := f + g \in \mathcal{B}_n$ has as variety the complement of the symmetric difference of the varieties of f and g.*

This is the same as the union of the intersection of $V(f)$ and $V(g)$ with the complement of the union of $V(f)$ and $V(g)$.

The sum of two polynomials has as variety the complement of the symmetric difference of the two varieties.

$$h = f + g \Rightarrow V(h) = \overline{V(f) \bigoplus V(g)} = (V(f) \cap V(g)) \cup \overline{(V(f) \cup V(g))}$$

Proof.

1. Let $x \in V(f) \cap V(g) \Rightarrow f(x) = 0 \wedge g(x) = 0 \Rightarrow x \in V(h)$.
2. Let $x \in \overline{V(f) \cup V(g)} \Rightarrow f(x) = 1 \wedge g(x) = 1 \Rightarrow x \in V(h)$.
3. Let $x \in V(h) \Rightarrow \underbrace{(f(x) = 0 \wedge g(x) = 0)}_{\Rightarrow x \in V(f) \cap V(g)} \vee \underbrace{(f(x) = 1 \wedge g(x) = 1)}_{\Rightarrow x \in \overline{V(f) \cup V(g)}}$

□

Given a single point in $\mathbb{F}_2^n$, it is not difficult to construct a polynomial in $\mathcal{B}_n$ that vanishes on exactly this point.

Lemma 1.1.6. *For each $y \in \mathbb{F}_2^n$ we can construct f_y so that $V(f_y) = \{y\}$.*

Proof. Let $(y_1, \dots, y_n) = y \in \mathbb{F}_2^n$. Consider the polynomial

$$f_y := \underbrace{(\prod_{i=0}^{n}(x_i + y_i + 1))}_{:=p} + 1 \in \mathcal{B}_n$$

This polynomial evaluates to zero iff p evaluates to 1, which is equivalent to $\forall i \; x_i = y_i$. Hence $V(f_y) = \{y\}$. □

Corollary 1.1.7. *For each $F \subseteq \mathbb{F}_2^n$ there exists $f \in \mathcal{B}_n$ so that $F = V(f)$. Formulated differently: For any subset F of $\mathbb{F}_2^n$ there is at least one* single *polynomial f with F as variety.*

Proof. Let $F := \{y_1, \dots, y_i\} \subseteq \mathbb{F}_2^n$. and $f := \prod_{j=0}^{i} f_{y_j}$. Then $V(f) = F$. □

We can regard this construction as a mapping

$$V^{-1} : \mathfrak{P}(\mathbb{F}_2^n) \to \mathcal{B}_n$$

Corollary 1.1.8. *The mapping V is a ring homomorphism between $(\mathcal{B}_n, +, \dot{})$ and $(\mathfrak{P}(\mathbb{F}_2^n), \overline{\bigoplus}, \cup)$.*

Proof. From 1.1.5 it follows that $V(f_1 + f_2) = \overline{V(f_1) \bigoplus V(f_2)}$.

From 1.1.3 it follows that $V(f_1 f_2) = V(f_1) \cup V(f_2)$.

Since $V(1) = \emptyset$ and $X \cup \emptyset = X \forall X \in \mathfrak{P}(\mathbb{F}_2^n)$, the neutral element maps to the neutral element and all requirements for a homomorphism are satisfied. □

Corollary 1.1.9. *The mapping V and V^{-1} form a bijection between $\mathfrak{P}(\mathbb{F}_2^n)$ and $\mathcal{B}_n$.*

Proof.

V is injective. Proof indirect: Assume that $\exists f, g \in \mathcal{B}_n, f \neq g, V(f) = V(g)$. Then $f + g \neq 0$, but $\overline{V(f) \bigoplus V(g)} = \overline{\emptyset} = \mathbb{F}_2^n$ in contradiction to 1.1.5.

Surjectivity follows from 1.1.7. □

So if we can construct a polynomial for each individual point, and if multiplication between polynomials is the same as the union of the solution sets, we see what the irreducible elements in $\mathcal{B}_n$ are: Any set $s \in \mathfrak{P}(\mathbb{F}_2^n)$ can be decomposed into individual points, and every polynomial f in $\mathcal{B}_n$ is therefore a product of single-point-solution-polynomials as described in 1.1.6.

Corollary 1.1.10. *From 1.1.7, 1.1.3 and 1.1.6 it follows that the polynomials in 1.1.6 are exactly the irreducible elements of $\mathcal{B}_n$*

Corollary 1.1.11. $(\mathcal{B}_n, +, \dot{})$ *is isomorphic to* $(\mathfrak{P}(\mathbb{F}_2^n), \overline{\bigoplus}, \cup)$.

We can now use this bijection to replace some polynomial multiplications with additions:

Corollary 1.1.12. *For $f, g \in \mathcal{B}_n$ with $V(f) \cap V(g) = \emptyset$, we can calculate the product $h := fg$ by two additions: $fg = f + g + 1$*

Proof.

$$V(f) \cup V(g) = V(f) \bigoplus V(g) = \overline{\overline{V(f) \bigoplus V(g)}} = \overline{V(f+g)} = V(f+g+1)$$

□

What shape do the irreducible elements in our ring have ? What is the relation between the values of the solution to a single-point-solution-polynomial f and the monomials in f ? The following yields a first answer, relating the Hamming-weight of the solution to the number of monomials in f.

Corollary 1.1.13. *For $y \in \mathbb{F}_2^n, f_y$ contains 2^j monomials where $j = |\{y_i \in y, y_i = 0\}|$.*

Proof. Inductively:

Let $|y| = n \Rightarrow y = (1, \ldots, 1)$. Then $f_y = (\prod_{i=1}^n x_i) + 1$ and the number of monomials is 1.

Now let $|y| = n - 1$, without loss of generality $y_n = 0$. Then

$$f_y = (\prod_{i=1}^{n-1} x_i)(x_n + 1) + 1 = \prod_{i=1}^{n} x_i + \prod_{i=1}^{n-1} x_i + 1$$

and the number of monomials is 2.

Now let $|y| = n - 2$, without loss of generality $y_n = 0, y_{n-1} = 0$. Then

$$f_y = (\prod_{i=1}^{n-2} x_i)(x_{n+1} + 1)(x_n + 1) + 1 = (\prod_{i=1}^{n-1} x_i)(x_n + 1) + (\prod_{i=1}^{n-2} x_i)(x_n + 1) + 1$$

and the number of monomials is 4. Each time an $y_i = 0$, the products are split in two. □

Our above constructions have given us union, complement, and complement-of-symmetric-difference as elementary operations. It is immediately clear how to recover symmteric difference from this – but what about other operations ? The ability to *intersect* sets would be very valuable. The next result shows how.

Lemma 1.1.14. *Intersection*

Given $f, g \in \mathcal{B}_n$, we can easily calculate h so that $V(h) = V(f) \cap V(g)$.

$$V(f) \cap V(g) = V(f + g + fg)$$

Proof.

$$\begin{aligned} V(f) \cap V(g) &= \overline{\overline{V(f)} \cup \overline{V(g)}} \Rightarrow V(f) \cap V(g) \\ &= V(\langle (V(f) + 1)(V(g) + 1) + 1 \rangle) \\ &= V(\langle f_1 + f_2 + f_1 f_2 \rangle) \end{aligned}$$

□

Corollary 1.1.15. *$\mathcal{B}_n$ is a principal ideal ring.*

Proof. Let $I := \langle f_1, \ldots, f_i \rangle$. Then we can construct

$$h := (\prod_{j=1}^{i} (f_j + 1)) + 1$$

for which $V(h) = V(I)$ and $\langle h \rangle = I$. □

Corollary 1.1.16. *The last corollary yields us a sort-of-Euclidian algorithm that allows us to calculate the generator g of an Ideal $I \subset \mathcal{B}_n$ from a set of generators $g_1, \ldots, g_n$ in $n+1$ ring additions and n ring multiplications.*

This algorithm which will be called *pseudo-Euclidian* algorithm is investigated in further detail in Chapter 6. The (initially surprising) generalization of this algorithm to arbitrary finite fields is given in Chapter 7.

1.2 Representation of Boolean Functions

We trail the exposition found in [Carle07]. For more compact representation, we use multi-exponents: The monomial $\prod_{i=1}^{n} x_i^{u_i}$ is written as x^u.

1.2.1 Algebraic Normal Form (ANF)

The classical representation of elements of $\mathcal{B}_n$ is the *algebraic normal form.*

$$f(x) = \sum_{i \in \mathbb{F}_2^n} a_i x^i$$

Let $cov(x) := \{y \in \mathbb{F}_2^n, y_i \geq x_i \forall i\}$ and $supp(x) := \{i \in \mathbb{N}, x_i = 1\}$. The Boolean function f takes value

$$f(x) = \sum_{i \in cov(x)} a_i = \sum_{supp(i) \subseteq supp(x)} a_i$$

This is due to the fact that a monomial x^u evaluates to 1 if and only if $x_i \geq u_i$ holds for all i.

Since all coefficients are elements of $\mathbb{F}_2$, we can associate a different Boolean function with f: Given a vector $x \in \mathbb{F}_2$, this function retrieves a_x from f. This Boolean function is called the *binary Möbius transform* and denoted f°.

Definition 1.2.1 (Binary Möbius Transform)**.** Let $f \in \mathcal{B}_n$ and $f(x) = \sum_{i \in \mathbb{F}_2^n} a_i x^i$. Then the Boolean function f° with

$$f^\circ(u) := a_u \quad \text{for } u \in \mathbb{F}_2^n$$

is called the *binary Möbius transform* of f.

Lemma 1.2.2. *Let* $f = \sum_{u \in \mathbb{F}_2^n} a_u x^u$. *Then*

$$a_u = \sum_{supp(x) \subseteq supp(u)} f(x)$$

.

Proof. We refer to [Carle07] for the proof. □

Interestingly, applying the Möbius transform twice yields the identity mapping:

Lemma 1.2.3. *Möbius-Image-Lemma*

Let $f := \sum_{i \in \mathbb{F}_2^n} a_i x^i$ *and* $f^\circ := \sum_{i \in \mathbb{F}_2^n} b_j x^i$ *with* $f^\circ(u) = a_u$ *for* $u \in \mathbb{F}_2^n$. *Then* $f(u) = b_u$ *holds for* $u \in \mathbb{F}_2^n$.

Proof. The proof will be conducted below in the context of the square lemma. □

Möbius-Transform on the ANF

One can derive the Möbius transform of f in an alternate way (not taken in [Carle07]) which is potentially more enlightening and yields a canonical algorithm for calculating it:

We already know the mappings V and V^{-1}. We now add a third mapping: log. This mapping takes an element f of $\mathcal{B}_n$ and maps it to a subset of $\mathbb{F}_2^n$ by taking the 'vectors of exponents' from each monomial in f (which are elements of $\mathbb{F}_2^n$).

Definition 1.2.4 (log)**.** log : $\mathcal{B}_n \to \mathfrak{P}(\mathbb{F}_2^n)$. This maps a polynomial to the *set of vectors of exponents of it's monomials*, in the following manner: $f \in \mathcal{B}_n$ can be written as

$$\sum_{e_i \in E} x_1^{e_{i1}} \dots x_n^{e_{in}} \quad \text{for } E \subset \mathbb{F}_2^n$$

The mapping log now maps a polynomial to the set of 'vectors of exponents', so $\log(f) = E$.

Through this mapping, we can associate *two* elements of $\mathfrak{P}(\mathbb{F}_2^n)$ with each $f \in \mathcal{B}_n$: The *variety* $V(f)$ and also $\log(f)$.

Now, we can use the mappings log and V^{-1} to construct a permutation M on $\mathcal{B}_n$ in the following manner:

$$M : \mathcal{B}_n \to \mathcal{B}_n, \quad M := V^{-1} \circ \log$$

$$\begin{tikzcd} \mathcal{B}_n \arrow[r, "\log"] & \mathfrak{P}(\mathbb{F}_2^n) \arrow[r, "V^{-1}"] & \mathcal{B}_n \end{tikzcd}$$

This means we take a polynomial, collect the $\mathbb{F}_2$-vectors formed by its exponents, and interpolate a polynomial that vanishes exactly there.

Likewise, we have an inverse permutation M^{-1} on $\mathcal{B}_n$ given by

$$M^{-1} : \mathcal{B}_n \to \mathcal{B}_n, \quad M^{-1} := \log^{-1} \circ V$$

$$\begin{tikzcd} \mathcal{B}_n & \mathfrak{P}(\mathbb{F}_2^n) \arrow[l, "\log^{-1}"'] & \mathcal{B}_n \arrow[l, "V"'] \end{tikzcd}$$

This means we take a polynomial, calculate the set of solutions, and use these solution vectors as exponent-vectors for a new polynomial.

Proposition 1.2.5. *Square Lemma: $M \circ M \circ M \circ M = id$. The following diagram holds:*

$$\begin{tikzcd} \mathcal{B}_n \arrow[r, "\log"] & \mathfrak{P}(\mathbb{F}_2^n) \arrow[r, "V^{-1}"] & \mathcal{B}_n \arrow[d, "\log"] \\ \mathfrak{P}(\mathbb{F}_2^n) \arrow[u, "V^{-1}"] & & \mathfrak{P}(\mathbb{F}_2^n) \arrow[d, "V^{-1}"] \\ \mathcal{B}_n \arrow[u, "\log"] & \mathfrak{P}(\mathbb{F}_2^n) \arrow[l, "V^{-1}"'] & \mathcal{B}_n \arrow[l, "\log"'] \end{tikzcd}$$

Proof. We need a bit of machinery first:

Proposition 1.2.6. *Basic Monomial Lemma: Let f be monomial. Then*

$$\log \circ V^{-1} \circ \log(f) = \overline{V(f)} \bigoplus \begin{pmatrix} 0 \\ \dots \\ 0 \end{pmatrix} = V(f+1) \bigoplus \begin{pmatrix} 0 \\ \dots \\ 0 \end{pmatrix}$$

Proof. Consider $\mathcal{B}_n$ with $n = 1$. The truthfulness of the above lemma can be verified manually:

1. Let $f = x$. $\log \circ V^{-1} \circ \log(x) = \{(1), (0)\}$. Likewise, $V(x) = \{(0)\} \Rightarrow \overline{V(x)} = \{(1)\} \Rightarrow \overline{V(x)} \oplus (0) = \{(1), (0)\}$

2. Let $f = 1$. $\log \circ V^{-1} \circ \log(1) = \{(1)\}$. Likewise, $V(1) = \emptyset \Rightarrow \overline{V(x)} = \{(0), (1)\} \Rightarrow \overline{V(x)} \oplus (0) = \{(1)\}$

This can be extended to higher n. □

Corollary 1.2.7. *For f monomial, it holds that*

$$V^{-1} \circ \log(f) = \log^{-1}(V(f+1)) + \log^{-1}(0) = \log^{-1}(V(f+1)) + 1$$

Proposition 1.2.8. *The above diagram holds for all monomial f.*

Proof.

$$\begin{aligned}
M \circ M \circ M \circ M(f) &= M \circ M \circ M \circ V^{-1} \circ \log(f) \\
&= M \circ M \circ M(\log^{-1}(V(f+1)) + 1)) \\
&= M \circ M \circ V^{-1} \circ \log(\log^{-1}(V(f+1)) + 1) \\
&= M \circ M \circ V^{-1}(V(f+1) \oplus \log(1)) \\
&= M \circ M \circ V^{-1}(V(f+1) \oplus 0) \\
&= M \circ M(f + 1 + V^{-1}(0))
\end{aligned}$$

From this it follows that

$$= (f + 1 + V^{-1}(0)) + 1 + V^{-1} = f$$

and we are done. □

Corollary 1.2.9. *The above diagram holds for all polynomials f with uneven number of monomials.*

Proof. Let $f = f_1 + \cdots + f_k$. The basic lemma for monomials extends cleanly:

$$\log \circ V^{-1} \circ \log(f_1 + \cdots + f_k) = \log \circ V^{-1} \circ \log(f_1) \oplus \cdots \oplus \log \circ V^{-1} \circ \log(f_k)$$

Therefore

$$\log \circ V^{-1} \circ \log(f_1 + \cdots + f_k) = V(f_1 + 1) \oplus \cdots \oplus V(f_k + 1) \oplus \begin{pmatrix} 0 \\ \ldots \\ 0 \end{pmatrix}$$

Since the $\begin{pmatrix} 0 \\ \ldots \\ 0 \end{pmatrix}$ occurs an uneven number of times, it remains in the final result and does not cancel. □

The basic monomial lemma holds for polynomials $f \in \mathcal{B}_n$ as well: In case of an uneven number of monomials in f, the basic monomial lemma extends immediately, and with it everything else.

In case of an even number of monomials in f, the basic monomial lemma loses the extra *xor*, making everything even simpler:

Corollary 1.2.10. *If $f = f_1 + \cdots + f_k$ contains an even number of monomials, it holds that:*

$$\log \circ V^{-1} \circ \log(f_1 + \cdots + f_k) = V(f_1 + 1) \oplus \cdots \oplus V(f_k + 1)$$

Proof. This follows from the basic monomial lemma. The $\begin{pmatrix} 0 \\ \ldots \\ 0 \end{pmatrix}$ cancel since they occur an even number of times. □

Corollary 1.2.11. *If f contains an even number of monomials, it follows that*

$$V^{-1} \circ \log(f) = \log^{-1}(V(f+1))$$

This finally brings us to the last bit of the proof:

Corollary 1.2.12. *The above diagram holds for all f with even number of monomials:*

Proof.

$$\begin{aligned} M \circ M \circ M \circ M(f) &= M \circ M \circ M \circ V^{-1} \circ \log(f) \\ &= M \circ M \circ M(\log^{-1}(V(f+1)))) \\ &= M \circ M \circ V^{-1} \circ \log(\log^{-1}(V(f+1))) \\ &= M \circ M \circ V^{-1}(V(f+1)) \\ &= M \circ M(f+1) \end{aligned}$$

Since $M \circ M(f) = f + 1$, it is clear that $M \circ M \circ M \circ M = f$. □

We're finally done, the square lemma is proved. □

1.2.2 Hadamard-Transform

We circle back to the exposition in [Carle07]. An important tool for studying Boolean functions for cryptography is the Walsh-Hadamard transform, which is equally commonly known as the discrete Fourier transform. For analysing the resistance of a Boolean function f to linear and differential attacks, it is important to know the weights of $f \oplus l$ (where l is an affine function) or $f(x) \oplus f(x+a)$ (where $a \in \mathbb{F}_2^n$). The discrete Fourier transform provides us with ways of measuring these weights.

The discrete Fourier transform maps any Boolean function f to another function $\widehat{f}$ by

$$\widehat{f}(u) = \sum_{x \in \mathbb{F}_2^n} f(x)(-1)^{x \cdot u}$$

Remark 1.2.13. Please keep in mind that $x \cdot u$ denots the vector product / inner product.

1.2.3 Walsh-Transform

The *Walsh-Transform* is defined quite similarly to the Hadamard-transform:

$$\widehat{f}_{\chi}(u) = \sum_{x \in \mathbb{F}_2} (-1)^{f(x) \oplus x \cdot u}$$

Remark 1.2.14. Please keep in mind that $x \cdot u$ denots the vector product / inner product.

These two transforms are of fundamental importance in cryptography - every SBox that is designed nowadays is designed so that these transform satisfy certain properties that rule out differential and linear attacks on the block cipher in question. For more details about the properties of these transforms, please refer to [Carle07].

1.3 The ring of Boolean functions as a lattice

Since the mappings V and V^{-1} and result 1.1.9 provide us with a bijection between $\mathcal{B}_n$ and $\mathbb{F}_2^n$, $\mathcal{B}_n$ receives the structure of a lattice through pulling the usual partial order $\subset$ on $\mathbb{F}_2^n$ back to $\mathcal{B}_n$. The $\wedge$ operator is given through 1.1.14, the $\vee$ operator is given through multiplication in $\mathcal{B}_n$.

Similarly, the mapping $\log^{-1}$ combined with $\subset$ puts a different lattice structure on $\mathcal{B}_n$.

This lattice structure will be used in section 6.3 to give an interpretation of an algorithm introduced in chapter 6.

1.4 The MQ problem

The exposition here follows [Daum01] closely.

Definition 1.4.1 (The MQ Problem). Let K be any finite field and $R := K[x_1, \ldots, x_n]$ the polynomial ring in n variables over this field. Let $f_1, \ldots, f_m \in R$ be quadratic polynomials, e.g. of the form

$$f_i = \sum_{1 \leq i \leq j \leq n} q_{ij} x_i x_j + \sum_{i=1}^{n} l_i x_i \text{ with } q_{ij}, l_i \in K$$

The problem MQ is the problem of finding at least one $x \in K^n$ with $\forall i \;\; f_i(x) = 0$.

Theorem 1.4.2 (MQ with $K = \mathbb{F}_2$ is NP-hard). *Proof.* Let (X, C) be an instance of $3SAT$ where $X = \{x_1, \ldots, x_n\}$ the set of variables and C the set of clauses.

Each clause is a disjunction of 3 literals, hence of the form $\tilde{x}_i \vee \tilde{x}_j \vee \tilde{x}_k$ with $\tilde{x}_l \in X \cup \overline{X}$. Such a clause is satisfied iff

$$\tilde{x}_i + \tilde{x}_j + \tilde{x}_k + \tilde{x}_i\tilde{x}_j + \tilde{x}_i\tilde{x}_k + \tilde{x}_j\tilde{x}_k + \tilde{x}_i\tilde{x}_j\tilde{x}_k = 1$$

It follows that each $3SAT$ instance can be transformed into a system of equations of degree less or equal 3. The degree of said system can be lowered to quadratic by introducing extra variables $\tilde{z}_{ij} := \tilde{x}_i\tilde{x}_j$. □

Theorem 1.4.3 (MQ with arbitary K is NP-hard)**.** *Proof.* See above. □

Chapter 2

Generating Equations

Before one can even attempt an algebraic attack, one has to first generate a system of equations that needs solving. Unfortunately, there are many different ways of generating equation systems from a given cipher. It appears that no 'universally accepted' method for generating the equations exists, and it appears to be much more common to publish analysis results than to publish the actual equation systems under consideration.

Furthermore, several choices during the process of generating equations influence later attempts at solving, and different researchers have taken different paths. This text discusses how [Armknecht] described the creation of equation systems for LFSR-based stream ciphers and a “naive” attempt at generating equations for the block cipher PRESENT.

2.1 Generating Equations for LFSR-based stream ciphers

The following section follows Frederik Armknecht's [Armknecht] dissertation very closely.

Definition 2.1.1 (LFSR)**.** A *linear feedback shift register* (LFSR) consists of

1. An internal state of n bits
2. A *feedback matrix* L of the shape

$$L := \begin{pmatrix} 0 & \dots & \dots & 0 & \lambda_0 \\ 1 & 0 & \dots & 0 & \lambda_1 \\ 0 & 1 & \dots & 0 & \lambda_2 \\ \dots & \dots & \dots & 0 & \dots \\ 0 & \dots & 0 & 1 & \lambda_{n-1} \end{pmatrix}$$

3. An index i to indicate which bit of the internal state to output. This gives rise to the vector $v := (\delta_{0,i}, \dots, \delta_{n,i})$ (where δ is the Kronecker symbol).

The k-th keystream bit s_k is generated as $s_k = S_0 L^k v$ where S_0 is the initial internal state.

Definition 2.1.2 ((ι, m)-Combiners)**.** A (ι, m)-combiner consists of

1. s LSFR with lengths $n_1, \dots, n_s$ and feedback matrices $L_1, \dots, L_s$. These feedback matrices form the matrix L as follows:

$$L := \begin{pmatrix} L_1 & \dots & 0 \\ \dots & \dots & \dots \\ 0 & \dots & L_s \end{pmatrix}$$

In the following $n := \sum_i n_i$

2. The internal state $S \in \mathbb{F}_2^m \times \mathbb{F}_2^{n_1} \times \dots \times \mathbb{F}_2^{n_s}$. These are the internal states of the individual LFSRs along with some memory to save bits of past states. The notation S_m will be used to refer just to the memory bits. $S_{m,t}$ will denote the state of the memory bits at clock cycle t.

3. A projection matrix P of size $n \times \iota$. This is used to select ι different bits from the internal states of the LFSRs.

4. A memory update function $\psi : \mathbb{F}_2^m \times \mathbb{F}_2^\iota \to \mathbb{F}_2^m$. This is used to update the state of the internal memory based on the previous state and the selected ι different bits.

5. An output function $\chi : \mathbb{F}_2^m \times \mathbb{F}_2^\iota \to \mathbb{F}_2$. This is used to combine the ι different bits selected using the projection matrix in order to have the result output as a keystream bit.

If $m \geq 1$, we call this a *combiner with memory*, else a *simple combiner*

The keystream is generated as follows: The initial state is initialized by an element $(S_{m,0}, K)$ of $\mathbb{F}_2^m \times \mathbb{F}_2^n$ - the first component is used to initialize the memory bits S_m, the second is used to initialize the internal state of the LSFR. At each clock t, the following happens now:

$$\begin{aligned} z_t &\leftarrow \chi(S_{m,t}, KL^tP) && \text{The output bit is calculated} \\ S_{m,t+1} &\leftarrow \psi(S_{m,t}, KL^tP) && \text{The memory bits are updated} \end{aligned}$$

2.1.1 Generating equations for simple combiners

Equations for simple combiners can be generated more or less trivially: One simply collects the output bits of the combiner, with a resulting equation system of the form

$$z_t = \chi(KL^tP)$$

This equation system can be made arbitrarily overdefined by collecting more equations.

2.1.2 Generating equations for combiners with memory

Setting up equations for combiners with memory is a little more convoluted. One cannot just collect equations of the form

$$z_t = \chi(S_{m,t}, KL^tP)$$

since the contents of $S_{m,t}$ are not known, and thus no clear solution could be calculated. But the $S_{m,t}$ are not independent: One could generate equations that describe $S_{m,t+1}$ in terms of $S_{m,t}$ and KL^tP.

In [Armknecht], a different path is chosen: Since the goal in that case is using a relinearization procedure for solving the resulting system, the high degree of the output equations that occurs if the memory states are “naively” modeled needs to be avoided. Instead, equations over several clocks of the combiner are generated by use of a device called r-functions:

Definition 2.1.3 (r-function)**.** Given a (ι, m)-combiner and a fixed r, a function F is an r-function for the combiner if

$$F(X_1, \ldots, X_r, y_1, \ldots, y_r) = 0$$

holds whenever $X_1, \ldots, X_r$ and $y_1, \ldots, y_r$ are successive inputs respective outputs of the output function χ.

Given an r-function, equations can then be generated over series of keystream bits: If $z_1, \ldots, z_n$ are the keystream bits (and n is some multiple of r), the equation system would be of the form

$$\begin{array}{rcl} F(X_1, \ldots, X_r, z_1, \ldots, z_r) & = & 0 \\ \ldots & = & \ldots \\ F(X_{n-r}, \ldots, X_n, z_{n-r}, \ldots, z_n) & = & 0 \end{array}$$

The use of r-functions has a geometric interpretation (and might have some unclear consequences on the equation system solving) which is discussed in 6.4.2.

2.1.3 Z-functions

It turns out that there are a few complications to using r-functions right away. In order to deal with these complications, [Armknecht] introduces another device called Z-functions:

Definition 2.1.4 (Z-function)**.** Let $Z \in \mathbb{F}_2^r$, $K_t \in \mathbb{F}_2^\iota$ the inputs of χ at clock t, and z_t the output bits. A function $F : \mathbb{F}_2^{\iota+r} \to \mathbb{F}_2$ is a Z-function if

$$\forall t \quad Z = (z_t, \dots, z_{t+r}) \Rightarrow F(K_t, \dots, K_{t+r-1}) = 0$$

A Z-function can be thought of as a *concretization* of r-functions: One single r-function can give rise to many different Z-functions. One can think of Z-functions as a device to make "better" choices for the equations to be generated based on the observed outputs from the combiner. Similarly to r-functions, Z-functions can be put into a more geometric framework. This is done in 6.4.2.

2.2 Generating Equations for block ciphers

The situation for generating equations for block ciphers is a little bit trickier and requires some work to be invested. Before we start actually generating any equations, we begin with an algebraic description of a block cipher.

2.2.1 Algebraic description of an SPN

Let $\mathcal{M} := \mathbb{F}_2^m$ be the message space, $\mathcal{K} := \mathbb{F}_2^c$ the key space. A mapping $\mathcal{C} : \mathcal{M} \times \mathcal{K} \to \mathcal{M}$ is called a block cipher. Most modern constructions can be described as follows:

Let $K : \mathbb{F}_2^c \to (\mathbb{F}_2^m)^r$ be the key-schedule and $R : (\mathbb{F}_2^m)^2 \to \mathbb{F}_2$ the round function. The key schedule is used to expand a single key into many *round keys*, and the round function is then used to iteratively combine the individual round keys with the message:

$$\mathcal{M} \times \mathcal{K} \xrightarrow{id \times K} (\mathbb{F}_2^m)^{r+1} \xrightarrow{R \times id^{r-1}} (\mathbb{F}_2^m)^r \xrightarrow{R \times id^{r-2}} \dots \xrightarrow{R} \mathbb{F}_2^m = \mathcal{M}$$

This diagram is best read like this: The initial mapping $id \times K$ expands the key and leaves the message unchanged. The subsequent $R \times id^{r-1}$-mapping combines the first round key with the message bits, leaving the other round keys unchanged. The next mapping $R \times id^{r-2}$ adds the next round key to the result of the previous mapping, and so forth.

Normally, R consists of the composition of three individual mappings: A *key addition* $+ : (\mathbb{F}_2^m)^2 \to \mathbb{F}_2^m$, a *substitution* $S : \mathbb{F}_2^m \to \mathbb{F}_2^m$ and a *permutation* $P : \mathbb{F}_2^m \to \mathbb{F}_2^m$.

In total, the following diagram emerges:

$$\begin{tikzcd}
 & (\mathbb{F}_2^m)^r \arrow[r, "\mathrm{S}\times id^{r-1}"] & (\mathbb{F}_2^m)^r \arrow[d, "\mathrm{P}\times id^{r-1}"] & \dots \arrow[r, "S"] & \mathbb{F}_2^m \arrow[d, "P"'] \\
\mathcal{M}\times\mathcal{C} \arrow[r, "id\times K"] & (\mathbb{F}_2^m)^{r+1} \arrow[u, "+\times id^{r-1}"] \arrow[r, "R\times id^{r-1}"] & (\mathbb{F}_2^m)^r \arrow[r, "R\times id^{r-2}"] \arrow[d, "+\times id^{r-2}"'] & \dots \arrow[r, "R"] & \mathbb{F}_2^m = \mathcal{M} \\
 & & (\mathbb{F}_2^m)^{r-1} \arrow[r, "S\times id^{r-2}"] & \dots &
\end{tikzcd}$$

2.2.2 Naive attempts at generating equations

Theoretically, it is possible to express the entire cipher as a set of n Boolean polynomials in the variables $m_1, \dots, m_n$ (the message bits) and $k_1, \dots, k_c$ (the key bits). The letters m and k denote these bits as Boolean vectors of length n (e.g. the blocksize of the cipher) and respective c (the key length of the cipher). This would yield a system of the form:

$$\begin{aligned}
f_1(m_1, \dots, m_m, k_1, \dots, k_c) &= C(m,k)_1 \\
\dots &= \dots \\
f_m(m_1, \dots, m_m, k_1, \dots, k_c) &= C(m,k)_n
\end{aligned}$$

For each known plaintext / ciphertext pair, one could then "insert" the known bits into above polynomials, and thus generate polynomials in nothing but the key variables. Given, for example, 100 known-plaintext/ciphertext pairs, one would obtain $100n$ polynomials in the key bits. Solving this system of polynomials would yield the key.

Unfortunately, in practice things do not work this way:

A good cipher should in essence generate almost-perfectly-random polynomials in the key and plaintext bits. We have seen in previous chapters that each polynomial can be thought of as a subset of the solution space. From this it follows that each f_i is an almost-perfectly-random element of $\mathfrak{P}(\mathbb{F}_2^{n+c})$.

This leads to the following problem: Half of all such elements have more than 2^{n+c-1} monomials. Given that a monomial in $n + c$ variables will (naively stored) take at least $n + c$ bits of storage, we see that the computational effort required to even *write down* the equations far outstrips the computational effort for brute forcing a few keys.

It will become evident shortly that these "theoretical" considerations do come into full effect when we try to generate equations for a block cipher.

2.2.3 Generating equations for PRESENT

At CHES2007, a particularly lightweight substitution-permutation network was proposed: PRESENT. The design goal for this block cipher was the to minimize the gate count required to implement it.

Due to it's particularly simple design, it makes for an appealing target for generating equations: The hope is that the extremely restricted design will simplify analysis. One could argue that PRESENT is the "simplest" strong block cipher we know at the moment.

Description

PRESENT is a simple SPN with 64-bit block size and an 80-bit key, iterated over 32 rounds.

The key schedule

The 32 round keys of width 64 are generated as follows: A key register $K = k_0, \ldots, k_{79}$ is initialized with the 80 key bits. The first round's key is initialized with bits 16 to 79. After each round, the key register is rotated left by 61 bits, and the high bits, 76 to 79, are passed through the SBox of the cipher (see below). Bits 19 to 15 are then XOR'ed with the round counter. This is explicitly described below:

```
Data: Input key $K = k_0, \ldots, k_{79}$
Result: Set of round keys $K_0, \ldots, K_{31}$ of 64 bit width
foreach Round $r \in \{0, \ldots 31\}$ do
    $K_r \leftarrow k_{79} \ldots k_0$;
    $K \leftarrow K << 61$;
    $K \leftarrow S(k_{79}, k_{78}, k_{77}, k_{76}), \ldots k_0$;
    $K \leftarrow k_{79}, \ldots, k_{19} + r_4, k_{18} + r_3, \ldots k_{15} + r_0, \ldots, k_0$;
end
```

Algorithm 1: The PRESENT key schedule

Key addition

Key addition is a regular 64-bit exclusive-or of the current cipher state with the round key.

The 4-bit SBox

PRESENT uses a single 4-bit SBox for all computations:

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

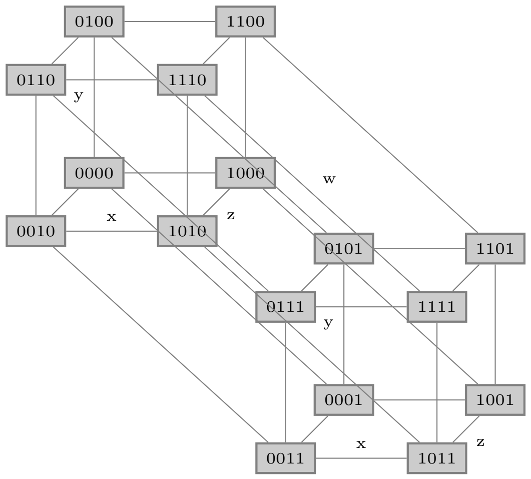
This 4-bit SBox can be represented by the following equations:

$$\begin{aligned}
y_0 &= x_0 + x_2 + x_1x_2 + x_3 \\
y_1 &= x_1 + x_0x_1x_2 + x_3 + x_1x_3 + x_0x_1x_3 + x_2x_3 + x_0x_2x_3 \\
y_2 &= x_0x_1 + x_2 + x_3 + x_0x_3 + x_1x_3 + x_0x_1x_3 + x_0x_2x_3 + 1 \\
y_3 &= x_0 + x_1 + x_1x_2 + x_0x_1x_2 + x_3 + x_0x_1x_3 + x_0x_2x_3 + 1
\end{aligned}$$

According to [Present], the SBox has been chosen to provide strong resilience towards differential and linear cryptanalysis, while also being cheap to implement in hardware.

Due to the low dimensionality, one can draw diagrams for the solution sets of these equations. This can help in building "geometric intuition" about the equations.

The diagrams are to be read as follows: $\mathbb{F}_2^3$ can be visualized using a standard cube consisting of the points $(x, y, z) \in \mathbb{F}_2^3$ – every point of $\mathbb{F}_2^3$ thus corresponds to the "corner" of this cube embedded in $\mathbb{R}^3$. We can thus visualize subsets of $\mathbb{F}_2^4$ as two cubes whose corners have been connected. For easier explanation, this is a picture of $\mathbb{F}_2^4$ (e.g. the coordinates of the corners have been drawn into the diagram):



In order to visualize the Boolean functions corresponding to the four equations describing the PRESENT SBox, the points on which these functions vanish have been colored in green in these diagrams. The diagrams in figure 2.2.3 are interesting from another perspective: They help in understanding under which conditions the Raddum-Semaev algorithm succeeds or fails in solving equations (see chapter 4).

The permutation layer

The permutation layer in PRESENT can be described by the following lookup table:

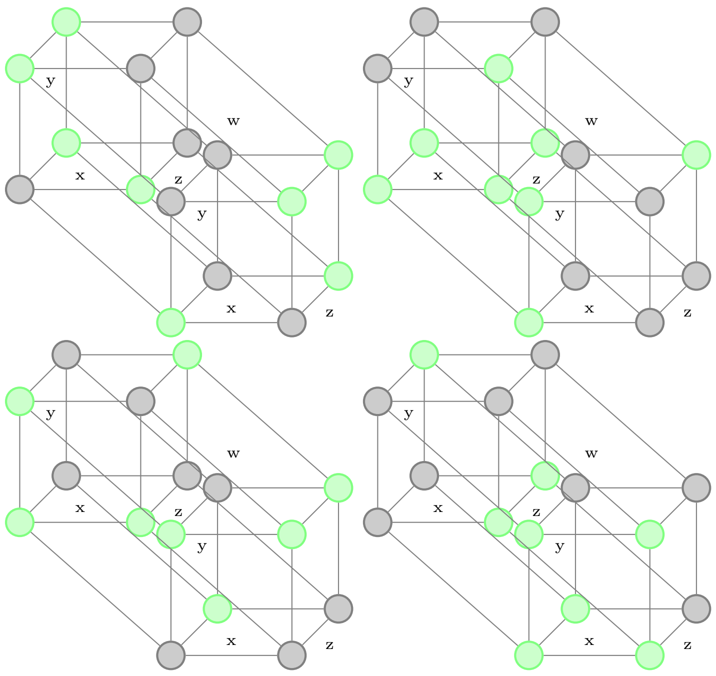


Figure 2.1: The varieties for the equations of the PRESENT SBox.

0	16	32	48	1	17	33	49	2	18	34	50	3	19	35	51
4	20	36	52	5	21	37	53	6	22	38	54	7	23	39	55
8	24	40	56	9	25	41	57	10	26	42	58	11	27	43	59
12	28	44	60	13	29	45	61	14	30	46	62	15	31	47	63

Generating equations

A symbolic implementation of PRESENT was built in C++.

Boolean polynomials are represented as `std::set`'s of Boolean monomials, which are in turn arrays of 32-bit integers. Memory consumption was minimized by the use of template parameters to determine the required size of Boolean monomials at compile time.

The code allows for different ways of generating equations: Running the command

```
./BooleanBench PRESENTFULL 3 80 0
```

will generate equations that describe 3 rounds of 80-bit PRESENT with a "weakening" parameter of 0. In this mode, the plaintext bits will be kept variable, too, so the generated equations will be equations in the variables $x_0, \ldots, x_{79}$ (for the key bits) and $x_{80}, \ldots, x_{144}$ (for the plaintext bits).

Unfortunately, any attempt to calculate these equations for more than 3 rounds ran out of available memory on the test machine (4 GB). The sizes for 2 and 3 rounds can be read from figure 2.2.3 and figure 2.2.3.

As a next step, code was implemented that applies PRESENT with variable key bits to a given *fixed* plaintext. The command

```
./BooleanBench PRESENT FFFFFFFFFFFFFFFF 3 80 0
```

will calculate equations for 3 rounds of present with 80 bit key over the plaintext 64-bit value 0xFFFFFFFFFFFFFFFF.

While this significantly lowered the sizes of the polynomials after 3 rounds (from a maximum of about 150000 to about 3000), calculating equations for 4 full rounds remained infeasible without more memory – the machine ran out of available RAM about $\frac{1}{3}$ into the substitution layer of the fourth round.

From this data, we can observe the following:

- It appears that the complexity of the polynomial representation of about $\frac{1}{4}$ of the bits differs drastically after 3 rounds. This is probably due to the diffusion not being complete after 3 rounds yet.
- Each round seems to increase the complexity of the polynomial representation of the worst-case output bits by approximately factor 1000
- With the current implementation, generating equations for more than 3 rounds of PRESENT appears infeasible.

The situation is quite unsatisfactory: Three rounds really do not amount to much. In order to gain a better understanding, we weaken the cipher in several steps in order to generate more manageable equation systems.

2.2.4 Weakening to 40 bits

As a first step, we weaken the cipher by fixing all uneven key bits to equal zero. The resulting cipher allows us the computation of equations for 4 full rounds if we fix an arbitrary plaintext. While this is more than the 3 rounds we can calculate without weakening, we nonetheless exhaust available memory in round 5. Please see figure 2.2.4 for the sizes of the generated polynomials.

Weakening to 26 bits

Since 40 bits still prove to be too much to generate explicit equations, we weaken the cipher to 26 bits. This weakening was done in a very similar way to the weakening to 40 bits: Instead of fixing every uneven bit to zero, we set all key bits whose indices are not divisible by three to zero.

While generating equations was feasible, the time requirements exceeded those available to the author.

Weakening to 20 bits

Weakening the cipher to 20 bits allowed the calculation of intermediate-free equation systems for more than 5 rounds. At the time of the submission of this thesis, the calculation of equation systems for round 6 was still running. The weakening was done by setting all key bits whose indices are not divisible by four to zero.

The development of the equation sizes can be viewed in the figures following 2.2.4.

Weakening to 16 bits

The cipher was also weakened to 16 bits. All key bits whose indices are not divisible by 5 are fixed to equal zero. Weakening the cipher to 16 bits allowed the calculation of intermediate-free equation systems up to round 5. At the time of the submission of this thesis, the calculation of equation systems for round 6 was still running. The sizes of the equations for 4 and 5 rounds can be viewed in the figures following 2.2.4.

2.2.5 The question of intermediate variables

As visible in the above, generating Boolean equations in nothing but the key (not even to speak of key and plaintext variables) can easily prove infeasible – the sizes of the polynomials quickly approach the expected size of 2^{c-1} monomials, making any operations on them as expensive as exhaustive search. In practice, *intermediate variables* are introduced at strategic points in the cipher as to reduce the size of the equations that are generated. This is usually done by creating new intermediate variables after each round and assigning the result of the round to these intermediate variables.

This clearly carries a disadvantage: It implies that the number of variables that need to be managed grows, and with it the size of the polynomials that need to be represented during all computations. It is unclear what the introduction of intermediate variables means for the generated equation system. While it certainly helps in writing the equation down, it also increases the dimensionality of the problem significantly. Furthermore, it increases the storage requirements for storing individual monomials – and as we have seen, the number of monomials in our intermediate results can grow quite drastically, therefore even small increases in monomial size can have significant impact on overall memory consumption.

2.2.6 The geometry of introducing variables

It is quite unclear what the introduction of new variables does to our equation systems, and specifically to the “geometry” of our solution sets. It would be worthwhile to investigate these questions more thoroughly, specifically the relation between the geometry of introducing new variables and the effects of the Raddum-Semaev algorithm (which might to be the "inverse" of this).

2.3 Summary

It appears that the process of generating equations for a cipher might actually be the key component in algebraic attacks: Naively generated equation systems exhaust available memory just *writing them down* - not to mention solving them. Without sophisticated methods to pre-process the equation systems coming from ciphers (such as Z-functions and low- degree annihilators), performing any algebraic attack appears pretty much hopeless. In 6.4.1, the different methods will be unified and a common theme will emerge: Algebraic attacks need a form of *approximation* in order to be fruitful.

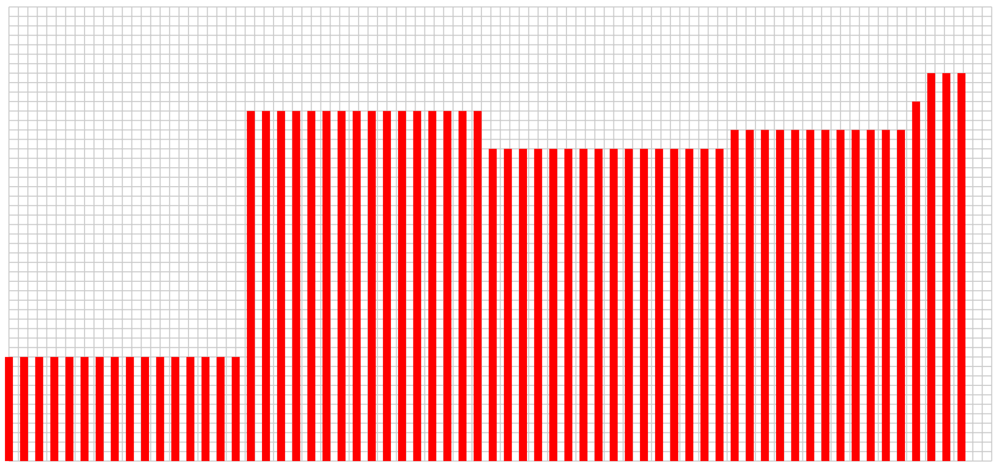


Figure 2.2: Equation sizes for all 64 output bits of full PRESENT with plaintext and key bits kept variable. # of monomials after 2 rounds. 1 square = 1 monomial

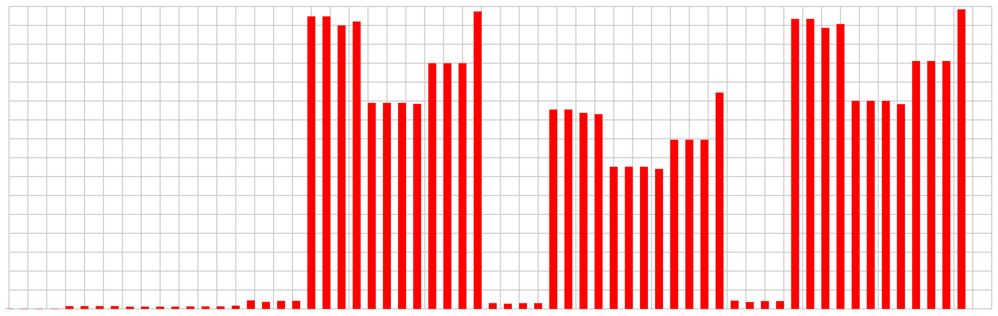


Figure 2.3: Equation sizes for all 64 output bits of full PRESENT with plaintext and key bits kept variable. # of monomials after 3 rounds. 1 square = 10000 monomials. One can see that diffusion is still weak: Several bits have very low monomial counts.

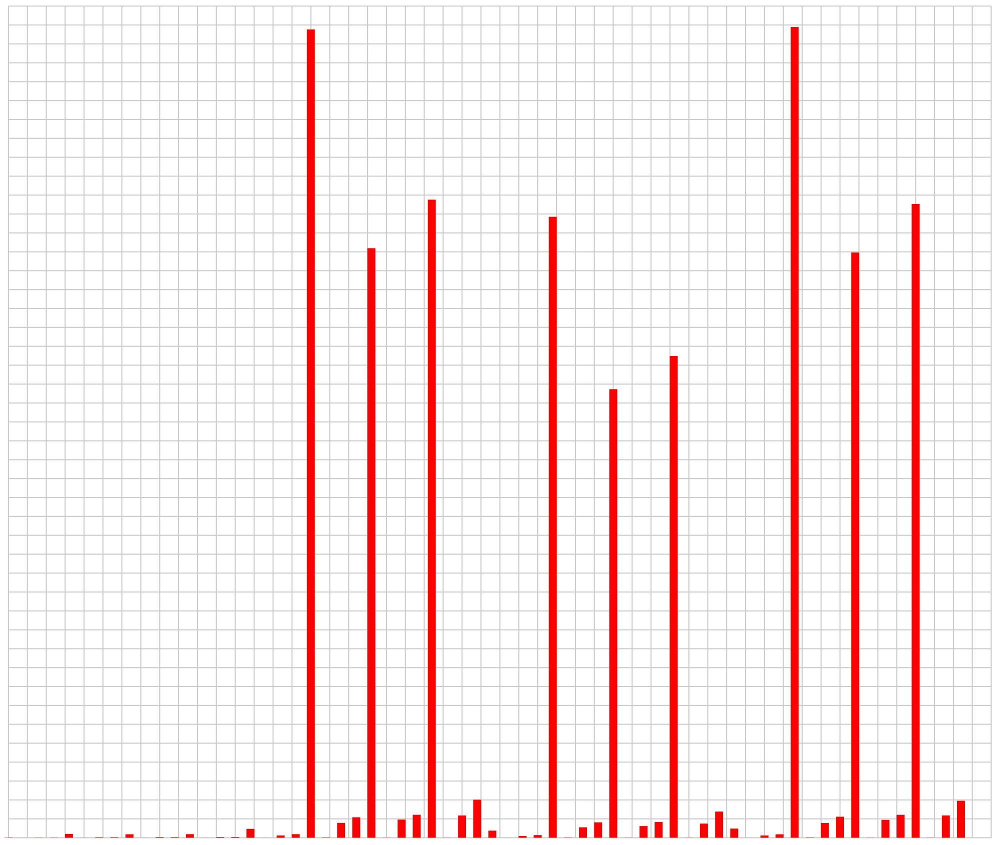


Figure 2.4: Equations sizes for all 64 bits of PRESENT output after 4 rounds. The cipher was weakened by setting half of all keybits to zero. 1 square = 10000 monomials. It appears that weakening the cipher in this manner severely impacts diffusion.

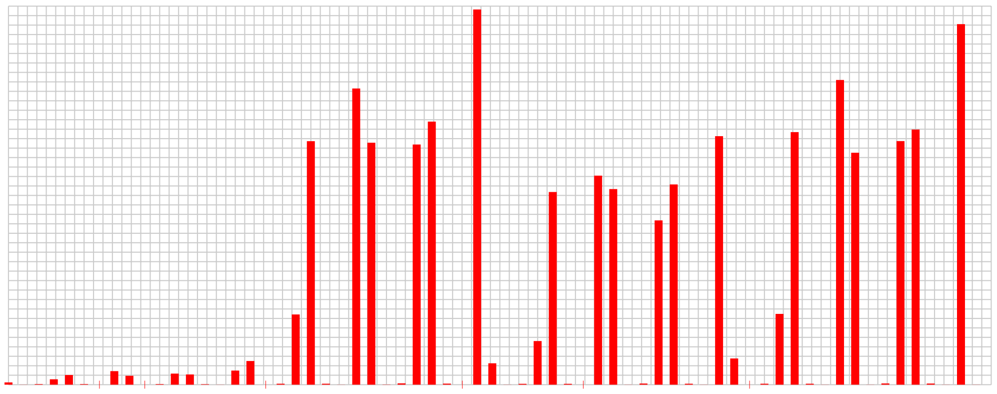


Figure 2.5: # of monomials after 4 rounds of 20-bit PRESENT. 1 square = 100 monomials

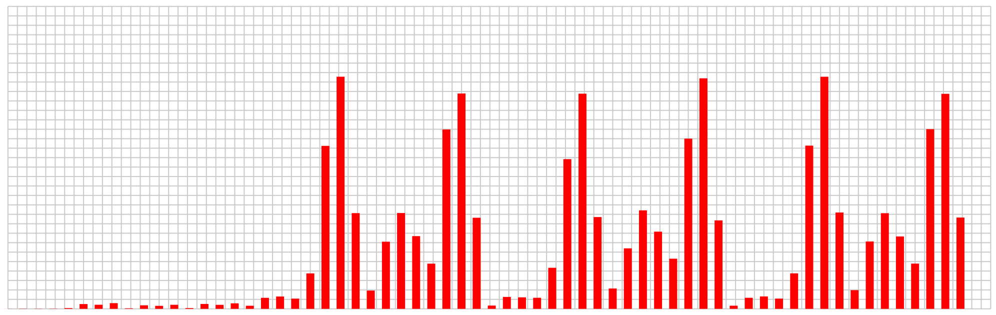


Figure 2.6: # of monomials after 5 rounds of 20-bit PRESENT. 1 square = 10000 monomials

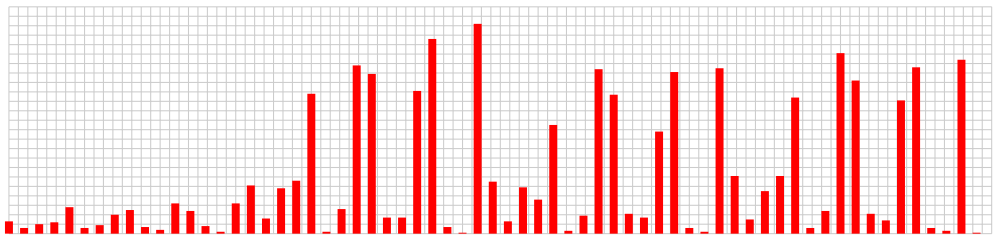


Figure 2.7: # of monomials after 4 rounds of 16-bit PRESENT. 1 square = 10 monomials

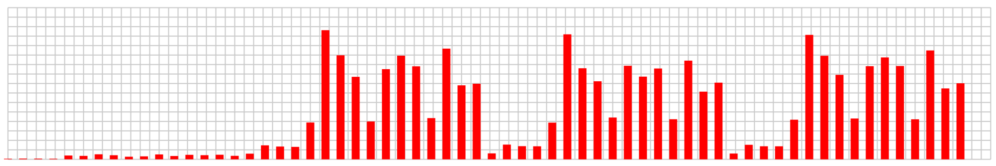


Figure 2.8: # of monomials after 5 rounds of 16-bit PRESENT. 1 square = 1000 monomials

Chapter 3

Gröbner basis and the Buchberger Algorithm

3.1 Introduction

The "workhorse" of large parts of computational commutative algebra nowadays is certainly the concept of a *Gröbner basis*. Aside from being a very general tool (which works over an arbitrary multivariate polynomial ring over arbitrary fields), a Gröbner basis allows much more than "just" the solving of multivariate polynomial equation systems. While this thesis will not go into too much depth, a brief overview of the topic will be given.

Working with multivariate polynomial systems is generally more difficult than working with univariate equations: In the univariate case, the univariate polynomial ring happens to be Euclidian, resulting in the availability of the Euclidian algorithm. This algorithm allows us, amongst other things, to

- Given a set of generators $g_1, \dots, g_r$ of an ideal $I \subset K[X]$, calculate $f \in K[X]$ so that $\langle f \rangle = \langle g_1, \dots, g_r \rangle = I$.
- Calculate a normal form for each $f \subset K[X]/I$ by simply dividing by f.

This clearly doesn't work in the multivariate case, as multivariate polynomial rings are not principal ideal domains and as such not Euclidian. In order to be able to solve question such as *submodule membership* or *ideal membership* as well as for 'normalized' calculations in $K[X]/I$, one needs a method that allows one to calculate a *normal form* of elements in $K[X]/I$.

While this section will trail [KreuRobb] closely, some simplifications in exposition will be made: The entire construction of Gröbner basis can be built for arbitrary modules over multivariate polynomial rings, yielding more powerful theory at the cost of more complex notation. The more general construction is not needed in most applications of Gröbner Bases in cryptography, and therefore this exposition restricts the 'arbitrary module' to 'ideal'. This is also the path chosen in [AdamsLou]. Proofs are almost completely omitted, the interested reader is referred to [AdamsLou] and [KreuRobb].

3.1.1 Term Orders

Definition 3.1.1 ($\mathbb{T}$)**.** Let $\mathbb{T}$ be the set of monomials of $\mathbb{F}[x_1, \ldots, x_n]$, e.g. the set of monomials in a multivatiate polynomial ring over $\mathbb{F}$. This set has a natural monoid structure through multiplication of monomials.

Definition 3.1.2 (Term Order)**.** A term order is a total relation on $\mathbb{T}^n$ that satisfies for $t_1, t_2, t_3 \in \mathbb{T}^n$:

1. $t_1 \geq t_1$
2. $t_1 \geq t_2, t_2 \geq t_3 \Rightarrow t_1 \geq t_3$ (Transitivity)
3. $t_1 \geq t_2, t_2 \geq t_1 \Rightarrow t_1 = t_2$ (Antisymmetry)
4. $t_1 \geq t_2 \Rightarrow t_1 t_3 \geq t_2 t_3$
5. $t_1 \geq 1$

3.1.2 Macaulay's Basis Theorem

It was discussed earlier that one would like to calculate a set of ideal generators so that a normal form of elements in $\mathbb{F}[x_1, \ldots, x_n]$ can be constructed easily. In order to do so, the following theorem is of fundamental importance:

Theorem 3.1.3 (Macaulay's Basis Theorem)**.** *Let $\mathbb{F}$ be a field and $P = \mathbb{F}[x_1, \ldots, x_n]$ a polynomial ring over $\mathbb{F}$. Let $M \subseteq P^r$ be a P-submodule and let σ be a module term ordering on $\mathbb{T}^n\langle e_1, \ldots, e_r\rangle$. Then the residue classes of the elements of $\mathbb{T}^n\langle e_1, \ldots, e_r\rangle \backslash LT_\sigma\{M\}$ form a basis of the $\mathbb{F}$-vector space P^r/M.*

The Macaulay Basis Theorem tells us that the $\mathbb{F}$ vector space P^r/M has as basis all those elements of $\mathbb{T}^n\langle e_1, \ldots, e_r\rangle$ that cannot occur as leading terms of any elements of M. If we translate this theorem into an ideal-centric notation, it reads as follows:

Theorem 3.1.4 (Macaulay's Basis Theorem, simplified)**.** *Let $\mathbb{F}$ be a field and $P = \mathbb{F}[x_1, \ldots, x_n]$ a polynomial ring over $\mathbb{F}$. Let $M \subseteq P$ be a an ideal in P and σ be a term ordering on $\mathbb{T}^n$. Then the residue classes of the elements of $\mathbb{T}^n \backslash LT_\sigma\{M\}$ form a basis of the $\mathbb{F}$-vector space P/M.*

Unfortunately, these theorems are a bit nonconstructive – the basis of P/M is in general infinite, and we know of no "nice" ways of calculating $LT_\sigma\{M\}$ yet.

3.1.3 Multivariate Division

Similarly to the Euclidian division algorithm, a multivariate division algorithm can be defined as follows:

```
**Data**: $m \in K[X], (g_1, \dots, g_r) \in K[X]^r$ generators of the ideal
**Result**: $q_1, \dots, q_r, r$ so that $m = (\sum q_i g_i) + r$
$v \leftarrow m$;
$q_1 = \dots = q_r = r = 0$;
**while** $v \neq 0$ **do**
    Find first index $i$ with $LM(g_i)$ divides $LM(v)$;
    **if** *such an index exists* **then**
        $q_i \leftarrow q_i + \frac{LM(v)}{LM(g_i)}$;
        $v \leftarrow v - \frac{LM(v)}{LM(g_i)} g_i$;
    **else**
        $r \leftarrow r + LM(v)$;
        $v \leftarrow v - LM(v)$;
    **end**
**end**
```

The trouble with this division algorithm is that the result r depends on the order in which the elements $g_1, \dots, g_r$ are arranged, and not every element of $I := \langle g_1, \dots, g_r \rangle$ reduces to zero using this algorithm.

Why is this so ? One of the fundamental properties of univariate polynomial rings is the fact that $\deg(fg) \geq \max\{\deg(f), \deg(g)\}$ and specifically $LT(f)|LT(fg), LT(g)|LT(fg)$. This does not hold in multivariate polynomial rings - it is easily possible that the leading terms cancel, thus yielding a situation where $LT_\sigma(f) \not| LT_\sigma(fg), LT_\sigma(g) \not| LT_\sigma(g)$. In such a situation, even though fg is clearly divisible by both f and g, the described algorithm would not reduce fg any further.

By implication, if both f and g were generators of an ideal, fg would be an element of the ideal. But since the multivariate division algorithm would not reduce to zero, in it's current form it would not be usable to calculate normal forms for elements of P/M.

This leads us to the first of a number of descriptions for Gröbner basis:

Definition 3.1.5 (Gröbner Basis, Definition 1). A set $g_1, \dots, g_r \in I \subseteq P$ forms a Gröbner basis for an ideal I with respect to a term ordering σ if for each element $f \in I$ there exists an index i so that $LT_\sigma(g_i)|LT(f)$.

Definition 3.1.6 (Gröbner Basis, Definition 2). A set $g_1, \dots, g_r \in I \subseteq P$ forms a Gröbner basis for an ideal I with respect to a term ordering σ if each element $f \in I$ reduces to zero when divided by $g_1, \dots, g_r$ using the multivariate polynomial division algorithm.

3.1.4 S-Polynomials

It is evident that pairs of polynomials that can be combined in a way so that the leading term gets cancelled are the "culprit" that causes the difficulties with the multivariate division. Such pairs of polynomials (that can be combined in a way that the leading terms cancel, and the remainder does not reduce to zero) can be characterized.

Definition 3.1.7. The S-Polynomial of f and g is defined as

$$spoly(f,g) = LC(g)\frac{lcm(LT(g),LT(f))}{LT(f)}f - LC(f)\frac{lcm(LT(g),LT(f))}{LT(g)}g$$

In essence, *spoly* is a construct where the leading terms of f and g are changed in such a way (without "leaving" the ideal from which f and g originate) so that the leading terms of the results *cancel* through the subtraction.

It is useful to note the following properties of *spoly*:

1. If f, g are in an ideal I, then so is $spoly(f,g)$
2. If $spoly(f,g)$ does not reduce to zero, it will have a different LT_σ from f and g

These properties lead directly to the *classical Buchberger algorithm*:

3.2 The classical Buchberger Algorithm

Let $G := \{g_1, \ldots, g_r\}$ be generators for the ideal M. Let $LT_\sigma(G)$ be the set of leading terms of these polynomials. The classical Buchberger algorithm is based on the following principle: Since P is Noetherian, $LT_\sigma(M)$ is finitely generated. Each time an $spoly(g_i, g_j)$ is calculated that does not reduce to zero using the multivariate division algorithm and G, the result is a new polynomial g_{r+1} that is an element of M and *also* has a leading term that is different from those in $LT_\sigma(G)$. By adding this polynomial to G, we move $LT_\sigma(G)$ "closer" to a full generator of $LT_\sigma(M)$. By iterating this process, we eventually end up with a G so that $LT_\sigma(G)$ is a generator for $LT_\sigma(M)$. Once this has been achieved, G is a Gröbner basis. The algorithm can be read in a (slightly) more formal notation in figure 3.2.

3.2.1 Interpretation: An almost purely syntactic algorithm

To understand the genericity of the Buchberger Algorithm, it is important to realize that it is, at it's core, a purely *syntactic* algorithm: Actual properties of the the base field are not used, only purely symbolic properties of the polynomial ring in which the calculations are performed.

```
Data: Set of polynomials $g_1 \dots g_r$
Result: Set of symbols $g'_1, \dots, g'_k$
$G \Rightarrow \{g_1, \dots, g_r\}$;
foreach unordered pair of polynomials $g_i \in G, g_j \in G, g_i \neq g_j$ do
    calculate $spoly(g_i, g_j)$;
    reduce $spoly(g_i, g_j)$ via multivariate division and $G$ to $g'_{ij}$;
    if $g'_{ij} \neq 0$ then
        $G = G \cup g'_{ij}$;
    end
end
return $G$;
```

Algorithm 2: The classical Buchberger algorithm. Please not that adding elements to G will imply more pairs to iterate over.

A possible way to understand the workings of this algorithm is, again, lattice-theoretic:

Fix an arbitrary base field. Consider the set of all multivariate polynomials ordered by some σ, with normalized leading term (e.g. the coefficient of the LT is 1). The divisibility relations between the leading terms provide this set with a partial order and a lattice structure. Call this lattice $\mathcal{L}$.

The given generators of the ideal, $g_1, \dots, g_r$, correspond to points on $\mathcal{L}$. Each time a new element g' is added to G, it is an element of the ideal $I := \langle g_1, \dots g_r \rangle$ that is "minimal" with respect to the rest of G on $\mathcal{L}$. This means there is no $g_i \in G \backslash \{g'\}$ with $g_i \leq g'$.

Step by step, the Buchberger algorithm thus transforms G. Once the algorithm terminates (and the result is interreduced, see [AdamsLou] or [KreuRobb]), G is the set of "minimal" elements on $\mathcal{L}$ that belong to I. In short: The Buchberger calculates the "minimal" elements of I on $\mathcal{L}$.

Because $\mathcal{L}$ "inherits" Noetherianess from P, the result is finite and the algorithm eventually terminates. How many steps are involved in tranforming G is quite unclear though.

The fact that the Buchberger algorithm is almost purely syntactic is its greatest strength: It can be implemented using little knowledge of the underlying field and will work without much adaption. It can also be generalized to other situations, e.g. having a principal ideal domain or even a general commutative ring as base structure. There has even been work on extending the algorithm to noncommutative scenarios.

At the same time, this generality raises the nagging suspicion that different algorithms, specialized to one particular field, or to the systems of equations of interest in cryptography, might exist and perform better.

3.3 Maximum sizes of polynomials in the computation

The running times of Gröbner-basis calculations are notoriously difficult to estimate. While the worst-case running times can in theory be doubly exponential (and in most cryptographic situations singly exponential), it has occured many times in practice that equation systems ended up being much easier to solve.

A small fact is easily overlooked when complexity estimates for Gröbner basis calculations are made:

If a canonical (or naive) representation of the polynomials is chosen (e.g. each monomial a string of bits, each polynomial a collection of such bit strings), the cost of representing intermediate results can become substantial. Polynomials stored in this manner can grow to be quite significant in size: a degree k-polynomial in n variables can have up to

$$\sum_{i=1}^{k} \binom{n}{i}$$

monomials. This means, for example, that in a 128-variable system in which intermediate results of degree up to 5 can occur, an intermediate result can contain up to 275584032 monomials. With each monomial costing (at least) 128 bit to represent (e.g. 4 32-bit words, 16 bytes), representing one such polynomial in memory can cost up to 4409344512 bytes of storage (e.g. roughly 4 GB). The situation grows worse quickly: If we reach degree 6, the worst-case storage requirements are in the order of 90 GB per polynomial. Even if the number of steps, measured in polynomial additions, is limited, one still has to keep in mind the cost of storage, *and* the fact that even linear operations (such as polynomial addition) will now cost a few billion cycles *each.*

This implies that this canonical representation of polynomials is far from ideal - it is unclear ,though, which representation would be superior in most cases. [BrickDrey07] uses a specialized data structure called *zero-suppressed binary decision diagrams* (ZBDD) which offer much-improved efficiency for many polynomials, but results are known that most Boolean functions will have an exponential-size ZBDD-representation, too.

Chapter 4

The Raddum-Semaev-Algorithm

Most results in this chapter are the outcome of a cooperative meeting with *Martin Albrecht* and *Ralf-Phillip Weinmann* [Lesezirkel].

The Raddum-Semaev-Algorithm was introduced in [RadSem06] and was able to solve some systems significantly faster than existing Gröbner-Basis algorithms. The paper is written from a computer-science perspective, which models the problem of simultaneously solving a multivariate polynomial equation system as a problem of 'passing messages on a graph'.

The exposition which focuses on the graph obfuscates the algebraic-geometric meaning of the algorithm: It is much more intuitive to represent the algorithm as a combination of projection of varieties to lower-dimensional spaces, intersecting such varieties, and then lifting the results to higher dimensions again.

In the following, the algorithm will first be explained along the same lines as the original paper did. After that, a more algebraic/geometric interpretation will be developed.

4.1 The original description

Let $f_1, \ldots, f_r \in \mathcal{B}_n$ be the set of equations that are supposed to be solved.

Definition 4.1.1 (Variable Map)**.** Let $\text{vars} : R_n \to \mathfrak{P}(\{x_1, \ldots, x_n\})$ be a mapping that maps a polynomial to the set of variables that occur in this polynomial.

Definition 4.1.2 (RS-Assumption)**.** Assume that $|\text{vars}(f_i)| <= k$ for some small k. This k needs to be small enough so that performing 2^k evaluations of f_i is feasible, and storing $r2^k$ bit strings is feasible.

```
Data: Set of symbols $S_1 \dots S_k$
Result: Set of symbols $S'_1 \dots S'_k$
while last iteration deleted a configuration do
    foreach pair of symbols $S_i, S_j$ do
        calculate $X_{ij}$;
        calculate $L_{ij} \cap L_{ji}$;
        Delete configurations from $S_i, S_j$ that do not cover any
        element of $L_{ij} \cap L_{ji}$;
    end
end
```

Algorithm 3: The agreeing algorithm

Definition 4.1.3 (Configuration)**.** All solutions to f_i can be represented as elements of $\mathbb{F}_2^{|\text{vars}(f_i)|}$, as only the values of variables occuring in f_i need to be stored. Such a vector is called a *configuration.*

Definition 4.1.4 (Symbol)**.** A symbol (X, L) consists of an ordered set of variables X and a list L of configurations for X.

Definition 4.1.5 (Covering)**.** Let $X_1 \subseteq X$. A configuration for X is said to *cover* a configuration for X_1 if the two are equal for all variables in X_1.

The core of the RS-Algorithm is the procedure called 'agreeing'. Consider S_i, S_j and $X_{ij} = X_i \cap X_j$.

Let L_{ij} be the set of configurations for X_{ij} that is covered by some configuration in L_j.

Let L_{ji} be the set of configurations for X_{ij} that is covered by some configuration in L_i.

Two symbols *agree* if $L_{ij} = L_{ji}$

Remark 4.1.6 (Example:)*.* Consider

$$\begin{aligned} S_1 &= (\{x_1, x_2\}, \{10, 11\}) \\ S_2 &= (\{x_1, x_3, x_4\}, \{011, 000, 101, 110\}) \end{aligned}$$

Then $X_{ij} = \{x_1\}, L_{ij} = \{1\}, L_{ji} = \{0, 1\}$. The two symbols do not agree. We delete all configurations in S_2 that do not cover any element in $L_{ij} \cap L_{ji} = \{1\}$. The result is $S'_2 = \left(\{x_1, x_3, x_4\}, \begin{array}{c} 101 \\ 110 \end{array} \right)$.

The authors interpret this algorithm as 'message passing on a graph' – the individual equations form nodes on the graph, and the agreeing algorithm passes information about the solutions on one node to it's neighboring nodes.

They note that the system of equations will often be in 'agreeing state' without showing a unique solution. They propose two ways of remedying the situation: *Splitting* and *Gluing.*

Splitting is performed as follows: If the agreeing algorithm fails to produce a solution, an arbitrary symbol S is chosen. It is split into two halves S', S'' by splitting the list of configurations into halves. The algorithm is re-run, once with S' and once with S''.

This step is the same as guessing that the solution lies in a particular half of the symbol under consideration, and running the algorithm on both possible guessed values.

Another step that can be taken to restart the agreeing step is *gluing*.

Two symbols S_i, S_j are chosen. $X_i \cup X_j =: X'$ is calculated. Then the symbols L_i, L_j are 'joined' (through adding new configurations where needed).

Example:

$$\begin{aligned} S_1 &= (\{x_1, x_2\}, \{01, 10\}) \\ S_2 &= (\{x_1, x_3, x_4\}, \{100, 010, 011\}) \end{aligned}$$

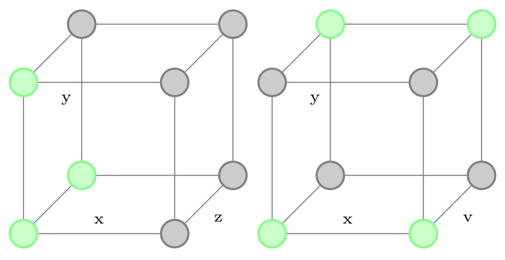
Then $X' = \{x_1, x_2, x_3, x_4\}, L' = \{0110, 0111, 1000\}$. This can have negative effects on the size of a symbol though: In the worst case, the size of the result is the product of the individual sizes:

$$|L'| = |L_1||L_2|$$

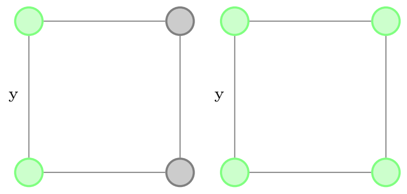
4.2 A geometric view

The RS-Algorithm can be much more easily understood as geometric operations on the varieties. This helps in understanding under which conditions the algorithm will suceed in producing a solution. It will also be the basis on which the algorithm can be reinterpreted in terms of commutative algebra in the next section.

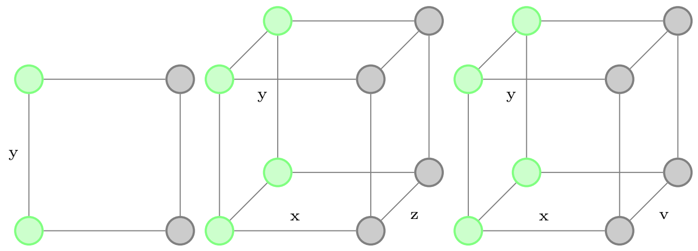
- The *splitting* step is simply a 'random cut' of the variety into two varieties of equal cardinality
- The *gluing* step is simply a 'union' of two varieties
- The *agreeing* step is a combination of 'projection', 'intersection', and 'lifting'. This is best illustrated with an example: Consider the two varieties below. The variety on the left is defined over over $\mathbb{F}_2[x, y, z]$, the one on the right is defined over $\mathbb{F}_2[x, y, v]$.



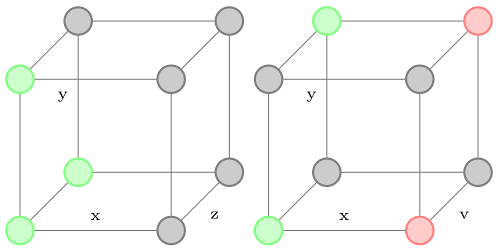
The varieties are projected to their common 'subspace', e.g. their projection into $\mathbb{F}_2[x, y]$ is calculated:



The projections are intersected. This intersection is then lifted back to the two spaces from which the original varieties were projected:



The original varieties are intersected with this lifting. This eliminates points in each varieties that cannot be present in the other variety. In our example, the first variety remains unchanged, whereas the second has the points highlighted in red removed:



4.3 An algebraic-geometric view

Through the previous geometric interpretation, the RS-algorithm can be translated into the language of commutative algebra / algebraic geometry, making the comparison to other algorithms easier and allowing the algorithm to operate on polynomials instead of solution sets.

Definition 4.3.1 (Subspace Map)**.** Let vars : $\mathcal{B}_n \to \mathfrak{P}(\{x_1, \ldots, x_n\})$ be a mapping that maps a polynomial to the set of variables that occur in this polynomial.

Let $f_{i,j}$ be the j-th monomial in f_i. Then $\mathrm{vars}(f_i) = \mathrm{vars}(\mathrm{lcm}_j(f_{i,j}))$.

Definition 4.3.2 (Projection Map)**.** For $M \subseteq \{x_1, \ldots, x_n\}$, let $\pi_M : \mathfrak{P}(\mathbb{F}_2^n) \to \mathfrak{P}(\mathbb{F}_2^{\#M})$ be the projection of the variety of a polynomial onto the smaller subspace $\mathbb{F}_2^{\#M}$ corresponding to the subring $R_M := \mathbb{F}_2[m_i]/\langle m_i^2 + m_i \rangle \subseteq \mathcal{B}_n, m_i \in M$.

Remark 4.3.3. A simple example is $\pi_{\text{vars}(f)}(V(\langle f \rangle))$ which maps the variety of f into $\mathbb{F}_2^{\#\text{vars}(f)}$ - in essence, f is treated as a polynomial in the Ring of Boolean functions that only contains the same variables as f. All in all, the projection map is just a technical construction – it doesn't carry any "deeper" significance.

4.4 Raddum-Semaev Algorithm

The RS-Assumption is now reformulated using the above constructions:

Definition 4.4.1 (RS-Assumption)**.** Assume that $\#\text{vars}(f_i) < k$ for some small k. This k needs to be small enough so that performing 2^k evaluations of f_i is feasible.

The entire algorithm can now be formulated in algebraic terms:

1. For each f_i, the set $v_i := \pi_{\text{vars}(V(\langle f_i \rangle))}$ is calulated. These are subsets of $\mathbb{F}_2^{|\text{vars}(f_i)|}$ which correspond to what is called "configurations" in the original formulation.

2. For each pair f_i, f_j, the set $S_{ij} := \text{vars}(f_i) \cap \text{vars}(f_j)$ is calculated.

3. For each pair v_i, v_j, the set $v_{ij} := \pi_{S_{ij}}(v_i) \cap \pi_{S_{ij}}(v_j)$ is calculated - exactly those points that agree on the smaller subspace $\mathbb{F}_2^{\#S_{ij}}$. These points are then lifted back and intersected with v_i, v_j:

$$v_i \leftarrow v_i \cap \pi_{S_{ij}}^{-1}(v_i j), v_j \leftarrow v_j \cap \pi_{S_{ij}}^{-1}(v_i j)$$

   This step is called the *agreeing* step in the original paper.

4. Once the *agreeing* has been run, the next step is called *splitting*. This divides an arbitrary variety v_i in two halves of equal cardinality and then re-runs the algorithm on the two systems thus created. This essentially guesses 'which half' of a variety a particular point is in.

5. New varieties are introduced in the step that the original paper called *gluing*: Given a pair v_i, v_j, $U_{ij} = \text{vars}(f_i) \cup \text{vars}(f_j)$ is calculated, and v_i and v_j are lifted to $\mathbb{F}_2^{\#U_{ij}}$. Their intersection is calculated and forms a new variety v_n. The agreeing step is run again.

4.4.1 Algebraization

Given 1.1.14 (intersecting varieties), 1.1.3 (unioning varieties), 1.1.7, the entire algorithm can be transformed to operate on polynomials in $\mathcal{B}_n$ instead of sets of points. This eliminates the need to precompute solutions for all polynomials and holding them in memory at all times (at the cost of having to compute with potentially quite large polynomials).

The only thing that we are still missing in order to perform all operations from the agreeing step directly on polynomials is the “projection to a smaller subspace”. But this can be easily achieved as follows:

Definition 4.4.2 (Projection)**.** Let $\text{vars}(f) = \{x_1, \dots x_n\}$. Then

$$\pi_{\text{vars}(f) \setminus \{x_1\}}(f) = f(0, x_2, \dots, x_n) f(1, x_2, \dots x_n)$$

We can hence recursively project larger polynomials to their restrictions on subspaces.

Gluing

The polynomial gluing step is even easier than the step performed on the list of solutions: Since gluing corresponds to unioning varieties over the larger space that is the union of the two spaces over which the varieties were defined, this step can be simply implemented by multiplying the two polynomials.

Splitting

The only step of the algorithm that cannot be trivially modeled to operate on polynomials is the *splitting* step, which randomly picks half the points on a variety and then continues to run the algorithm.

Since we cannot (without exhaustive search over one intermediate polynomial) determine the full variety of such polynomial, we have multiple options to replace the *splitting* step.

1. Pick the polynomial f with a low number of variables and determine it's variety by exhaustive search. Pick $S \subset V(\langle f \rangle)$ with cardinality roughly half that of $V(\langle f \rangle)$. Then interpolate a new polynomial from this set of points by the following formula:

$$f' := \sum_{s \in S} (\prod_{x \in \text{vars}(f)} (x + s_x + 1) + 1)$$

   This ensures that the new polynomial has exactly half the number of solutions that the previous polynomial had, but the cost of determining all points is non-negligible.

2. Assume that the points of the variety of a given f are more or less randomly distributed. Then intersecting f with another polynomial g where $\#V(\langle g \rangle) = 2^{\#\text{vars}(f)-1}$ should, on average, yield a new polynomial with half the number of solutions. This other polynomial g might be a random linear polynomial, or a random quadratic polynomial, or whichever polynomial that makes further calculations easy.

4.4.2 Summary

This chapter has put a geometric interpretation onto the RS-algorithm and provided a bridge with which this algorithm can be reformulated as operations on polynomials instead of operations on solution sets. While it is unclear whether this allows for any performance improvements, it should facilitate comparisons to other algorithms of algebraic nature.

Furthermore, the geometric interpretation of the agreeing step shows that it is particularly ill-suited for linear equations (the projection leads to an all-zero polynomial) – but fares quite well on many nonlinear ones.

Chapter 5

Published Algebraic attacks

One can argue, with some merit, that algebraic attacks have so far failed to deliver on their promise. Actual practical successes in breaking anything are few and far between, and it has been argued that for each algebraic attack on a block cipher, a more efficient statistical attack has been found.

On the other hand, a possible explanation for the disenchantment with algebraic attacks is the disappointment after having built unrealistic expectations: Extremely optimistic claims about the efficiency of algebraic attacks were publicly made in the past, and high hopes generated ("cryptanalysis using only one plain/ciphertext pair", "solving very overdefined systems is generally polynomial in nature" etc.). With such hype, it was nearly impossible for algebraic methods to live up to expectations.

In essence, successes of algebraic cryptanalysis have been limited to LFSR-based stream ciphers and asymmetric multivariate schemes (such as HFE).

In the following, a few of the published algebraic attacks will be discussed, and what the author perceives to be the "crucial" point that helped make the attack successful.

The algebraic attacks that were published so far are, overall, quite similar in nature: An equation system for the cipher in question is generated, some preprocessing is performed on the generated equations, and an attempt to solve the equation system is made. The attempt to solve the equation systems is usually done via either a *Gröbner-basis calculation* (using more efficient algorithms such as F4 or F5) or via *relinearization.*

Interestingly, it appears that "naively" generated equation systems are almost always beyond the abilities of common equation system solvers, be they Gröbner-basis calculations or relinarizations. The crucial step seems to be the "preprocessing" of equations.

5.0.3 Lowering the degree by multiplication

The first idea put forward in [CourMeier] and [Armknecht] is lowering the degree of the equations that are collected by multiplying them with well-chosen other polynomials.

In [CourMeier], several separate scenarios are discussed:

S3a There exists a function g of low degree such that $fg = h$ with $h \neq 0$ and of low degree

S3b There exists a function g of low degree with $fg = 0$

S3c There exists a function g of high degree such that $fg = h$ where $h \neq 0$ and of low degree.

The paper [MeiPaCar] goes on to show that in reality only two scenarios, namely S3a and S3b need to be considered. This thesis will give an alternate interpretation of the above scenarios in 6.4.2 .

Summary

The crucial idea here is that the degree of polynomials $f_1, \dots, f_r$ can sometimes be lowered by multiplying them with another polynomial. Multiplying our f_i with other polynomials does not risk "losing" solution points: If f_i is in the ideal generated by $\langle f_1, \dots, f_r \rangle$, then so is $f_i g$. Geometrically, we're only "adding" points.

5.0.4 Lowering the degree by finding linear combinations

Another idea was put forwards in [CourFast]: If the collection of equations and the subsequent multiplication with well-chosen polynomials still does not yield sufficiently low degree, linear combinations of these equations might be found that do.

In the context of stream ciphers, [Armknecht] describes the conditions under which such an attack works as follows:

- The equation system has been set up by use of an r-Function, e.g. is of the form:

$$\begin{aligned} F(X_1, \dots, X_r, z_1, \dots, z_r) &= 0 \\ \dots &\quad \dots \\ F(X_{n-r}, \dots, X_n, z_{n-r}, \dots, z_n) &= 0 \end{aligned}$$

- The r-Function F can be rewritten so that

$$\underbrace{F(X_1, \dots, X_r, z_1, \dots, z_r)}_{\deg(F)=d} = \underbrace{G(X_1, \dots, X_r)}_{\deg G=d} + \underbrace{H(X_1, \dots, X_r, z_1, \dots, z_r)}_{\deg(H)<d}$$

According to [Armknecht], the ciphers E_0, Toyocrypt and LILI-128 all satisfy this condition.

In such a situation, the equation system will look like

$$\begin{aligned}
G(X_1, \dots, X_r) + H(X_1, \dots, X_r, z_1, \dots, z_r) &= 0 \\
G(X_2, \dots, X_{r+1}) + H(X_2, \dots, X_{r+1}, z_2, \dots, z_{r+1}) &= 0 \\
\dots &= \dots \\
G(X_{n-r}, \dots, X_n) + H(X_{n-r}, \dots, X_n, z_{n-r}, \dots, z_n) &= 0
\end{aligned}$$

For shorter notation, rewrite $G(X_{1+i}, \dots, X_{r+i})$ as G_i.

Since G is bounded to be of degree $\leq d$, there are "only" $\mu(n,d) = \sum_{i=0}^{d} \binom{n}{d}$ different monomials that can occur. As a consequence, any sequence of G_i of length $\geq \mu(n,d)$ has to be linearly dependent, e.g. there exist coefficients $\lambda_1, \dots, \lambda_{\mu(n,d)}$ so that $\sum_{i=1}^{\mu(n,d)} \lambda_i G_i = 0$.

This alone would already yield equation systems of lower degree: The sum

$$\sum_{i=1}^{\mu(n,d)} \lambda_i F(X_{1+i}, \dots, X_{r+i})$$

would have lower degree than the original equations.

But the situation is even better: It can even be shown that G_i is a linear recurring sequence. The following is taken almost verbatim from [Armknecht]:

Lemma 5.0.3. *The sequence G_i forms a linear recurring sequence. This means that there is some integer T so that*

$$\sum_{i=0}^{T} \lambda_i G_{t+i} = 0 \ \ \forall_t \geq 0$$

The minimal polynomial of the linear recurring sequence can be calculated by a modified version of the Berlekamp-Massey algorithm.

Summary

The important idea here is that given a system $f_1, \dots, f_r$ of equations, elements can be added freely without "losing" any solution points: Arbitrary linear combinations of ideal generators stay in the same ideal. As such, any linear combination that transforms $f_1, \dots, f_r$ into something that is more nicely behaved with respect to the eventual equation system solver is quite useful.

5.0.5 Faugere-Ars attack on nonlinear filters

In [FaugArs03], the authors describe how they attacked nonlinear filter generators using Gröbner bases. No special preprocessing was mentioned – the authors simply generated equation systems from nonlinear filters and calculated Gröbner bases using F5. According to the paper, they were able to recover the full state of the underlying LFSR given L^d output bits (where L is the number of unknown bits in the LFSR and d the degree of the combining polynomial).

5.0.6 Experimental attack on Toyocrypt

An experimental attack that combines the techniques described in [CourMeier] and Gröbner basis algorithms has been carried out in [KaImWa]. The authors combined the technique of lowering the degree of the equation system with their own implementation of the F4 Gröbner base algorithm. This was done in 3 steps:

1. Equations were generated and their degree was lowered using 5.0.3
2. Using Berlekamp-Massey, relations between the equations to further lower the degree were calculated
3. The resulting equation system was fed into a parallelized version of F4

The second step took 10+ hours, but the result was an equation system which, according to the authors, could be solved in 20 seconds.

5.1 The notion of algebraic immunity

As a reaction to [CourMeier], criteria for judging the resilience of Boolean functions against algebraic attacks were proposed. The principal definition in use nowadays appears to be the following:

Definition 5.1.1 (Algebraic Immunity)**.** The algebraic immunity of $f \in \mathcal{B}_n$ is the minimal degree d of a function $g \in \mathcal{B}_n$ so that $fg = 0$ or $(f+1)g = 0$. Formally:

$$AI(f) := \min\{deg(g) | g \in \mathcal{B}_n, fg = 0 \vee (f+1)g = 0\}$$

The use of the name *algebraic immunity* is under debate though: [Armknecht] prefers to call it *annihilator immunity*, as *algebraic immunity* seems to imply resistance against a wide variety of algebraic attacks. This is not the case: The only thing that algebraic immunity implies resistance against are the degree-lowering tricks discussed in 5.0.3, and through that a certain resistance against Gröbner-basis calculations (as these usually take longer for higher-degree equation systems).

Chapter 6

The Pseudo-Euclidian Algorithm

This chapter discusses the actual 'meat' of the present thesis: A new equation system solving algorithm specialized to $\mathbb{F}_2$ and some of its properties.

The result of Lemma 1.1.14 establishes that $\mathcal{B}_n$ is a principal ideal ring and also provides a constructive method to calculate the single generator of an ideal from a set of generators in 1.1.16, mimicking the behavior of the Euclidian algorithm in Euclidian rings.

Given a set of generators $g_1, \dots, g_n$ with $V(\langle g_1, \dots, g_n \rangle) = \{y\}, y \in \mathbb{F}_2^n$, we know the exact form of the generator g through the result of 1.1.6. The restricted form of g allows us to 'read off' the single-point solution y in question.

The result is an algorithm that will, given a system of multivariate polynomial equations over $\mathbb{F}_2$ which have exactly one common solution, produce the solution to a given set of n equations in $n + 1$ additions in $\mathcal{B}_n$ and n multiplications in $\mathcal{B}_n$.

This sounds exciting, but at the same time we know that the mentioned problem is NP-hard - we therefore know that we have exponential complexity hiding somewhere in the above (it turns out it's hiding within multiplications in $\mathcal{B}_n$).

6.0.1 A naive first algorithm

Let $\mathcal{G}$ be a set of polynomials $g_1, \dots g_k \in \mathcal{B}_n$ and $\langle g_1, \dots, g_k \rangle$ the ideal generated by these polynomials in $\mathcal{B}_n$. Furthermore, assume that the variety of the ideal consists of exactly one point, e.g. $|V(\langle g_1, \dots, g_n \rangle)| = 1$.

One can make use of the results 1.1.7 and 1.1.14 in order to compute the single point on the variety.

Remark 6.0.2. One can use 1.1.16 to calculate $g \in \mathcal{B}_n$ with $G := V(\langle g \rangle) = V(\langle g_1, \dots, g_n \rangle)$.

Remark 6.0.3. The variety G consists of only one point. It is thus in the intersection of the variety $V(\langle x_1 \rangle)$ with some other set $H \subset \mathbb{F}_2^n$ *or* in the intersection of the variety $V(\langle x_1 + 1 \rangle)$ with some other set $H \subset \mathbb{F}_2^n$. G can always be 'decomposed' in one of the two following ways:

1. $\exists H \subset \mathbb{F}_2^n$ so that $G = V(\langle x_1 \rangle) \cap H$
2. $\exists H \subset \mathbb{F}_2^n$ so that $G = V(\langle x_1 + 1 \rangle) \cap H$

Lemma 6.0.4. *Let g be so that $G = V(\langle g \rangle)$. For any i exists $h \in \mathcal{B}_n$ so that*

$$g = x_i + h + hx_i \text{ or } g = x_i + 1 + hx_i$$

Proof. Let $G \subset V(\langle x_i \rangle) \Leftrightarrow G = \{y\}$ with $y_i = 0$. Then there exists a $H \subset \mathbb{F}_2^n$ so that $G = H \cap V(\langle x_i \rangle)$, and from 1.1.14 it follows that if $h \in \mathcal{B}_n$ with $V(\langle h \rangle) = H$, then

$$g = x_i + h + hx_i$$

Let $G \subset V(\langle x_i + 1 \rangle) \Leftrightarrow G = \{y\}$ with $y_i = 1$. Then there exists a $H \subset \mathbb{F}_2^n$ so that $G = H \cap V(\langle x_i + 1 \rangle)$, and from 1.1.14 it follows that if $h \in \mathcal{B}_n$ with $V(\langle h \rangle) = H$, then

$$g = (x_1 + 1) + h + h(x_1 + 1) = x_1 + 1 + h + hx_1 + h = x_1 + 1 + hx_1$$

$\square$

Lemma 6.0.5. *Given g, one can easily test for the individual components of y:*

1. *Test whether* $(x_i + 1) \mid (g + x_i) \ \Rightarrow y_i = 0$
2. *Test whether* $x_i \mid (g + 1 + x_i) \ \Rightarrow y_i = 1$

Proof. Let $g = x_i + h + hx_i \Leftrightarrow g + x_i = h(x_i + 1)$. But $g = x_i + h + hx_i$ is equivalent to $y_i = 0$.

Let $g = x_i + 1 + hx_i \Leftrightarrow g + x_i + 1 = hx_i$. But $g = x_i + 1 + hx_i$ is equivalent to $y_i = 1$.

$\square$

So in summary, one calculates a generator g then tries to decompose it in intersections of univariate linear polynomials in order to determine the values of the individual components of the solution.

We can see from the above algorithm that once we have calculated g we can pretty much determine it's (single-point) solution in linear time.

The difficulty (and computational complexity) is hidden in the calculation of g. If we follow 1.1.16, we need n polynomial multiplications and $n+1$ polynomial addition.

With each polynomial multiplication, the number of monomials can grow to the product of the numbers of monomials of the factors, e.g. if $|f|$ and $|g|$ denote the number of monomials in f resp. g, then $|fg| \leq |f||g|$ in the worst case.

6.1 Some empirical data

6.1.1 Hard instances of MQ

In order to put the algorithm to the test, a mail was sent to JC Faugere asking for a good way to generate equation systems that are difficult to solve using Gröbner base algorithms. His reply was:

> From the Groebner Basis computation point of view the most difficult case is to consider a system of random quadratic equations:
> Let $g_1(x_11, \ldots, x_n), \ldots, g_n(x_1, \ldots, x_n)$ where g_i is random and
> $$\deg(g_i) = 2$$
> Then select randomly $(a_1, \ldots, a_n) \in \mathbb{F}_2^n$. Then try to solve the algebraic system $S = \{f_1(x_1, \ldots, x_n), \ldots, f_n(x_1, \ldots, x_n)\}$ where $f_i(x_1, \ldots, x_n) = g_i(x_1, \ldots, x_n) - g_i(a_1, \ldots, a_n)$.
> S has in general one solution.

From experimentation, it is presumed that in the above the number of equations should outnumber the number of variables - at least empirically, single-solution systems were only generated about 50% of the time if n was both the number of unknowns and equations.

This is not surprising: Empirically, each random quadratic polynomial in n variables has as variety a more or less random subset of $\mathbb{F}_2^n$ with approximately 2^{n-1} points. A system of k such equations has as solution set the intersection of k such sets. This means that for each point, the odds of this point "being a solution" are roughly $\frac{1}{2^k}$. At the same time, we have 2^n points.

The experiment

The following experiment was conducted:

1. 20000 overdefined systems with 15 unknowns and 28 equations describing the system were generated. These systems were constructed so that they have randomly chosen single-point varieties. 1.1.13 tells us that the size of the final polynomial in the last step of the algorithm will be 2^l where l is the number of zero-bits in the solution.

2. The algorithm was run on each of these systems. After each step of intersection, the number of monomials in the intermediate polynomial was recorded

3. Once all equations had been processed, the result (e.g. the solution) was read off and verified to be correct

4. The process was repeated with 20000 similar equation systems that, instead of a random single-point solution, the fixed solution of $(1, \ldots, 1)$. This was done because the number of monomials in the final equation depends on the number of 0-entries in the solution vector - as such, setting all solutions to a fixed known value should eliminate fluctuations in intermediate results that are due to the nature of the final result.

The following observations were made:

1. For almost all equation systems, the maximum number of monomials in the intermediate polynomial did not exceed $2^{n-1} = 2^{14}$. In the test set of 20000 random-solution equations, 500 exceeded the 2^{14} boundary, accounting for 2.5% of the total number of systems that were generated.

2. No equation system stayed below 13273 monomials in the intermediate polynomial during all steps of the algorithm. This means that while only 2.5% exceeded the size of $2^{n-1} = 2^{14}$ monomials, *all* equation systems exceeded the size of $2^{n-2} = 2^{13}$ monomials.

3. 98.65% of all equation systems had their maximum intermediate polynomial size before the seventh intersection. 95.255% of all equation systems had their maximum intermediate polynomial size before the sixth intersection. 46.72% of the equation systems had their maximum intermediate polynomial size at exactly the fourth intersection.

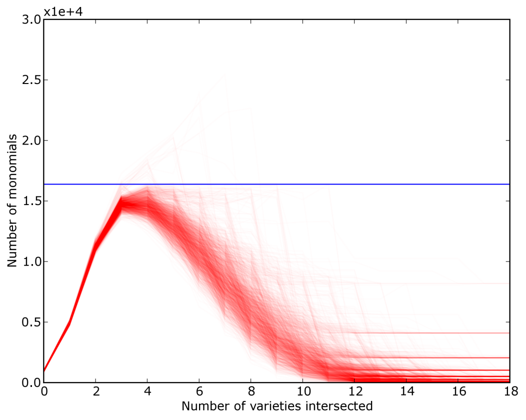


Figure 6.1: The number of monomials at each step in the algorithm for 2000 randomly generated systems. The blue line denotes the 2^{14} monomial-boundary. Each curve is drawn transparently to help identify 'density'.

Observations and Conclusion

As expected, we encounter exponential growth in running time and memory complexity. The size of the intermediate polynomials explodes - and through this, the cost of performing polynomial multiplication. It is interesting to see that the algorithm looks cheap on paper only because no proper boundaries on the sizes of the intermediate results are given – this appears to imply that in any complexity analysis for equation system solvers, the size of the intermediate results is no less important than the total number of calculations involved.

6.1.2 Equation systems generated from cryptographic problems

In previous chapters, it became evident that constructing full equation systems for PRESENT is infeasible without introducing spurious quantities of intermediate variables. As result of this, only severely weakened variants of the cipher could be investigated here. Outside of the difficulty of generating equation systems, solving was an entirely different issue - many equation systems that could be generated could not be solved without running out of memory.

Solving more than 2 rounds failed with the full cipher, and even weakening the cipher to 40 bits did not allow more than 2 rounds to be solved without exceeding 4 GB of memory consumption. Only the extreme measure of weakening the cipher to 20 respective 16 bit allowed for the algorithm to run in reasonable time.

From our experience with truly hard MQ-instances, we should expect the intermediate polynomials to grow to a size between 2^{15} and 2^{19} monomials. We can see in the following diagram that this was achieved for 20-bit PRESENT in round 5, but not for 16-bit PRESENT.

One oddity arises in the equations generated from 5 rounds of 20-bit PRESENT: The complexity peak is much earlier than for 4 rounds, and drops off quite sharply: After taking 12 equations into account, the complexity of the intermediate results is consistently lower than the intermediate results for 4 rounds. A good explanation for this is, unfortunately, not available.

6.2 Getting other information about the variety

6.2.1 Determining equality between variables

The algorithm 6.0.5 tries to determine properties of the variety of a multivariate polynomial $f \in \mathcal{B}_n$ by attempting to split the polynomial in two polynomials - one of which has an easily-understood variety. In 6.0.5 the

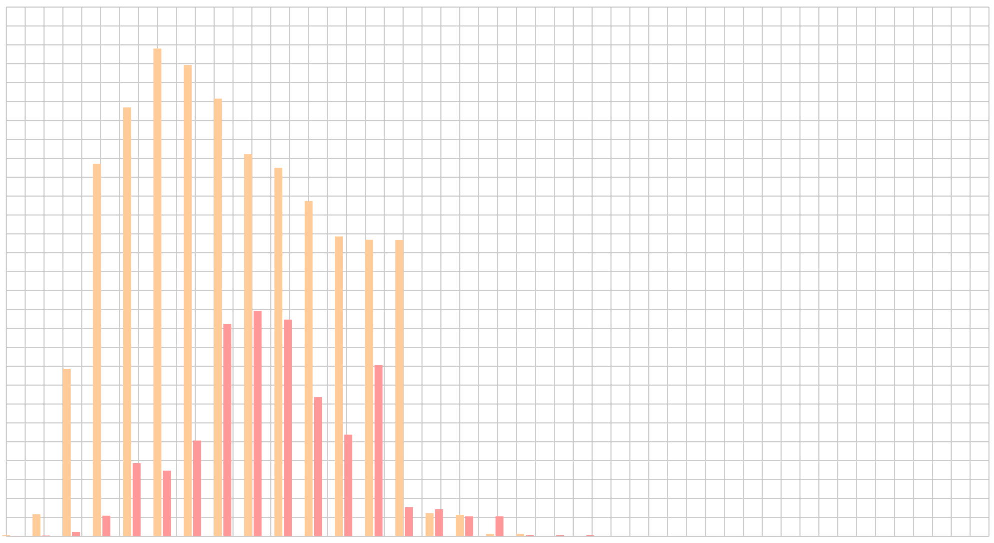


Figure 6.2: Sizes of the intermediate polynomials while solving 4 (red) respective 5 (orange) rounds of PRESENT weakened to 16 bits. The size peaks at 11925 monomials for 4 rounds and 25805 monomials for 5 rounds. A peak around 32768 would be expected for random systems. This seems to imply that after 5 rounds, the hardness of truly random systems is not achieved yet. 1 square = 1000 monomials

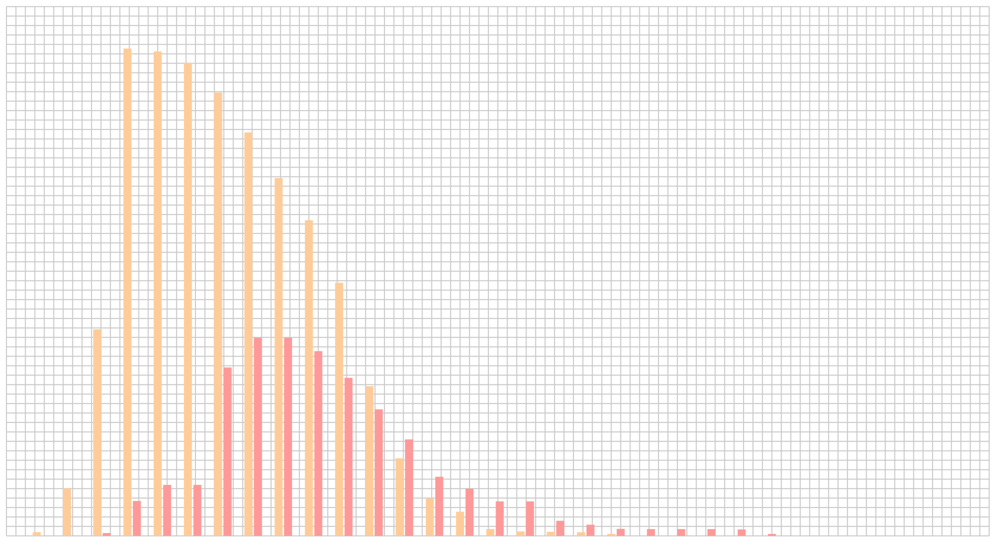


Figure 6.3: Sizes of the intermediate polynomials while solving 4 (red) respective 5 (orange) rounds of PRESENT weakened to 20 bits. The size peaks at 209735 monomials for 4 rounds and 512513 monomials for 5 rounds. For 5 rounds, this is quite close to the 524288 monomials that would be expected of a truly random system. 1 square = 10000 monomials

'simple' polynomials $(x_i + 1, x_i)$ had 'half-spaces' as varieties, but this is not a necessity. It is possible to ask comparatively complicated questions such as 'is the first variable always equal to the second variable in each point in the solution ?' by applying the same principle.

In order to determine wether a given polynomial g has above-mentioned property, we attempt to decompose g as follows:

$$g = (x_1 + x_2) + h + h(x_1 + x_2) = x_1 + x_2 + h + hx_1 + hx_2$$

$$\Leftrightarrow g + x_1 + x_2 = h(1 + x_1 + x_2)$$

This leads us to a simple test for said property that can be evaluated via a simple multivariate polynomial division.

6.2.2 Monomial count and hamming weight

A trivial corollary of 1.1.13 is that even if we end up not being capable of calculating f explicitly, other things can give us information about the solution of the equation system:

If we can calculate the number of monomials in f, or even approximate the number of monomials in f with reasonable error boundaries, we can deduce boundaries on the hamming weight of the solution vector.

6.2.3 Attempting solutions without calculating the product

An interesting factoid was pointed out by L. Gerritzen:

Lemma 6.2.1. *Let $f \in \mathcal{B}_n$ with exactly one solution $(y_1, \dots, y_n) \in \mathbb{F}_2^n$. Then $y_i = 0 \Leftrightarrow \prod_{j=0, j\neq i}^{n} x_j$ is a monomial in f.*

Proof. Follows from the proof of 1.1.13 with a small amount of effort. □

This implies that it if we were capable of testing for presence of a particular monomial in $\prod_{i=1}^{n}(f_i + 1)) + 1$, we are able to determine the hamming weight of the solution.

6.3 Viewing the algorithm as traversal of a lattice

Warning: The word "lattice" is overloaded. The meaning that is used here is the order-theoretic one !

The pseudo-Euclidian algorithm presented above can be interpreted as the traversal of a particular lattice. There are several ways to interpret the lattice.

Set-theoretic interpretation

The algorithm can be interpreted as a traversal of the lattice formed by $\mathfrak{P}(\mathbb{F}_2^n)$ with the operations $\cup, \cap$. Given the system of equations $f_1, \ldots, f_n$ one looks at $V(f_1), \ldots, V(f_n) \in \mathfrak{P}(\mathbb{F}_2^n)$ and then iteratively calculates $\bigcap_{i=1}^n V(f_i)$.

Coordinate-Ring interpretation

An alternative view of the algorithm is as the iterated construction of coordinate rings over $\mathcal{B}_n$ along with projection mappings. The algorithm can be interpreted as constructing a series $\mathcal{BF}_i = \mathcal{BF}_{i-1}/\langle f_{i-1}\rangle, \mathcal{BF}_1 := \mathcal{B}_n$ and mappings $\phi_i : \mathcal{BF}_{i-1} \to \mathcal{BF}_i$:

$$\mathcal{B}_n \underbrace{\rightarrow}_{\phi_1} \mathcal{BF}_1 \rightarrow \ldots \underbrace{\rightarrow}_{\phi_{n-1}} \mathcal{BF}_{n-1}$$

as well as $f := \phi_1 \circ \cdots \circ \phi_{n-1}(f_n)$.

By calculating $f_{ij} = (f_i + 1)(f_j + 1) + 1 = f_i + f_j + f_i f_j$, one effectively projects f_i into $\mathcal{B}_n/\langle f_j\rangle$ (or, equivalently, f_j into $\mathcal{B}_n/\langle f_i\rangle$).

Commonalities

Wether the algorithm is viewed as a traversal of the powerset-lattice of $\mathbb{F}_2^n$ or as the traversal of the lattice of coordinate rings is irrelevant since both lattices are isomorphic in this case. The important points are:

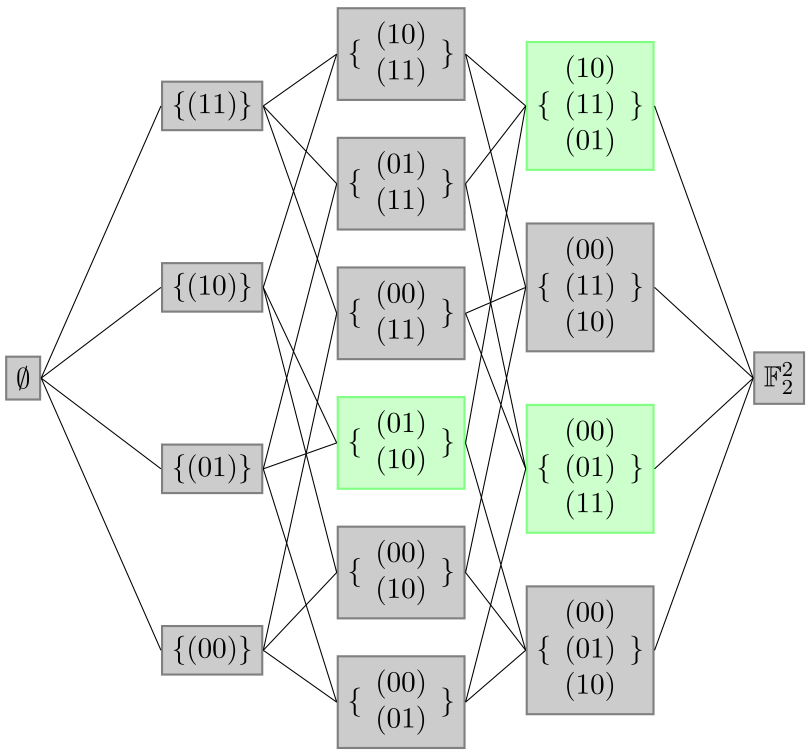
1. Initially, one starts with a set of lattice elements $f_1, \ldots, f_n$
2. One can incrementally calculate $f_i \wedge f_j$ for pairs of points
3. The goal is to calculate $\bigwedge_{i=1}^n f_i$
4. Some elements of the lattice are very expensive to represent, others are cheap to represent.
5. The order in which the calculation happens corresponds to a traversal of the lattice.

A small example

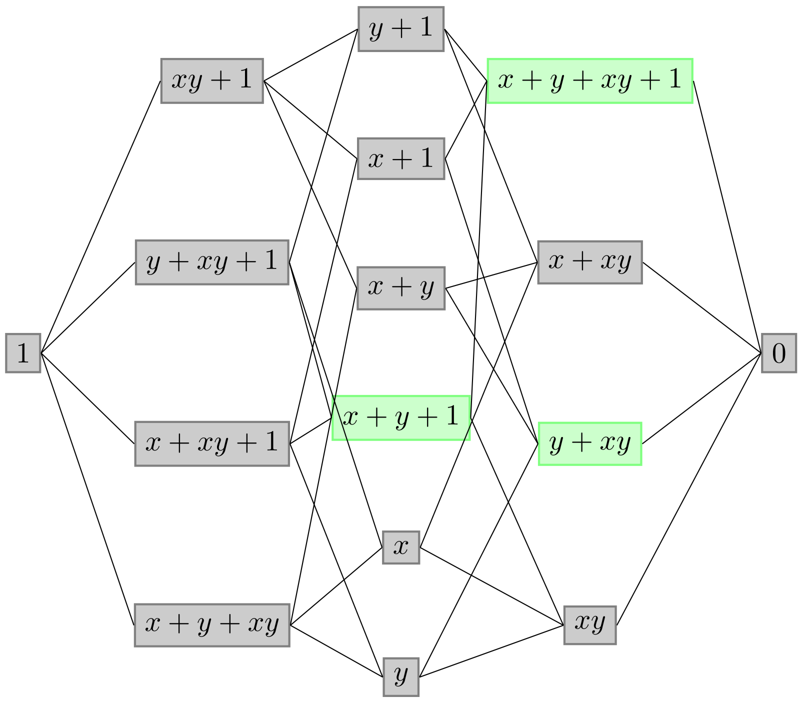
In order to clarify the above, consider a small example. Let

$$f_1 := y + xy, f_2 := x + y + xy + 1, f_3 := x + y + 1$$

We identify polynomials in $\mathcal{B}_2$ with their subsets in $\mathfrak{P}(\mathbb{F}_2^2)$ and draw the resulting lattice (the varieties of f_1, f_2, f_3 have been highlighted):

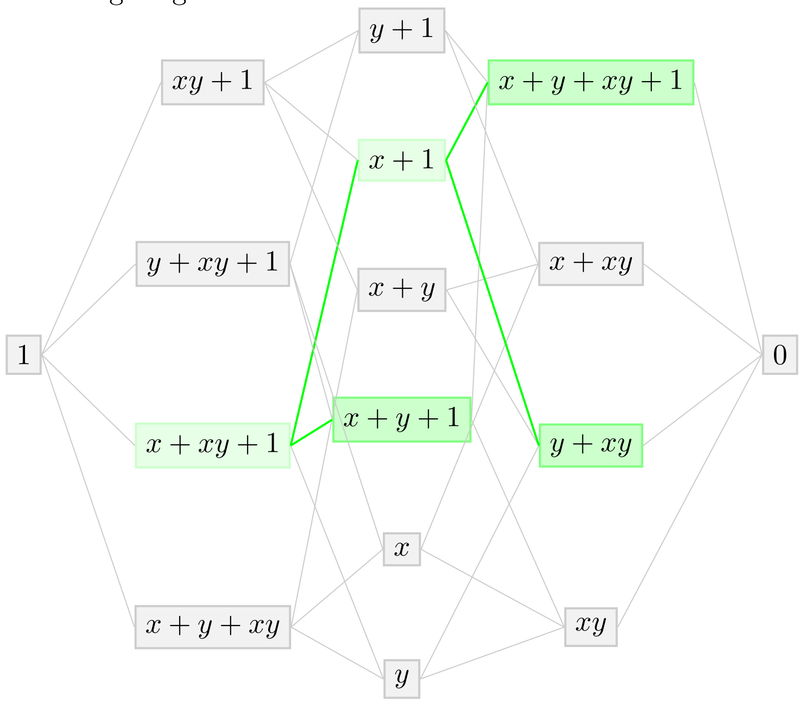


Instead of set notation, one can also label the lattice elements with their corresponding polynomials. The polynomials f_1, f_2, f_3 are highlighted:



There are 3 different ways of calculating $\bigwedge_{i=1}^{3} f_i$: $(f_1 \wedge f_2) \wedge f_3, (f_2 \wedge$

$f_3) \wedge f_1, (f_1 \wedge f_3) \wedge f_2$. Each such variant corresponds to a path through the lattice. As an example, the first variant corresponds to the path drawn in the following diagram:



From this it becomes clear that the algorithm is a traversal of this lattice, and that the "showstopper" for our attempts at solving equation systems is the fact that we hit "expensive" points in the lattice during our traversal. *Warning: The following are a set of unproven conjectures and intuitions. None of this is based on proof.* This appears to have a number of interesting implications:

- If more equations are available, more "beginning points" for the traversal exist, thus yielding more "degrees of freedom". This appears to correspond to the results in many papers about algebraic cryptanalysis that prefer overdetermined equation systems to non-overdetermined ones.

- Most equation solving algorithms, including the classical Buchberger, can be viewed as some form of mapping of lattice elements to lattice elements. As such, most equation system solvers need to confront the problem of representation sizes of intermediate results in the same way.

6.4 The notion of approximation

After one has understood the discussed algorithm as traversal of a lattice, it becomes evident that paths through the lattice that do not involve any 'expensive' points are desirable. This leads directly to the question of approximation:

Given an 'expensive' polynomial, can we approximate it through a 'cheap' polynomial ?

Before tackling this question in any depth, it is first necessary to clarify the notion of 'cheap', 'expensive', and 'approximation':

Definition 6.4.1 (Cost function)**.** A mapping $\phi : \mathcal{B}_n \to \mathbb{R}$ that, in some form, encodes the 'cost' of representing (or working with) a particular Boolean function, is called a *cost function*

Examples of cost functions:

- $\phi_V : \mathcal{B}_n \to \mathbb{N} \subset \mathbb{R}, \phi_V(f) \mapsto |V(f)|$. As one can represent a Boolean function by its variety, this function measures the cost of representing the function by explicitly storing all points of the variety.
- $\phi_M : \mathcal{B}_n \to \mathbb{N} \subset \mathbb{R}, \phi_M(f) \mapsto |\{m \text{ Monomial of } f\}|$. One can represent a Boolean function by the set of monomials. This function measures the cost of doing so by counting the number of monomials in f.
- $\phi_{VOBDD} : \mathcal{B}_n \to \mathbb{N} \subset \mathbb{R}$
  $$\phi_{VOBDD}(f) \mapsto |\{v, v \text{ Node in the OBDD-representation of } V(f)\}|$$
  The variety might be more compactly represented as OBDD, therefore it makes sense to know the cost of representing a function in this manner.
- $\phi_{OBDD} : \mathcal{B}_n \to \mathbb{N} \subset \mathbb{R}$
  $$\phi_{OBDD}(f) \mapsto |\{v, v \text{ Node in the OBDD-representation of } f\}|$$
  . Likewise, the set of monomials might be more compactly represented as OBDD, so the size of the OBDD-representation of the set of monomials is a cost function, too.

6.4.1 Approximation of polynomials

Now that cost functions are in place, a clear definition of *approximation* is needed. Intuitively, an approximation g of f is a function that differs on a small number of points from f. One can think of *overapproximations* (e.g. g vanishes on more points than f) and *underapproximations* (e.g. g vanishes on fewer points than f), which form subclasses within the set of approximations. The *quality* of an approximation is measured as the number of points on which the two functions differ, e.g. their *hamming distance.* Given f and g, we can easily calculate a polynomial that vanishes exactly on the points where f and g differ: $f + g + 1$.

Definition 6.4.2 (ϕ-Approximation of quality k)**.** Let ϕ be a cost function. A polynomial $g \in \mathcal{B}_n$ is called a ϕ-approximation of quality k if $|V(f+g+1)| = k$ and $\phi(g) < \phi(f)$. This means that g is cheaper to represent than f, but differs from f at only k points.

Definition 6.4.3 (ϕ-Overapproximation of quality k)**.** A polynomial $g \in \mathcal{B}_n$ is called a ϕ-overapproximation of $f \in \mathcal{B}_n$ with quality k if $|V(f+g+1)| = k, V(f) \subseteq V(g)$ and $\phi(g) < \phi(f)$. This means that g is cheaper to represent than f and vanishes at k points at which f doesn't vanish. This is the same as stating that g's variety is the same as f's variety with k points added.

Definition 6.4.4 (ϕ-Underapproximation of quality k)**.** A polynomial $g \in \mathcal{B}_n$ is called a ϕ-underapproximation of $f \in \mathcal{B}_n$ with quality k if $|V(f+g+1)| = k, V(g) \subseteq V(f)$ and $\phi(g) < \phi(f)$. This means that g is cheaper to represent than f and does not vanishes at k at which f vanishes. This is equivalent to stating that g's variety is f's variety with k points removed.

A number of small corollaries follow immediately:

Corollary 6.4.5. *The set of all overapproximations* $\{g_1, \ldots, g_k\} \subset \mathcal{B}_n$ *of* $f \in \mathcal{B}_n$ *is isomorphic to the ideal generated by* f.

This follows from the fact that multiplying two polynomials results in the union of their varieties, and the fact that for all g the statement $fg \in \langle f \rangle$ holds.

Corollary 6.4.6. *There is a bijection between the set of all underapproximations* $\{g_1, \ldots, g_n\} \subset \mathcal{B}_n$ *of* $f \in \mathcal{B}_n$ *and the elements of the ideal of* $f+1$.

This follows because each underapproximation g yields an overapproximation $g+1$ of $f+1$.

6.4.2 Re-visiting Courtois-Armknecht attacks on stream ciphers

One of the few successes of algebraic cryptanalysis were the Courtois-Armknecht attacks on stream ciphers. The principal trick that was responsible for lowering the degree of the equation systems far enough to make them soluble was, for given $f_1, \ldots, f_n$ finding polynomials g so that $f_1 g, \ldots, f_n g$ had much lower degree.

With the above constructions, one sees now that the "degree-lowering" trick employed in [Armknecht], [CourMeier] (and described here in 5.0.3) is nothing different than finding *overapproximations* fg of low degree. This raises an interesting question that has not been treated in the literature:

Since lowering the degree of equations $f_1, \ldots, f_n$ consists of multiplying a polynomial with each of them, one effectively adds points to each variety.

If the same polynomial is used for all equations, one even adds *the same* points to each variety.

This has implications for the solution set: The uniqueness of the solution might be lost. At the very least adding points to our solution sets might mean that *more* equations are needed before the solution becomes unique.

There is another, less clear-cut risk, too, although it seems to not have happened in practice: If one adds the same additional solutions to each equation, solving will of course become much easier - at least the points that we added to all equations will be a solution for the entire system, albeit not the one we were looking for. One should keep in mind that the so-called "pre-processing"-steps might have an influence on the solution set (e.g. enlarge the solution set).

Corollary 6.4.7. *The r-Functions described in [Armknecht] are a form of approximation:*

Recall the definition of an r-function: Given a (ι, m)-combiner and a fixed r, a function F is an r-function for the combiner if

$$F(X_1, \ldots, X_r, y_1, \ldots, y_r) = 0$$

holds whenever $X_1, \ldots, X_r$ and $y_1, \ldots, y_r$ are successive inputs respective outputs of the output function χ.

This r-function is an overapproximation for the full equation system generated by the combiner: All valid "solutions" X to the equation system

$$z_t = \chi(S_{m,t}, KL^tP)$$

with appropriate equations to model $S_{m,t}$ as result of $S_{m,t-1}$ and $KL^{t-1}P$ also satisfy $F(X) = 0$. Therefore an r-function is an overapproximation of the equation system generated by multiple clocks of the combiner.

In response to the published algebraic attacks, the notion of *algebraic immunity* was introduced (see 5.1.1.

Corollary 6.4.8. *Algebraic immunity is is a criterion about nonexistence of overapproximations of low degree. It states that no overapproximations in regard to $\phi = deg$ exist.*

Generalisation of degree-lowering techniques

After looking at 5.0.4 and 5.0.3, it is difficult not to make a generalization:

Multipliying f_i with a polynomial g is a valid operation on our equation system: Geometrically, we're just adding solutions, algebraically, we remain in the same ideal. Creating linear combinations of f_i is a valid operation on our equation system: Geometrically, $V(f_i) \cap V(f_j) \subseteq V(f_i + f_j)$, e.g. the intersection of two varieties is a subset of the variety of the sum, and algebraically the sum is part of the same ideal.

It is therefore clear that any operation on the equation system that remains in the ideal in question is legitimate. This leads to a generalized notion of degree-lowering-tricks:

Definition 6.4.9 (Degree-Lowering-Polynomial)**.** Let

$$\langle f_1, \dots, f_r \rangle \subseteq \mathbb{F}[x_1, \dots, x_n] =: R$$

be an ideal. An element $h \in R[y_1, \dots, y_r]$ is a *degree-lowering polynomial* if

$$\deg(h(f_1, \dots, f_r)) \leq \min\{\deg f_i | i = 1, \dots, r\}$$

In essence, it is a polynomial with coefficients in $\mathbb{F}[x_1, \dots, x_n]$ so that when all f_i are substituted for y_i, the resulting polynomial has lower degree than any f_i.

One sees easily that 5.0.3 means that the degree-lowering polynomial has only one monomial, e.g. is of the form $h := gy_1$ (where g is the polynomial that our equation is multiplied with). One also sees easily that 5.0.4 uses degree-lowering polynomials that are linear, e.g. of the form $h := \sum_{i \in I} y_i$.

A first idea to calculate such h would be to use Gröbner bases: If one proceeds similarly to 5.0.4 and splits the f_i into high-degree components G_i and low-degree components H_i, one could attempt to calculate the syzygy module of the G_i. This would yield polynomials in R that evaluate to zero if the G_i are inserted. Unfortunately, things are not quite so easy: In the case of a linear h, calculating $h(G_1 + H_1, \dots, G_r + H_r)$ does not yield any interaction between the high-degree and low-degree components: If the high-degree parts cancel, only low-degree parts are left. In case of general polynomials for h, this is no longer the case: Several low-degree components might be combined to yield high-degree results, and complicated interactions occur. It is, as of now, wholly unclear how a good h might be constructed.

While not being very constructive, this definition allows us to define a *generalized algebraic resistance (with respect to* deg*)*:

Definition 6.4.10 (Generalized algebraic resistance (with $\phi = \deg$))**.** A system of equations

$$f_1, \dots, f_r$$

is has generalized algebraic resistance (with $\phi = \deg$) of d where

$$d = \min\{\deg(h(f_1, \dots, f_r)), h \in R[y_1, \dots, y_r]\}$$

From an algebraic perspective, this definition is equivalent to

$$d = \min\{\deg(f), f \in \langle f_1, \dots f_r \rangle\}$$

Looking at the situation from this angle, we see an amusing corolary:

Corollary 6.4.11. *Any system $F := \langle f_1, \dots, f_r \rangle \subseteq \mathcal{B}_n$ with unique solution in $\mathcal{B}_n$ satisfies $d = 1$.*

Proof. We know that any polynomial that has a superset of $V(F)$ as variety is an element of F. Therefore, either the polynomial x_1 or the polynomial $x_1 + 1$ is an element of F, and $d = 1$. □

This means that as long the system of equations has unique solution, there will be an h so that the degree of the system can be lowered, even to 1. In order to have a more meaningful measure of resistance, one would need to take more things into account, at least the following:

1. The cost of calculating $h(F_1, \dots, F_r)$
2. The "loss of accuracy" when calculating h

6.4.3 Constructing approximations

Since the problems we ran into during the solving of our equation systems using the pseudo-Euclidian algorithm had little to do with degree, and much more to do with exponential growth in the number of monomials in our intermediate results, a natural question arises: Can we construct an over-approximation g for a given large polynomial f in a systematic fashion ?

A first stab

Without loss of generality, let f contain the monomial '1'. Let l be the number of monomials in f. The goal is now to construct $g \in \mathcal{B}_n$ with $gf = g \Leftrightarrow f(g+1) = 0$ so that the number of monomials m of g are less than or equal l.

We fix $m < l$. Let f_i denote the i-th monomial of f with some ordering, and let g_i denote the i-th monomial of g with some ordering.

The standard way of multiplying f with g works by filling in the elements of the following matrix and removing all matrix elements that occur an even number of times:

$$\begin{array}{c|cccc} & f_1 & \dots & f_{l-1} & 1 \\ \hline g_1 & f_1 g_1 & & f_{l-1} g_1 & g_1 \\ \dots & & & & \\ g_m & f_1 g_m & & f_{l-1} g_m & g_m \end{array}$$

In order to satisfy the condition that $fg = g$, it has to be assured that all matrix elements except the right-most column occur an even number of times.

The algorithm now works as follows: The above matrix is considered without the rightmost column. It's elements are called $a_1, \dots, a_{l-1,m}$. An arbitrary decomposition into disjoint subsets of cardinality 2 is chosen (there are $\frac{k!}{\frac{k}{2}!2^{\frac{k}{2}}}$ possible such decompositions for sets of cardinality k when 2 divides k). Each such "pair" needs to "cancel", e.g. both elements of the pair have to be equal. Thus each subset of cardinality 2 of matrix elements of the form $\{a_i, a_j\}$ gives rise to an equation of the form $a_i = a_j$. This is the same as

$$f_\alpha g_\beta = f_\gamma g_\delta$$

for some $\alpha, \beta, \gamma, \delta$. This implies $\log(f_\alpha g_\beta) = \log(f_\gamma g_\delta)$. Since multiplication of monomials is the same as bit-wise "or" of the exponents, and since $xORy = x + y + xy$, the following system of equations arises:

$$\begin{aligned}
\log(f_\alpha)_1 + \log(g_\beta)_1 + \log(f_\alpha)_1 \log(g_\beta)_1 &= \log(f_\gamma)_1 + \log(g_\delta)_1 + \log(f_\gamma)_1 \log(g_\delta)_1 \\
\ldots &= \ldots \\
\log(f_\alpha)_n + \log(g_\beta)_n + \log(f_\alpha)_n \log(g_\beta)_n &= \log(f_\gamma)_n + \log(g_\delta)_n + \log(f_\gamma)_n \log(g_\delta)_n
\end{aligned}$$

This would normally look like a rather large linear equation system (with nm unknowns), but because the $\log(f_i)$ are all known, only the following four types of equations can arise:

$$\log(f_\alpha)_i = 0 = \log(f_\gamma)_i \quad \Rightarrow \quad \log(g_\beta)_i = \log(g_\delta)_i \tag{6.1}$$

$$\log(f_\alpha)_i = 0, \log(f_\gamma)_i = 1, \quad \Rightarrow \quad \log(g_\beta)_i = 1 \tag{6.2}$$

$$\log(f_\alpha)_i = 1, \log(f_\gamma)_i = 0, \quad \Rightarrow \quad \log(g_\delta)_i = 1 \tag{6.3}$$

$$\log(f_\alpha)_i = 1 = \log(f_\gamma)_i \quad \Rightarrow \quad 0 = 0 \tag{6.4}$$

This means that each equation can either imply equality between two particular exponents in the monomials g_β and g_δ, or that one of the two monomials needs to have a particular exponent set to 1. Interestingly, this also implies that the equation system will *always have a solution*: Since there is no way that a value of 0 can be implied, there is no way a contradiction could arise. This is not surprising: the polynomial $g = 0$ will always be an overapproximation to f (but of little value).

The above thoughts lead to the following algorithm that constructs a "random" overapproximation of f:

```
Data: $f \in \mathcal{B}_n, m \in \mathbb{N}$
Result: $g \in \mathcal{B}_n$ with $fg = g$ and $\phi_M(g) = m$
init;
$g \leftarrow 0$;
while $g == 0$ do
    choose random partition;
    generate system of equations;
    solve for $g$;
end
Return $g$;
```

Algorithm 4: Approximation of polynomials

It is important to keep in mind that no guarantuees regarding the *quality* of the approximation are made: This algorithm constructs *some* approximation, but quite possibly a very bad one.

Now consider the case that f does not contain the monomial '1'. We can use a variant of the above algorithm: Consider the notion that we wish to construct a polynomial $g+1$ that is an approximation of f. This leads to $f(g+1) = g+1 \Rightarrow fg+f = g+1 \Rightarrow fg+g = f+1 \Rightarrow g(f+1) = (f+1)$. We therefore wish to construct g so that $g(f+1) = (f+1)$. We know that f does not contain the monomial '1', hence we know that $f+1$ *does*. We *decide* that g should contain the monomial '1'.

$$\begin{array}{c|cccc}
 & f_1 & \dots & f_{l-1} & 1 \\
\hline
g_1 & f_1g_1 & & f_{l-1}g_1 & g_1 \\
\dots & & & & \\
g_m & f_1g_m & & f_{l-1}g_m & g_m \\
1 & f_1 & \dots & f_{l-1} & 1
\end{array}$$

This time, we choose a decomposition into subsets of cardinality 2 ignoring the last *row* instead of the last *column*. The algorithm proceeds analogously from here.

6.4.4 Constructing approximations: Complete failure

The described approximation algorithm was implemented in C++. A small test polynomial f consisting of 121 monomials in 8 variables was given to it, and the task of approximating this f with a polynomial of no more than 50 monomials. Unfortunately, the algorithm consistently produces $g = 0$, which, while being a valid overapproximation of f, is also completely useless.

On the reasons for failure

The algorithm picks a random partition of the matrix in pairs. While each partition of the matrix corresponds to one overapproximation, there are vastly more partitions than valid overapproximations. Most of these partitions will correspond to the trivial overapproximation, 0. Since we are blindly choosing amongst the possible partitions, the probability of us "accidentally" hitting a good overapproximation is vanishingly low.

6.5 Conclusion (for now)

This chapter ends in a bit of a disappointment: While a new, interesting, and surprisingly simple algorithm for finding the solution to an equation system consisting of elements of $\mathcal{B}_n$ has been found, it's performance on actual hard problems does not convince. There are a number of directions in which further research seems promising:

1. Understanding the “expensive” and “cheap” points on our lattice better:
   (a) How many “cheap” points are there ?
   (b) How many randomly chosen elements does a set of lattice elements need to have to have significant probability that a “cheap” path through the lattice exists ?
   (c) How many more “cheap” points do we get if we take OBDD-representations of the lattice elements (which, for many elements, are much cheaper) ?
2. Tackling “cheap” approximations of “expensive” functions:
   (a) Can we approximate some lattice elements with reasonable error boundaries ?
   (b) How dense are these lattice elements in the overall lattice ?
3. Is there some way of calculating $\prod f_i$ without calculating the products of smaller subsets ?
4. Is there any way of calculating *any* monomial except $\prod_j x_j$ in $\prod f_i$ without calculating the full product ?
5. Can we estimate (with reasonable error boundaries) the number of monomials in $\prod f_i$?

In summary: We do not understand the properties of this algorithm very well. We know what it does geometrically, but it is unclear which equation systems will be easy to solve, which equation systems will be hard to solve, and whether many cases exists where this algorithm is superior to already-known algorithms.

Chapter 7

Pseudo-Euclid over finite fields

7.1 Generalization to arbitrary finite fields

If one thinks geometrically about the algorithm (and the specific ring $\mathcal{B}_n$), one can conclude the following points:

- If one isn't interested in solutions in the algebraic closure and hence considers $\mathcal{B}_n$ instead of $\mathbb{F}_2[x_1, \ldots, x_n]$, one obtains an object that is isomorphic to the *ring of sets* of $\mathbb{F}_2^n$
- This object is a principal ideal domain, and the individual points correspond to the 'prime' elements in this domain – each object is composed as finite product of polynomials that have exactly one point as their solution.

There is little in these observations that would suggest difficulties over other finite fields than $\mathbb{F}_2$. Clearly, the *ring of sets* over $\mathbb{F}_q^n$ will always have the property that each element is composed of the finite union of individual points - so most parts of our construction should work regardless.

What do we need in order to 'port' the results from $\mathbb{F}_2$ to $\mathbb{F}_q$? In essence, something that allows us to *intersect varieties.* First, we define the analogous version of $\mathcal{B}_n$ for arbitrary base fields:

Definition 7.1.1 (Ring of $q$oolean functions)**.** Given a finite field $\mathbb{F}_q, q = p^n, p$ prime, the ring of $q$oolean functions in n variables is defined as

$$\mathbb{F}_q[x_1, \ldots, x_n]/\langle x_1^q - x, \ldots, x_n^q - x\rangle$$

and written as $\mathcal{F}_n$.

Varieties etc. are defined analogously to the definitions made in the beginning of this thesis. The only thing that is really required for the pseudo-Euclidian algorithm to work is a device for intersecting varieties. The following formalizes this:

Definition 7.1.2 (Intersection Polynomial)**.** Let $\mathbb{F}_q, q = p^n, p$ prime, be an arbitrary finite field. A polynomial $\theta \in \mathbb{F}_q[x,y]$ is called *intersection polynomial* if the following holds:

$$\theta(x,y) = 0 \Leftrightarrow x = 0 \wedge y = 0$$

An intersection polynomial is a device that is used for intersecting varieties: Let $f, g \in \mathcal{F}_n$ and $\theta \in \mathbb{F}_q[x,y]$ is an intersection polynomial. Since $\mathbb{F}_q \subset \mathcal{F}_n$, one can consider θ as an element of $\mathcal{F}_n[x,y]$, and evaluate $h = \theta(f,g) \in \mathcal{F}_n$.

The property that θ vanishes if and only if both parameters vanish leads to

$$V(\theta(f,g)) = V(f) \cap V(g)$$

and we have our 'intersection'. If one has such a polynomial f as well as two elements from $\mathcal{F}_n$,

Lemma 7.1.3. *The polynomial $\theta \in \mathbb{F}_q[x,y]$,*

$$\theta = x^{q-1} + y^{q-1} - x^{q-1}y^{q-1}$$

is an intersection polynomial for arbitrary finite fields.[1]

Proof. The proof simply applies Fermat's little theorem: $\forall a \in \mathbb{F}_q^* \; a^{q-1} = 1$. Clearly, θ vanishes if $x = y = 0$. If $x \neq y = 0$, the polynomial evaluates to 1, and thus does not vanish. If $y \neq x = 0$, likewise. If $x \neq 0, y \neq 0$ the polynomial evaluates to $1 + 1 - 1$, which does not vanish. $\square$

[1]I have to thank Prof. H. Flenner for his insight that allowed me to generalize this to arbitrary base fields by inverting the sign on the last term

Summary and Outlook

This thesis has, unfortunately, openend many more questions than it has answered. In 6.5, a number of interesting questions was listed in relation to the pseudo-Euclidian algorithm. Further questions show up in relation to the generalized pseudo-Euclidian algorithm over other finite fields: Experiments that see how it performs on various equation systems would be interesting indeed.

The section 6.4.2 shows that the general question on how to systematically construct (over)approximations of Boolean functions with desired properties would be of use not only for the pseudo-Euclidian algorithm, but also as preprocessing step to other equation solvers. It is hence a bit disappointing that the approximation algorithm proposed in 6.4.3 fails so thoroughly. A thorough and practical theory for the approximation of (sets of) Boolean functions would be fantastic to have.

On the defensive side, much better criteria than algebraic immunity are needed. Unfortunately, they require better understanding of the solution algorithms and their expected running times – which will, very likely, depend on the exact shape of the equations in question, and on any preprocessing that might be done prior to this.

It really appears that algebraic methods are not fully understood in their implications for cryptography, and in their efficiency. The only conclusive thing we can say is that "naive" attempts at algebraic cryptanalysis (e.g. "write equations, solve") fail, as even writing the equations easily outstrips the computation needed for a brute-force search.

In comparison to the statistical attacks such as differential or linear cryptanalysis, we understand much less about the conditions under which algebraic attacks work. The results on some stream-ciphers are encouraging, but it appears that much more work is required before these methods can be made fruitful on any realistic block cipher, and it is doubtful wether algebraic methods will ever outperform statistical/combinatorial methods on that front.

This does not imply that algebraic methods are not going to be useful in cryptanalysis – but the "simple" hope that we can just write & solve equation systems will, in all likelyhood, not come true.

Bibliography

[Carle07] Claude Carlet et. al *Boolean Methods and Models.* Cambridge University Press, to appear.

[BiSha90] Eli Biham, Adi Shamir: *Differential Cryptanalysis of DES-like cryptosystems.* Journal of Cryptology, 4(l):3-72, 1991

[Matsui93] Mitsuru Matsui: *Linear Cryptanalysis for DES cipher*, EUROCRYPT '93, pp 386-397, Springer

[Shan45] Claude Shannon: *Communication Theory of Secrecy Systems*, Bell System Technical Journal, vol.28(4), page 656715, 1949

[FauJou03] JC Faugere, Antoine Joux: *Algebraic Cryptanalysis of Hidden Field Equation Cryptosystem Using Gröbner Bases* Rapport de recherche de l'INRIA, RR-4738, February 2003

[FauF499] JC Faugere: *A new efficient algorithm for computing Gröbner bases (F4)*, Journal of Pure and Applied Algebra 139, 1-3 (June 1999), 61-88

[Faugere02] JC Faugere: *A new efficient algorithm for computing Gröbner bases without reduction to zero (F5)*, ISSAC '02: Proceedings of the 2002 international symposium on Symbolic and algebraic computation, pages 75 - 83

[FaugArs03] JC Faugere, Gwenole Ars: *An algebraic cryptanalysis of nonlinear filter generators using Gröbner bases* Rapport de recherche de l'INRIA, RR-4739, February 2003

[Patarin95] Patarin, J.: *Cryptanalysis of the Matsumoto and Imai Public Key Scheme of Eurocrypt '88* Des. Codes Cryptography, pages 175-209, Kluwer Academic Publishers, 2000

[MatImai88] T. Matsumoto, H. Imai *Public quadratic polynomial-tuples for efficient signature-verification and message-encryption*, Lecture Notes in Computer Science on Advances in Cryptology-EUROCRYPT'88, Springer

[RadSem06] H. Raddum, I. Semaev *New technique for solving sparse equation systems* , eprint.iacr.org preprint archive

[RadSem07] H. Raddum, I. Semaev *Solving MRHS linear equations* , eprint.iacr.org preprint archive

[BrickDrey07] M. Brickenstein, A.Dreyer *POLYBORI: A Gröbnerbasis framework for Boolean polynomials*

[Brick06] M. Brickenstein *SlimGB: Gröbner Bases with Slim Polynomials*

[BaCouJef07] G.V. Bard, N.T. Courtois, C. Jefferson *Efficient Methods for Conversion and Solution of Sparse Systems of Low-Degree Multivariate Polynomials over GF(2) via SAT solvers*

[McDChaPie07] C. McDonald, C. Charnes, J. Pieprzyk *Attacking Bivium with MiniSat* , eprint.iacr.org preprint archive

[CouBar07] G.V. Bard, N.T. Courtois *Algebraic and Slide Attacks on KeeLoq* , eprint.iacr.org preprint archive

[Bard07] G.V. Bard *Algorithms for solving linear and polynomial systems of equations over finite fields with applications to cryptanalysis*, PhD Thesis, University of Maryland, College Park, 2007

[Daum01] M. Daum *Das Kryptosystem HFE und quadratische Gleichungssysteme "uber endlichen K"orpern* Diplomarbeit, Bochum 2001

[Armknecht] Frederik Armknecht, *Algebraic Attacks on certain stream ciphers*, PhD thesis, Mannheim, 2006

[AdamsLou] William W. Adams, Philippe Loustaunau *An Introduction to Gr"obner Bases*

[KreuRobb] Martin Kreuzer, Lorenzo Robbiano *Computational Commutative Algebra 1*

[CourMeier] Nicolas T. Courtois, Willi Meier *Algebraic Attacks on Stream Ciphers with Linear Feedback*, eprint.iacr.org preprint archive

[MeiPaCar] Willi Meier, Enes Pasalic, Claude Carlet *Algebraic Attacks and Decomposition of Boolean Functions*, eprint.iacr.org preprint archive

[Present] A. Bogdanov, L.R. Knudsen, G. Leander, C. Paar, A. Poschmann, M.J.B. Robshaw, Y. Seurin, C. Vikkelsoe *PRESENT: An Ultra-Lightweight Block Cipher* Proceedings of CHES 2007

[KaImWa] Mitsuru Kawazoe, Hideki Imai, Hideyuki Watanabe, Shigeo Mitsunari Implementation of F4, experimental cryptanalysis of Toyocrypt and 58-round SHA1 using Gröbner bases

[CourFast] Nicolas Courtois Fast algebraic attacks on stream ciphers with linear feedback. CRYPTO 2003, LNCS 2729, pp: 177-194, Springer

[Lesezirkel] Ralf-Philipp Weinmüller, Martin Albrecht, Thomas Dullien Meetings of the "Lesezirkel Computeralgebra" in Bochum und Frankfurt

[Courtois02] Nicolas Courtois, J. Pieprzyk Cryptanalysis of Block ciphers with overdefined systems of equations pp 267-287, AsiaCrypt 2002, Editor: Y. Cheng, LNCS Volume 2501, Springer

Danksagung

Ich möchte an dieser Stelle Ralf-Philipp Weinmann für seine Geduld meinen Monologen gegenüber danken, und für viele gute und kritische Anregungen. Martin Albrecht gilt ebenfalls dank für die vielen konstruktiven Gedanken bei unseren Treffen. Desweiteren danke ich Dr. Marian Margraf und Prof. Roberto Avanzi für lange und oft interessante Diskussionen. Evelyn Ebert danke ich für ihre nahezu endlose Geduld.

Erklärung

Hiermit versichere ich, diese Diplomarbeit selbstständig und nur unter Verwendung der im Literaturverzeichnis angegebenen Hilfsmittel angefertigt zu haben.

Ort, den Datum

Thomas Dullien