國立臺灣大學電機資訊學院電機工程學系

碩士論文

Department of Electrical Engineering
College of Electrical Engineering and Computer Science
National Taiwan University

Master Thesis

解佈於二元體之多項式方程組之快速窮舉法

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

周彤

Tung Chou

指導教授：鄭振牟 博士

Advisor: Chen-Mou Cheng, Ph.D.

中華民國 99 年 6 月

June, 2010

誌謝

首先，我想感謝在這個 project 初期給予我很多幫助的 Ruben Niederhagen 和陳學中。Ruben 所建立的程式架構，以及使用 Perl 作爲 code generation 的方式，無疑都爲後來的程式發展奠定了穩固的基礎。陳學中提供了許多 CUDA programming 上的協助，包括在二次系統上第一個成功的實作。沒有他們的協助，也許今天就沒有這篇論文的產生。

接著我想感謝幾位沒有直接參與這個 project 的同學：蕭俊宏和我討論了許多複雜度上的議題以及數學推導，另外他也提供了許多有關 SSE intrinsics 的相關知識；陳明興學長則是在紀錄實驗數據時，給了我相當及時的協助。

在這兩年的碩士生涯當中，我的指導教授鄭振牟老師提供了一個很自由的實驗室環境，使我可以充分地發揮自己的研究能力。除此之外，鄭老師總是會確保實驗室擁有充分的研究資源，當我有疑問時，老師也總是會不厭其煩地提供解答和建議。而另一位指導教授，中研院的楊柏因老師，除了在他專精的數學和多變量密碼學領域授予我大量的知識以外，也經常提供各種實用的討論及見解。總之，兩位老師都給了我豐富的建言和指導，使我獲益良多，眞的很感謝他們。

最後我想感謝我的父母長期以來給我的關懷和支持，希望這樣一份小小的成就，可以讓他們引以爲傲。

摘要

解多變量系統的問題在許多領域，包含代數攻擊和多變量密碼學，都具有重要的地位。然而，現存的演算法如 $\mathbf{F_4}/\mathbf{F_5}$和 XL 雖然對於具有某些特性的系統效果特別顯著，但在面對一般的系統時，通常都無法有效率地解決問題，甚至會帶來嚴重的記憶體匱乏的問題。

基於上述的理由，我們便開始尋求另一個適合一般系統的解決方案，也就是窮舉搜尋 (exhaustive search) 的方式。在這篇論文中，我們提出並深入研究了一個窮舉搜尋的演算法 GGCE (generalized Gray code enumeration)，用以解佈於二元體的多項式系統。這個演算法不僅相當易於平行化，對於解低次數的系統也有非常好的表現。對於這個演算法理論上的效能和記憶體需求等重要議題，在之後也會有深入的分析。

以實際面而言，我們將 GGCE 實作於顯示卡及 CPU 上。經過適當的最佳化之後，我們發現 GGCE 不僅在理論上相當有效率，實際的表現甚至遠勝過現有的演算法。換句話說，在面對一般系統時，窮舉搜尋可能是最好的解法。

簡而言之，我們提出了一個不同於以往的策略來解多變量系統。GGCE 證明了窮舉搜尋本身的強大能力，對於需要解多變量系統的眾多領域，勢必也會產生相當的影響。

關鍵字: 顯示卡、代數攻擊、多變量密碼學、窮舉搜尋、平行化。

Abstract

Solving multivariate polynomial systems over finite fields is a problem of fundamental importance in algebraic cryptanalysis and multivariate cryptography. Existing Gröbner-basis solvers such as $\mathbf{F_4}/\mathbf{F_5}$ and XL (eXtended Linearization) have been well studied and proved to be powerful against modern cryptosystems such as HFE.

However, these solvers are useful only for systems with algebraic defects or excessively overdetermined systems. For generic systems, their run time can be quite substantial, not to mention the immense memory pressure. Moreover, study of exhaustive search algorithms seems to be a missing link in this field. These have aroused our curiosity and resulted in this work.

In this thesis, we propose and investigate an exhaustive-search algorithm and several variants of it intended to solve generic polynomial systems over $\mathbb{F}_2$, the field consisting of two elements. The algorithm is easy to parallelize and works especially well for low-degree systems, the reasons for which shall be clear after we analyze the complexity-theoretical performance, memory consumption, and impact of several adjustable parameters later in this thesis.

On the practical side, we have implemented our algorithm, along with several efficiency-enhancing optimizations, for quadratic, cubic, and quartic systems on prevailing GPUs and CPUs using the CUDA framework and SSE2 intrinsics, respectively. Even though the implementation may leave certain room for improvement (for they are not implemented in assembly code), they have outperformed all existing implementations of Gröbner-basis solvers to which we have access, a clear demonstration of the practicability of our algorithm.

Today, we can solve 48 quadratic equations in 48 binary variables with just an NVIDIA GeForce GTX 295 graphics card in 21 minutes. It would be 36 minutes for cubic equations and 126 minutes for quartics. In contrast, the implementation of F_4 in `MAGMA-2.15-5`, often cited as the best Gröbner-basis solver available today, would

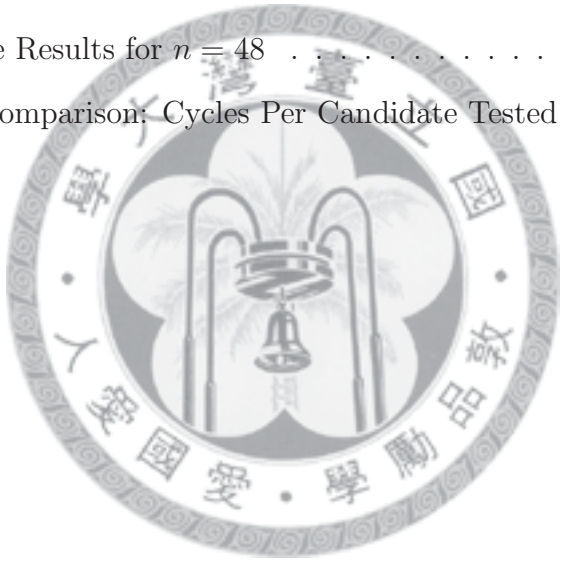
run out of memory on a system with 25 $\mathbb{F}_2$-variables in as many cubic equations. While it succeeds in solving 20 cubic equations in 20 $\mathbb{F}_2$-variables, it takes about 2.5 hours to finish. Either system can be solved by the proposed enumerative solver in less than a second.

Keywords: *algebraic cryptanalysis, multivariate cryptography, multivariate polynomials, solving systems of equations, exhaustive search, parallelization, CUDA, Graphic Processing Units (GPUs).*

Contents

List of Figures

List of Tables



Chapter 1

Introduction

1.1 Motivation

Solving a system of m nonlinear equations in n variables over $\mathbb{F}_q$ is an important problem in cryptography and many other fields [2]. Because the problem is NP-complete, it has been used to design asymmetric cryptographic primitives that collectively are known as multivariate cryptography. During the past few decades, this relatively young branch of cryptography has flourished, resulting in public-key cryptosystems such as HFE, SFLASH, and QUARTZ [15, 23, 24], as well as stream ciphers such as QUAD [7].

Another important direct application of system solving is algebraic cryptanalysis, a class of attacks against ciphers by converting the problem of breaking a cipher into an equivalent problem of solving a polynomial system. The importance of algebraic cryptanalysis resides in it generality: it has been used to attack a reduced-round DES [3] and block ciphers such as Keeloq [14]. It has also lead to a faster collision attack on 58 rounds of SHA-1 [27].

The problem, not surprisingly, has been long studied, and the most renowned solvers might be $\mathbf{F_4}/\mathbf{F_5}$ [18, 19]. These solvers, being the most advanced among Gröbner-basis solvers, broke the first HFE challenge [20]. Another notable solver is the XL algorithm [16] and its variants, which are simpler than $\mathbf{F_4}/\mathbf{F_5}$ but are expected to work well asymptotically due to the manipulation of sparse matrices

instead [1].

For "generic" quadratic systems, experts believe that Gröbner-basis methods will go up to degree D_0 and then require the solution of a system of linear equations with $T \gtrsim \binom{n}{D_0-1}$ variables, which will take at least $\text{poly}(n) \cdot T^2$ bit-operations [5, 28]. For example, if we assume we can operate a Wiedemann solver on a $T \times T$ submatrix of an extended Macaulay matrix of the original system, then the polynomial is $3n(n-1)/2$. When $m = n = 200$, D_0 is 25, making the value of T exceeds 2^{102}, while a basic version of our algorithm takes only $m2^{n+1}$ bit operations. Even taking into consideration guessing before solving [9, 29], we can still easily conclude that it would be impossible for Gröbner-basis methods to outperform exhaustive search in the practically interesting range of $m = n \leq 200$.

Knowing that existing solvers, though well studied, are not suitable for generic systems, naturally we would like to know the capabilities of exhaustive search solvers, a desolate area in the field. Is it possible to find more efficient exhaustive search algorithms? What is the best time complexity, though expected to be at least exponential, they can achieve? Can an exhaustive search solver be comparative to, or even outperform the best among the existing solvers? All these are questions to which we eager to know the answers.

As exhaustive search algorithms are usually highly parallelizable, the trend in parallel computing also motivates our work. Considering that CPU's clock rate is limited by critical issues like heat dissipation, the latest pursuits after faster computers have been taking a different route in semiconductor industry. In recent years, a series of multi-core processors like dual-cores and quad-cores have been released and soon become the mainstream. Moreover, thanks to the well-known series of Streaming SIMD Extensions (SSE), CPUs are capable of processing even more data simultaneously.

In addition to CPUs, GPUs are an extreme of hardware dedicated for parallel computing. With several hundreds of "cores," GPUs can achieve extremely high thread-level parallelism (TLP), resulting in potent computational power that often

outperforms that of CPUs by a factor of ten or more. Emergence of programming frameworks, such as NVIDIA's CUDA, that allow people to harness such computation power has gained GPUs more and more popularity as an implementation platform in past few years.

In this thesis, we will show how we build a fast exhaustive search algorithm intended for solving generic nonlinear multivariate systems from a theoretical perspective. We will also show how we design and implement a parallel program capable of exploiting the computation power of modern GPUs and CPUs from an implementation perspective.

1.2 Problem Statement

The problem we deal with can be formally defined as follows.

Problem Solve $f^{(0)}(\mathbf{x}) = f^{(1)}(\mathbf{x}) = \cdots = f^{(m-1)}(\mathbf{x}) = 0$, where each $f^{(i)}$ is a polynomial of degree d in $\mathbf{x} = (x_0, \ldots, x_{n-1})$. All coefficients and variables are in $\mathbb{F}_2$.

When $d = 2$, this problem is usually abbreviated as $\mathcal{MQ}$ (multivariate quadratic), which is proved to be NP-complete.

1.3 Contributions

Our contribution is twofold. On the theoretical side, we present an exhaustive search algorithm which is both asymptotically and practically faster than existing techniques. If we ignore the (usually negligible) cost for initialization, finding all zeroes of a single degree-d polynomial in n variables requires $O(d \cdot 2^n)$ bit operations. We can extend it and find the common zeroes of arbitrary number of degree-d polynomials in $O((d^2 + 2d + 2) \cdot 2^n)$ bit operations.

Our algorithm also possesses many other advantages. For one, it can be easily parallelized with negligible cost, which means it is quite suitable for hardware

platforms dedicated for parallel computing such as GPUs. Also, the time complexity grows linearly as the degree d increases. All these are strong advantages over Gröbner-basis methods.

On the practical side, we have implemented our algorithms on x86 CPUs and on NVIDIA GPUs. While our CPU implementation is fairly optimized using SIMD instructions, our GPU implementation running on one single NVIDIA GeForce GTX 295 graphics card runs up to nine times faster than the CPU implementation using all the cores of an Intel quad-code Core i7, one of the fastest CPUs currently available.

Today, we can solve 48+ quadratic equations in 48 binary variables using just an NVIDIA GeForce GTX 295 graphics card in 21 minutes, a device currently available for about 500 USD. It would be 36 minutes for cubic equations and two hours for quartics. The 64-bit Dragon signature challenge [22] can thus be broken with 10 such cards in 3 months, using a budget of 5000 USD. Even taking into account Moore's law, this is still quite an achievement.

In contrast, the implementation of F_4 in `MAGMA-2.15-5`, often cited as the best Gröbner-basis solver available today, requires more than 64 GB of memory to solve 25 cubic equations in as many $\mathbb{F}_2$-variables. When it does not run out of memory, it requires 2.5 hours to solve 20 cubic equations in 20 variables on one Opteron core running at 2.2 GHz, half an hour for 45 quadratic equations with 30 variables, or 7 minutes for 60 quadratic equations with 30 variables. Each of the above are solved in less than a second using negligible memory via our implementation of the exhaustive search algorithm on the same CPU.

Chapter 2

Gray Code Enumeration and Partial Evaluation

2.1 Notational Conventions

Our goal is to solve a polynomial system $f = (f^{(0)}, \ldots, f^{(m-1)})$ of the form described in section 1.2. We will use $\mathbf{C}^{(j)}_{\beta_1,\beta_2,\ldots,\beta_k}$ to denote the coefficient of the monomial $x_{\beta_1}x_{\beta_2}\cdots x_{\beta_k}$ of $f^{(j)}$ (we use $\mathbf{C}^{(j)}$ for the constant term), where $0 \le \beta_1 < \beta_2 < \cdots < \beta_k < n$ since any x_β^α where $\alpha \ge 1$ can be reduced to x_i in $\mathbb{F}_2$. When the superscript is omitted, it stands for a vector of coefficients, i.e., $\mathbf{C}_* = (\mathbf{C}_*^{(0)}, \ldots, \mathbf{C}_*^{(m-1)})$.

2.2 Gray Code

A k-bit Gray code is a special ordering of the binary numbers ranging from 0 to $2^k - 1$ such that the Hamming distance between any two successive numbers is exactly 1. Such a code is not unique, and the one we use is known as the binary-reflected Gray code (BRGC). Table 2.1 shows a list of 5-bit codewords along with their corresponding indices, where the b_i columns will be explained in the following definition.

Definition 1. *Let i be an nonnegative integer written in binary expansion (usually an index of Gray code), then $b_k(i)$ is defined as the index of the k-th least significant*

nonzero bit in i. If the Hamming weight of i is less than k, $b_k(i)$ is defined as -1.

We use $\mathbf{g}_i$ to denote the equivalent vector form of the codeword corresponding to the index i. Note that $\mathbf{g}_0 = 0$ is always true for BRGC. Thus, $\mathbf{g}_i$ can be defined recursively by the following equation for every $i > 0$:

$$\mathbf{g}_{i+1} = \mathbf{g}_i + \mathbf{e}_{b_1(i+1)}, \tag{2.1}$$

where $\mathbf{e}_j$ is a binary vector consisting of all zeros except in the j-th position.

Another way to define the codewords is to convert directly from an index to its corresponding codeword. Let $\mathbf{c} = \mathbf{g}_i$, then each bit of $\mathbf{c}$ can be derived from i according to the following equation:

$$\mathbf{c}_k = i_k \ \texttt{xor}\ i_{k+1}, \tag{2.2}$$

where the subscripts stand for bit indices.

Table 2.1: 5-bit Gray Code with Index and Enumeration Actions

index	code	b_1	b_2	b_3	b_4	actions(quadratic)	actions(quartic)
00000	00000	-1	-1	-1	-1		
00001	00001	0	-1	-1	-1	δ += δ_0	δ += δ_0
00010	00011	1	-1	-1	-1	δ += δ_1	δ += δ_1
00011	00010	0	1	-1	-1	δ += (δ_0 += $\mathbf{C}_{0,1}$)	δ += (δ_0 += $\delta_{0,1}$)
00100	00110	2	-1	-1	-1	δ += δ_2	δ += δ_2
00101	00111	0	2	-1	-1	δ += (δ_0 += $\mathbf{C}_{0,2}$)	δ += (δ_0 += $\delta_{0,2}$)
00110	00101	1	2	-1	-1	δ += (δ_1 += $\mathbf{C}_{1,2}$)	δ += (δ_1 += $\delta_{1,2}$)
00111	00100	0	1	2	-1	δ += (δ_0 += $\mathbf{C}_{0,1}$)	δ += (δ_0 += ($\delta_{0,1}$ += $\delta_{0,1,2}$))
01000	01100	3	-1	-1	-1	δ += δ_3	δ += δ_3
01001	01101	0	3	-1	-1	δ += (δ_0 += $\mathbf{C}_{0,3}$)	δ += (δ_0 += $\delta_{0,3}$)
01010	01111	1	3	-1	-1	δ += (δ_1 += $\mathbf{C}_{1,3}$)	δ += (δ_1 += $\delta_{1,3}$)
01011	01110	0	1	3	-1	δ += (δ_0 += $\mathbf{C}_{0,1}$)	δ += (δ_0 += ($\delta_{0,1}$ += $\delta_{0,1,3}$))
01100	01010	2	3	-1	-1	δ += (δ_2 += $\mathbf{C}_{2,3}$)	δ += (δ_2 += $\delta_{2,3}$)
01101	01011	0	2	3	-1	δ += (δ_0 += $\mathbf{C}_{0,2}$)	δ += (δ_0 += ($\delta_{0,2}$ += $\delta_{0,2,3}$))
01110	01001	1	2	3	-1	δ += (δ_1 += $\mathbf{C}_{1,2}$)	δ += (δ_1 += ($\delta_{1,2}$ += $\delta_{1,2,3}$))
01111	01000	0	1	2	3	δ += (δ_0 += $\mathbf{C}_{0,1}$)	δ += (δ_0 += ($\delta_{0,1}$ += ($\delta_{0,1,2}$ += $\mathbf{C}_{0,1,2,3}$)))
10000	11000	4	-1	-1	-1	δ += δ_4	δ += δ_4
10001	11001	0	4	-1	-1	δ += (δ_0 += $\mathbf{C}_{0,4}$)	δ += (δ_0 += $\delta_{0,4}$)
10010	11011	1	4	-1	-1	δ += (δ_1 += $\mathbf{C}_{1,4}$)	δ += (δ_1 += $\delta_{1,4}$)
10011	11010	0	1	4	-1	δ += (δ_0 += $\mathbf{C}_{0,1}$)	δ += (δ_0 += ($\delta_{0,1}$ += $\delta_{0,1,4}$))
10100	11110	2	4	-1	-1	δ += (δ_2 += $\mathbf{C}_{2,4}$)	δ += (δ_2 += $\delta_{2,4}$)
10101	11111	0	2	4	-1	δ += (δ_0 += $\mathbf{C}_{0,2}$)	δ += (δ_0 += ($\delta_{0,2}$ += $\delta_{0,2,4}$))
10110	11101	1	2	4	-1	δ += (δ_1 += $\mathbf{C}_{1,2}$)	δ += (δ_1 += ($\delta_{1,2}$ += $\delta_{1,2,4}$))
10111	11100	0	1	2	4	δ += (δ_0 += $\mathbf{C}_{0,1}$)	δ += (δ_0 += ($\delta_{0,1}$ += ($\delta_{0,1,2}$ += $\mathbf{C}_{0,1,2,4}$)))
11000	10100	3	4	-1	-1	δ += (δ_3 += $\mathbf{C}_{3,4}$)	δ += (δ_3 += $\delta_{3,4}$)
11001	10101	0	3	4	-1	δ += (δ_0 += $\mathbf{C}_{0,3}$)	δ += (δ_0 += ($\delta_{0,3}$ += $\delta_{0,3,4}$))
11010	10111	1	3	4	-1	δ += (δ_1 += $\mathbf{C}_{1,3}$)	δ += (δ_1 += ($\delta_{1,3}$ += $\delta_{1,3,4}$))
11011	10110	0	1	3	4	δ += (δ_0 += $\mathbf{C}_{0,1}$)	δ += (δ_0 += ($\delta_{0,1}$ += ($\delta_{0,1,3}$ += $\mathbf{C}_{0,1,3,4}$)))
11100	10010	2	3	4	-1	δ += (δ_2 += $\mathbf{C}_{2,3}$)	δ += (δ_2 += ($\delta_{2,3}$ += $\delta_{2,3,4}$))
11101	10011	0	2	3	4	δ += (δ_0 += $\mathbf{C}_{0,2}$)	δ += (δ_0 += ($\delta_{0,2}$ += ($\delta_{0,2,3}$ += $\mathbf{C}_{0,2,3,4}$)))
11110	10001	1	2	3	4	δ += (δ_1 += $\mathbf{C}_{1,2}$)	δ += (δ_1 += ($\delta_{1,2}$ += ($\delta_{1,2,3}$ += $\mathbf{C}_{1,2,3,4}$)))
11111	10000	0	1	2	3	δ += (δ_0 += $\mathbf{C}_{0,1}$)	δ += (δ_0 += ($\delta_{0,1}$ += ($\delta_{0,1,2}$ += $\mathbf{C}_{0,1,2,3}$)))

2.3 Naïve Evaluation

Let $\mathbf{v}_i$ be the equivalent vector form of an integer i. The most naïve way to search exhaustively is to evaluate $f(\mathbf{v}_i)$ for $i = 0, \ldots, 2^n - 1$. Determining whether $\mathbf{v}_i$ is a valid solution can be easily done by checking whether $f(\mathbf{v}_i) = 0$ or not. The pseudocode of this scheme is presented in Fig. 2.1.

An advantage of this scheme is that it takes little extra memory than storing the coefficients. Since the coefficients can be stored in read-only memory, we may define the required amount of storage in bits (per equation) to perform this scheme by:

$$M_{Eval}^{(ro)}(n, d) = \sum_{i=0}^{d} \binom{n}{i} \tag{2.3}$$

where the superscript $^{(ro)}$ is short for "read-only."

In contrast, each evaluation of $f(\mathbf{v}_i)$ can be expensive in time complexity. If the multivariate Horner's rule is used to evaluate the function value at each vector, the number of bit operations required (per equation) would be:

$$B_{Eval\_Horner}(n, d) = 2 \sum_{i=1}^{d} \binom{n}{i}. \tag{2.4}$$

An alternative, theoretically optimal way would be summing up all $\mathbf{C}_{\beta_1,\ldots,\beta_k}$'s with $(\mathbf{v}_i)_{\beta_1} = \cdots = (\mathbf{v}_i)_{\beta_k} = 1$. Apparently, the bit operations (per equation) required for this scheme depend on the Hamming weight of $\mathbf{v}_i$, which we denote as h:

$$B_{Eval\_Sum}(h, d) = \sum_{i=0}^{d} \binom{h}{i}. \tag{2.5}$$

If $\mathbf{v}_i$ follows a uniform distribution in $\mathbb{F}_2^n$, then the expected bit operations (per equation) required would be:

$$B_{Eval\_Sum\_Avg}(n, d) = \sum_{i=0}^{d} \binom{n}{i} 2^{-i}. \tag{2.6}$$

Thus, total bit operations (per equation) required in naïve evaluation would be:

$$B_{Eval}(n,d) = \sum_{i=0}^{d} \binom{n}{i} 2^{n-i}. \tag{2.7}$$

In summary, when d is a fixed constant, each evaluation of $f(\mathbf{v}_i)$ should take $O(mn^d)$ on average. The whole enumeration would thus require $O(mn^d 2^n)$ bit operations.

Figure 2.1: Pseudocode of Naïve Evaluation

```
NaïveEval(f, n, d)
 1: Sol ← ∅
 2: for i from 0 to 2^n − 1 do
 3:     δ ← f(v_i)
 4:     if δ = 0 then
 5:         Sol ← Sol ∪ {v_i}
 6:     end if
 7: end for
 8: return Sol
```

2.4 Basic Gray Code Enumeration

The following definition and proposition are important to the our discussion.

Definition 2. *Let* $\mathbf{v}, \mathbf{w}$ *be vectors over* $\mathbb{F}_2$ *(or equivalently integers written in binary expansion) and* f *be a vector of multivariate polynomials over* $\mathbb{F}_2$*. Let* $S = \{j \mid f \text{ contains } x_j\}$*, then* $\mathbf{v}$ *and* $\mathbf{w}$ *are said to be* i*-close (or simply close) w.r.t.* f *if* i *is the only element in* S *such that* $\mathbf{v}_i \neq \mathbf{w}_i$*. This definition can be extended to refer to a sequence of vectors where any two successive vectors are close.*

For instance, 101 and 110 are 1-close w.r.t. $x_1x_2 + 1$.

Proposition 1. *Let* f *be a vector of multivariate polynomials over* $\mathbb{F}_2$*. Let* $\mathbf{v}$ *and* $\mathbf{w}$ *be* i*-close w.r.t.* f*. Then* $f(\mathbf{v}) - f(\mathbf{w}) = \frac{\partial f}{\partial x_i}(\mathbf{v}) = \frac{\partial f}{\partial x_i}(\mathbf{w})$.

Proof. f can be converted into the form $f = \frac{\partial f}{\partial x_i} x_i + h$. Since $\mathbf{v}$ and $\mathbf{w}$ are i-close w.r.t. f, we have:

$$
\begin{aligned}
& f(\mathbf{v}) - f(\mathbf{w}) \\
&= \tfrac{\partial f}{\partial x_i}(\mathbf{v})\mathbf{v}_i - \tfrac{\partial f}{\partial x_i}(\mathbf{w})\mathbf{w}_i + h(\mathbf{v}) - h(\mathbf{w}) \\
&= \tfrac{\partial f}{\partial x_i}(\mathbf{v})\mathbf{v}_i - \tfrac{\partial f}{\partial x_i}(\mathbf{v})\overline{\mathbf{v}_i} + h(\mathbf{v}) - h(\mathbf{v}) \\
&= \tfrac{\partial f}{\partial x_i}(\mathbf{v}) \\
&= \tfrac{\partial f}{\partial x_i}(\mathbf{w})
\end{aligned}
$$

□

Now we can make use of this proposition to construct a better exhaustive search algorithm, which we shall refer to as the basic Gray code enumeration (BGCE) algorithm. As the name implies, the candidate vectors are tested in the order of Gray code instead of a counter. That is, instead of evaluating $f(\mathbf{v}_i)$, we compute $f(\mathbf{g}_i)$ for $i = 0, \ldots, 2^n - 1$. Since any two successive codewords $(\mathbf{g}_{i-1}, \mathbf{g}_i)$ are $b_1(i)$-close w.r.t. f by Eq. 2.1, the difference between $(f(\mathbf{g}_i), f(\mathbf{g}_{i-1}))$ is actually $\frac{\partial f}{\partial x_{b_1}}(\mathbf{g}_i)$ by Proposition 1. This implies that we can evaluate the next $f(\mathbf{g}_i)$ by updating the last one with their difference. In this way, the bit operations (per equation) required to perform this schemes would be:

$$
B_{BGCE}(n, d) = 2^n B_{Eval\_Sum\_Avg}(n-1, d-1) = \sum_{i=0}^{d-1} \binom{n-1}{i} 2^{n-i}. \tag{2.8}
$$

In other words, when d is a fixed constant, the cost of each attempt can be reduced to $O(mn^{d-1})$, and the whole enumeration would require $O(mn^{d-1}2^n)$ bit operations.

The pseudocode of this scheme is shown in Fig. 2.2(a). Note that $\beta_1 < 0$ if and only if $i = 0$, which means this is the first time of computing $f(\mathbf{g}_i)$ (there is no $f(\mathbf{g}_{i-1})$ for computing a difference). In this case, the image we need is actually $f(\mathbf{g}_0) = f(0) = \mathbf{C}$. Thus, δ is initialized to $\mathbf{C}$ in line 2, and line 6 will not be executed in the first attempt.

Figure 2.2: Pseudocode of Gray Code Enumeration

```
BGCE(f, n, d)
 1: Sol ← ∅
 2: δ ← C
 3: for i from 0 to 2^n − 1 do
 4:     β_1 ← b_1(i)
 5:     if β_1 ≥ 0 then
 6:         δ ← δ + ∂f/∂x_{β_1}(g_i)
 7:     end if
 8:     if δ = 0 then
 9:         Sol ← Sol ∪ { g_i }
10:     end if
11: end for
12: return Sol
```

(a) Basic Gray Code Enumeration

```
GGCE(f, n, d)
 1: Sol ← ∅
 2: for each coefficient C_{β_1,...,β_k} of f
 3:     δ_{β_1,...,β_k} ← ∂^k f/(∂x_{β_1}···∂x_{β_k})(g_{2^{β_1}+···+2^{β_k}})
 4: end if
 5: for i from 0 to 2^n − 1 do
 6:     α ← min(HammingWeight(i), d)
 7:     β_1, ..., β_α ← b_{1,...,α}(i)
 8:     for j from α down to 1 do
 9:         δ_{β_1,...,β_{j−1}} ← δ_{β_1,...,β_{j−1}} + δ_{β_1,...,β_j}
10:     end for
11:     if δ = 0 then
12:         Sol ← Sol ∪ { g_i }
13:     end if
14: end for
15: return Sol
```

(b) Generalized Gray Code Enumeration

2.5 Generalized Gray Code Enumeration

Algorithm and Correctness. In the last section, we have shown that evaluations of a system for a sequence of vectors can be accelerated if the vectors in the sequence are "close" in certain way. In this section, we will show that the same technique can be used recursively, resulting in the generalized Gray code enumeration (GGCE) algorithm with a complexity much lower than BGCE.

The pseudocode of GGCE is shown in Fig. 2.2(b). At the beginning of the code (line 2 to line 4), a set of variables $\delta_{\beta_1,\ldots,\beta_k}$'s are initialized. As we shall explain later, at the end of an attempt accessing $\delta_{\beta_1,\ldots,\beta_k}$, its value would always be $\frac{\partial^k f}{\partial x_{\beta_1}\cdots\partial x_{\beta_k}}(\mathbf{g}_i)$. Thus, same as in BGCE as shown in Fig. 2.2(a), δ is meant to store $f(\mathbf{g}_i)$. Each of these variables will be referred to as a "differential" henceforth.

Each attempt (line 6 to line 13) of GGCE can be divided into three steps.

1. (Line 6–7) Indexing: Find indices of α least significant nonzero bits in i.
2. (Line 8–10) Accumulating: Use a sequence of differentials determined by the indices and perform an in-place prefix-sum operation.
3. (Line 11–13) Testing: Examine the image δ, which is identical to line 8–10 in Fig. 2.2(a).

The accumulating is where we use the technique in the last section recursively to compute $f(\mathbf{g}_i)$. Actions taken in this step (with $\alpha \geq 3$) can be illustrated by the following expression:

$$\delta += (\delta_{\beta_1} += (\delta_{\beta_1,\beta_2} += (\delta_{\beta_1,\beta_2,\beta_3} += \cdots))).$$

The action $\delta += \delta_{\beta_1}$ is consistent with line 6 of Fig. 2.2(a), since the value of δ_{β_1} should be $\frac{\partial f}{\partial x_{\beta_1}}(\mathbf{g}_i)$. However, δ_{β_1} is not naïvely evaluated. Instead, it is updated by adding δ_{β_1,β_2} to it. In the same way, we update $\delta, \delta_{\beta_1}, \delta_{\beta_1,\beta_2}, \ldots$, etc, until some termination condition is satisfied. We will come back to this condition later in this section.

Definition 3. *Given a sequence I of indices running from 0 to $2^n - 1$, the subsequence $I_{j_1,j_2,\ldots,j_k}$ $(0 \leq j_1 < j_2 < \cdots < j_k < n)$ consists of $\{i \in I \mid b_1(i) = j_1, \ldots, b_k(i) = j_k\}$.*

Lemma 1. *For all $i \in I_{j_1,j_2,\ldots,j_k}$, $b_{j_k+1}(i) \geq 0$ if and only if i is not the first index in the sequence.*

Proof. Apparently, the first element in $I_{j_1,j_2,\ldots,j_k}$ is $2^{j_1} + \cdots + 2^{j_k}$, for it is the smallest index satisfying the condition of the sequence. Since this is the only element with Hamming weight k, we may conclude that other elements must have Hamming weight greater than k. □

Lemma 2. *$I_{j_1,j_2,\ldots,j_k}$ $(k \leq d)$ consists of the indices of all attempts in which $\delta_{j_1,\ldots,j_k}$ is accessed.*

Proof. According to line 6 to line 10 of the pseudocode, $\delta_{j_1,\ldots,j_k}$ is accessed if and only if $b_1(i) = j_1, \ldots, b_k(i) = j_k$. □

Lemma 3. *Let f be a multivariate polynomial system over $\mathbb{F}_2$. For any (i, i') successive in $I_{j_1,j_2,\ldots,j_k}$, $\mathbf{g}_i$ and $\mathbf{g}_{i'}$ are $b_{k+1}(i')$-close w.r.t. $\frac{\partial^k f}{\partial x_{j_1} \cdots \partial x_{j_k}}$.*

Proof. It can be inferred from Definition 3 that $i' = i + 2^{j_k+1}$. According to Eq. 2.2, the bits with indices greater than j_k of $\mathbf{g}_i$ and $\mathbf{g}_{i'}$ should differ in exactly the bit with

index $b_{k+1}(i')$, and those with indices less than j_k of $\mathbf{g}_i$ and $\mathbf{g}_{i'}$ should be identical. Thus, $\mathbf{g}_i$ and $\mathbf{g}_{i'}$ must be $b_{k+1}(i')$-close w.r.t. any system that does not involve x_{j_k}.

□

Proposition 2. *Let $1 \leq j \leq d$. If $\delta_{\beta_1,\ldots,\beta_j}$'s can be correctly computed (equal to $\frac{\partial^j f}{\partial x_{\beta_1}\cdots\partial x_{\beta_j}}(\mathbf{g}_i)$ by the end of any attempt accessing them), so can $\delta_{\beta_1,\ldots,\beta_{j-1}}$'s.*

Proof. According to the pseudocode, $\delta_{\beta_1,\ldots,\beta_{j-1}}$ is updated by adding $\delta_{\beta_1,\ldots,\beta_j}$ to it in those attempts with index $i \in I_{\beta_1,\ldots,\beta_{j-1}}$ and $b_j(i) \geq 0$. According to lemma 1, lemma 2 and lemma 3, from the second attempt accessing $\delta_{\beta_1,\ldots,\beta_{j-1}}$ it is updated by a correct difference $\delta_{\beta_1,\ldots,\beta_j} = \frac{\partial^j f}{\partial x_{\beta_1}\cdots\partial x_{\beta_j}}(\mathbf{g}_i)$, which is exactly the core concept in BGCE. Thus, the only remaining problem is that $\delta_{\beta_1,\ldots,\beta_{j-1}}$ should be correctly computed in the first attempt accessing it. According to the pseudocode and lemma 1, $\delta_{\beta_1,\ldots,\beta_{j-1}}$ is not modified in the first attempt accessing it. Thus, line 3 has shown that $\delta_{\beta_1,\ldots,\beta_{j-1}}$ contains the correct value by the end of the first attempt accessing it.

□

Note that the differentials $\delta_{\beta_1,\ldots,\beta_d}$ are initialized to $\frac{\partial f}{\partial x_{\beta_1}\cdots\partial x_{\beta_d}} = \mathbf{C}_{\beta_1,\ldots,\beta_d}$. Since these differentials remain the same values after initialization (they would not be modified by line 9) as they ought to be, they always contain the correct values. Thus, by using this fact as the base case and Proposition 2 as the inductive step, it can be concluded that δ can be correctly computed, establishing the correctness of GGCE.

Now we may explain the termination condition that we have mentioned earlier. Note the first iteration of line 9:

$$\delta_{\beta_1,\ldots,\beta_{\alpha-1}} \leftarrow \delta_{\beta_1,\ldots,\beta_{\alpha-1}} + \delta_{\beta_1,\ldots,\beta_\alpha}.$$

From a recursive perspective, this is already the deepest level of recursion, and $\delta_{\beta_1,\ldots,\beta_\alpha}$ is not updated using the same technique. There are two possible situations for this. The first one is when $\alpha = d$, which means $\delta_{\beta_1,\ldots,\beta_\alpha}$ is a constant and should not be updated. The other one is when the candidate vector $\mathbf{g}_i$ is the first one in the

sequence close w.r.t. $\frac{\partial^\alpha f}{\partial x_{\beta_1} \cdots \partial x_{\beta_\alpha}}$. Therefore there is no previous image to compute a difference, in which case the image is pre-evaluated in the initialization phase.

To help the reader further understand how the GGCE algorithm works, we show an example with a small d. In the example, the list of accumulating actions in each attempt for $d = 2$ and $d = 4$ are presented in the last two columns of Table 2.1.

Memory Issues. The differentials take almost all memory needed in GGCE. Thus, according to line 2 to line 4 of Fig. 2.2(b), memory needed for GGCE should be commensurate with memory needed to store all the coefficients.

One may argue that we need additional memory to store the target system (the coefficients) to perform initialization. However, since $\frac{\partial^k f}{\partial x_{\beta_1} \cdots \partial x_{\beta_k}}$ does not involve any of $\mathbf{C}_{\beta_1,\ldots,\beta_{k'}}$'s where $k' \leq k$ except $\mathbf{C}_{\beta_1,\ldots,\beta_k}$, the initialization process can be made "in-place" by initializing d using the space of $\mathbf{C}$, initializing δ_{β_1}'s using the spaces of $\mathbf{C}_{\beta_1}$'s, and so on. In this way, the initialization can be done with only the memory space of coefficients of the target system.

To be precise, the $\delta_{\beta_1,\ldots,\beta_d}$'s ($\mathbf{C}_{\beta_1,\ldots,\beta_d}$'s) can be stored in read-only memory, while other differentials should be stored in read-write memory. Thus, number of bits required (per equation) to perform GGCE is defined as:

$$M_{GGCE}^{(ro)}(n,d) = \binom{n}{d}, \quad M_{GGCE}^{(rw)}(n,d) = \sum_{i=0}^{d-1} \binom{n}{i}. \tag{2.9}$$

where the superscript $^{(rw)}$ is apparently short for "read-write."

We note that in this sense, GGCE is providing a form of space-time trade-off, a renowned technique in exhaustive-search type of cryptanalysis. However, the efficiency of GGCE in trading memory for execution time is extraordinary. By using roughly the same amount of read-write memory, GGCE can achieve a speed-up that is several orders of magnitudes faster than the naïve enumeration.

Indexing. The pseudocode for indexing merely describes "what it actually does," not "how it works." Thus in this paragraph, we would like to show a way to construct

the indexing.

The main idea is to maintain a stack S containing all indices of nonzero bits in i. For example, if $i = (10101)_2$, then S should contain three elements 0, 2, and 4 (from top to bottom). When a new attempt starts, i would increase by 1. When this happens, we can update S easily using a procedure Update presented in Fig. 2.3. Note that the cost for Update is directly proportional to number of the bits flipped as i increases. Thus, an amortized analysis can show that the function can be done in constant time.

Figure 2.3: Pseudocode of Update

```
Update(S)
 1: j ← 0
 2: while top(S)=j do
 3:     pop(S)
 4:     j ← j + 1
 5: end while
 6: push(S,j)
```

Computing α would be easy since the Hamming weight of i always equals to number of elements in the stack. After that the indexing is done, as for all $1 \leq j \leq \alpha$, β_j must be the $j - th$ element in the stack. Thus, the whole indexing can actually be done in constant time.

Time Complexity. Before determining the complexity of the whole enumeration, we should first argue that number of bit operations (per equation) for initialization can be bounded by:

$$B_{GGCE\_Init}(n,d) = \sum_{i=0}^{d-1} \binom{n}{i} B_{Eval\_Sum}(i, d-i) = \sum_{i=0}^{d-1} \binom{n}{i} \sum_{j=0}^{d-i} \binom{i}{j}. \tag{2.10}$$

Now consider the in-place initialization scheme we have previously mentioned. Note that initializations for $\delta_{\beta_1,\ldots,\beta_d}$'s do not take any bit operation since they should be initialized to $\mathbf{C}_{\beta_1,\ldots,\beta_d}$'s. Any of other differentials, say $\delta_{\beta_1,\ldots,\beta_k}$ (where $1 \leq k \leq$

$d-1$), should be initialized to

$$\frac{\partial^k f}{\partial x_{\beta_1}\cdots\partial x_{\beta_k}}(\mathbf{g}_{2^{\beta_1}+\cdots+2^{\beta_k}}) = \frac{\partial^k f}{\partial x_{\beta_1}\cdots\partial x_{\beta_k}}((\mathbf{e}_{\beta_1}+\mathbf{e}_{\beta_1-1})+\cdots+(\mathbf{e}_{\beta_1}+\mathbf{e}_{\beta_1-1}))$$

$$= \frac{\partial^k f}{\partial x_{\beta_1}\cdots\partial x_{\beta_k}}(\mathbf{e}_{\beta_1-1}+\cdots+\mathbf{e}_{\beta_k-1})$$

according to Eq. 2.2. Now, it is clear that initialization of $\delta_{\beta_1,\ldots,\beta_k}$ can be done by evaluating a system of degree $d-k$ for a vector with k nonzero entries. Thus, we may conclude that bit operations (per equation) for initialization can be expressed by Eq. 2.10.

In one iteration of accumulating, $\alpha = \min(\text{HammingWeight}(i), d)$ m-bit vector additions (XORs) would be executed. Thus, total bit operations (per equation) needed in GGCE can be expressed by:

$$B_{GGCE}(n,d) = B_{GGCE\_Init}(n,d) + \sum_{i=0}^{n}\binom{n}{i}\min(i,d). \tag{2.11}$$

Since the summation can be bounded by $d\cdot 2^n$ (which should be tight when d is small), the time complexity of GGCE can be bounded by $O(mB_{GGCE\_Init}(n,d) + md2^n)$. When d is a constant, in which case $B_{GGCE\_Init}(n,d)$ would be polynomial in n, the complexity of GGCE would be $O(m2^n)$.

2.6 Partial Evaluation

In the last few sections, we introduced various exhaustive-search solvers intended for solving a single system. However, as mentioned in section 1, parallel computing is a more efficient way of taking advantage of Moore's law. This leads to the need of parallelization. In other words, we need to divide our problem into pieces, so that resulting subproblems can be solved concurrently.

An intuitive idea is partial evaluation. That is, we shall divide the target system into multiple subsystems by substituting all possible values for s variables. In this

way, there would be 2^s subsystems, each with $n-s$ variables, and the number and size of subsystems can be controlled simply by changing s. Moreover, we have found that partial evaluation can be made efficient by using GGCE as a subroutine.

Recall that we use $\mathbf{c}_*$ for coefficients of subsystems and $\mathbf{C}_*$ for those of the original system. Now consider a specific coefficient, say $\mathbf{c}$, of one of the subsystems. The coefficient is an image of a polynomial system h defined over the substituted variables. In fact, if we collect the same coefficient in all subsystems, the resulting set actually forms the range of h, which can be computed by GGCE (without testing) since it generates all images of a system. Thus, partial evaluation can be done by computing with GGCE all $\mathbf{c}$'s, $\mathbf{c}_0$'s, $\mathbf{c}_1$'s, ..., etc, until all $\sum_{i=0}^{d}\binom{n-s}{i}$ coefficients of each subsystem are known.

We use an example to show this concept more clearly. Let us consider a case with $n=4$, $d=2$, and $s=2$, where the variables to be substituted are x_2 and x_3. Now, the original system can be written in the following expression:

$$\mathbf{C}_{0,1}x_0x_1+(\mathbf{C}_{0,2}x_2+\mathbf{C}_{0,3}x_3+\mathbf{C}_0)x_0+(\mathbf{C}_{1,2}x_2+\mathbf{C}_{1,3}x_3+\mathbf{C}_1)x_1+(\mathbf{C}_{2,3}x_2x_3+\mathbf{C}_2x_2+\mathbf{C}_3x_3+\mathbf{C}).$$

It can be inferred from the expression that the c's form the range of $\mathbf{C}_{2,3}x_2x_3 + \mathbf{C}_2x_2 + \mathbf{C}_3x_3 + \mathbf{C}$, the $\mathbf{c}_1$'s form the range of $\mathbf{C}_{1,2}x_2 + \mathbf{C}_{1,3}x_3 + \mathbf{C}_1$, ..., and so on.

Note that the degree-d terms of all subsystems actually come from the original system. In other words, $\mathbf{c}_{\alpha_1,\ldots,\alpha_d} = \mathbf{C}_{\alpha_1,\ldots,\alpha_d}$ for any subsystem. Thus, we do not need to evaluate $\mathbf{c}_{\alpha_1,\ldots,\alpha_d}$'s by GGCE. Instead, we may simply copy from the original system those coefficients when we need them.

Time complexity. According to our arguments and example, it is clear that coefficients (in subsystems) of a degree-k monomial can be generated by running GGCE on a degree-$(d-k)$ system defined over the s substituted variables. Thus, the bit operations required (per equation) for partial evaluation can be expressed by:

$$B_{Partial}(n,d,s) = \sum_{k=0}^{d-1}\binom{n-s}{k} B_{GGCE}(s,d-k). \tag{2.12}$$

Memory Issues. Since in partial evaluation, the generation of $\mathbf{c}'s$ is done by running GGCE on a degree-d system with s variables, at most $M^{(ro)}_{GGCE}(s,d) + M^{(rw)}_{GGCE}(s,d)$ bits per equation is required to complete the whole process. While this memory cost is usually affordable, memory problems brought by partial evaluation are usually due to the subsystems it generates. To be exact, the subsystems should take

$$\binom{n-s}{d} + 2^s \sum_{i=0}^{d-1} \binom{n-s}{i}$$

bits per equation, where $\binom{n-s}{d}$ bits are for $c_{\alpha_1,\ldots,\alpha_d}$'s. Thus, when s is sufficiently large, the subsystems can take a huge amount of memory. However, sometimes we do not need all the subsystems at the same time, in which case there are at least two ways to mitigate the memory problem.

The first solution is to perform a multi-level partial evaluation, which is suitable when only $2^{s'}$ of all 2^s subsystems need to be dealt with at the same time. A two-level partial evaluation goes like this. First we divide the target system into $2^{s-s'}$ "intermediary" systems. Then, we pick one of them at a time and divide it into $2^{s'}$ "final" systems which are meant to be processed together. In this way, as long as the intermediary systems do not take much memory, we only need the memory for the $2^{s'}$ final systems.

The second solution is to run all instances of GGCE (in partial evaluation) at the same time, which might be useful when the 2^s subsystems are dealt with one by one. Note that attempts with the same index of all instances actually generate all coefficients of the same subsystem. Thus, by running all instances synchronously, we may generate the subsystems one by one. In fact, there is a one-to-one mapping between the terms in the original system and the terms in all coefficient-generating systems (h's); specifically, there is a one-to-one mapping between the highest-degree terms in coefficient-generating systems and the degree-d terms in the original system. Thus, by the discussions about memory issues in GGCE, we may conclude that this scheme uses the same amount of read-only and read-write memory with running GGCE directly on the original system.

Chapter 3

Variants and Analysis

3.1 Early-abort Strategy

3.1.1 In Naïve Evaluation

While the naïve evaluation has been proved to be outperformed by several solvers, it can actually be improved by taking advantage of an early-abort strategy. All we need to do is to treat the equations as a sequence of candidate filters, and each candidate vector in $\mathbb{F}_2^n$ would be examined by the filters one by one until it is filtered out. Let $V^{(0)} = \mathbb{F}_2^n$, the initial space of candidate vectors. Formally speaking, for each $f^{(i)}$ we can compute $f^{(i)}(V^{(i)})$ and arrive at $V^{(i+1)} = \{\mathbf{v} \in V^{(i)} \mid f^{(i)}(\mathbf{v}) = 0\}$. Since on average each candidate vector is filtered out with probability 0.5, we only need to examine two filters on average. Consequently, the average number of bit operations required would be:

$$2 \cdot B_{Eval}(n, d) = \sum_{i=0}^{d} \binom{n}{i} 2^{n-i+1}. \tag{3.1}$$

3.1.2 In GGCE

The way we treat equations as filters is apparently not suitable for GGCE, for it needs to enumerate all $\mathbb{F}_2^n$. However, can GGCE be modified so that it computes only $f^{(i)}(V^{(i)})$? One method that has come across our mind goes like this. To

compute $f^{(i)}(V^{(i)})$, $f^{(i)}$ is first partially evaluated with some well-chosen $s^{(i)}$. Then, for each $\mathbf{v} \in V^{(i)}$, $f^{(i)}(\mathbf{v})$ can be evaluated by substituting $n - s$ bits of $\mathbf{v}$ into the corresponding subsystem.

Since the costs for partial evaluation and naïve evaluation are known, the number of bit operations required for computing $f^{(i)}(V^{(i)})$ can be expressed by:

$$B_{Partial}(n, d, s^{(i)}) + 2^{(n-i)} \cdot B_{Eval\_Sum\_Avg}(n - s^{(i)}, d),$$

where $2^{(n-i)}$ stands for the expected number of $|V^{(i)}|$. According to this expression, the cost of computing $f^{(i)}(V^{(i)})$ is fully dependent of $s^{(i)}$. Therefore, minimizing the total number of bit operations required can be simply done by finding the best $s^{(i)}$ for each i independently. Unfortunately, after trying mathematical techniques such as the first derivative test, we found it hard to express the best $s^{(i)}$ in a closed general form. Thus, we use an empirical approach instead, in which we search for the best $s^{(i)}$ in the interval $[0, n]$ for each i. The same procedure can be repeated several times for different settings of (m, n, d) to gain enough generality. According to our experiment result, the sequence $[s^{(0)}, s^{(1)}, l \ldots, s^{(m-1)}]$ is usually in the pattern of $[n, n, n-k_1, n-k_1-1, \ldots, k_2, 0, \ldots, 0]$, where k_1, k_2 are some small positive integer. This implies that $[n, n, n-1, n-2, \ldots, 1, 0, \ldots, 0]$ might be generally a good choice. Thus, we can approximate the total number of bit operations required with:

$$\begin{aligned}
&\textstyle\sum_{i=0}^{m-1} [B_{Partial}(n, d, s^{(i)}) + 2^{n-i} \cdot B_{Eval_Sum_Avg}(n - s^{(i)}, d)] \\
&= B_{Partial}(n, d, s^{(0)}) + 2^n B_{Eval_Sum_Avg}(n - s^{(0)}, d) + \\
&\quad \textstyle\sum_{i=1}^{m-1} [B_{Partial}(n, d, s^{(i)}) + 2^{n-i} B_{Eval_Sum_Avg}(n - s^{(i)}, d)] \\
&\doteq d \cdot 2^n + \textstyle\sum_{i=1}^{m-1} [2^{n-i+1} \sum_{j=0}^{d} \binom{i-1}{j} \cdot (d-j) + 2^{n-i+1} \sum_{j=0}^{d} \binom{i-1}{j} 2^{-j-1}] \\
&\leq d \cdot 2^n + \textstyle\sum_{i=1}^{\infty} [2^{n-i+1} \sum_{j=0}^{d} \binom{i-1}{j} \cdot (d-j) + 2^{n-i+1} \sum_{j=0}^{d} \binom{i-1}{j} 2^{-j-1}] \\
&= d \cdot 2^n + \textstyle\sum_{i=0}^{\infty} [2^{n-i} \sum_{j=0}^{d} \binom{i}{j} \cdot (d-j) + 2^{n-i} \sum_{j=1}^{d} \binom{i}{j} 2^{-j-1}] \\
&= d \cdot 2^n + 2^n [\textstyle\sum_{j=0}^{d} (d-j) \sum_{i=0}^{\infty} \binom{i}{j} 2^{-i} + \sum_{j=0}^{d} 2^{-j-1} \sum_{i=0}^{\infty} \binom{i}{j} 2^{-i}] \\
&= d \cdot 2^n + 2^n [\textstyle\sum_{j=0}^{d} (d-j) \cdot 2 + \sum_{j=0}^{d} 2^{-j-1} \cdot 2] \\
&\leq d \cdot 2^n + 2^n [\textstyle\sum_{j=0}^{d} (d-j) \cdot 2 + 2] \\
&= (d^2 + 2d + 2) 2^n.
\end{aligned}$$

We note that this upper bound may not be precise since the cost of initialization in GGCEs (in partial evaluation) is ignored from the line starting with "$\doteqdot$." However, when $32 \leq m = n \leq 64$ and $2 \leq d \leq 4$, we find that this latter cost is indeed negligible in theory. Also remember that we assume $|V^{(i)}| = 2^{n-i}$, which might not be accurate for some cases.

In some sense, this scheme is a mix of GGCE (or partial evaluation) and naïve evaluation. By choosing $s^{(i)}$, we may determine the weights of the two methods in the computation of $f^{(i)}(V^{(i)})$. That is, when $s^{(i)}$ is close to n, the scheme highly resembles GGCE. On the other hand, when $s^{(i)}$ is close to 0, the scheme is more like the naïve evaluation. Actually, this viewpoint is consistent with our experiment result: GGCE is more suitable for computation of $f^{(i)}(V^{(i)})$ when $|V^{(i)}|$ is close to 2^n (or equivalently, when i is small), and vice versa.

The importance of this scheme resides in its flexibility. This scheme not only contains the solvers described in Section 2.3, 2.5, and 3.1.1, but also allows time-memory trade-off. Furthermore, it can be easily adapted according to the implementation hardware platform.

Note that using different $s^{(i)}$ for each equation might not be suitable for general hardware platform such as GPUs and CPUs, for they lack the capability of efficient handling of bit vectors of a wide variety of widths. For special devices such as FPGAs, however, the scheme might work well.

3.2 Gaussian Elimination

In the previous subsection, we have shown that running GGCE on $\mu < m$ equations can be useful. In this section, we will show that the well-known Gaussian elimination can make our solver even faster.

Note that in GGCE, there are some constant data, namely the $\mathbf{C}_{\beta_1,\ldots,\beta_d}$'s (or $\delta_{\beta_1,\ldots,\beta_d}$'s). According to Fig. 2.2(b), actions involving them are always in the form

$$\delta_{\beta_1,\ldots,\beta_{d-1}} += \mathbf{C}_{\beta_1,\ldots,\beta_d}.$$

The key point is that if $\mathbf{C}_{\beta_1,\ldots,\beta_d} = 0$, all such actions can be simply omitted.

Now suppose GGCE is used to solve the first μ equations $f^{(0)}$ to $f^{(\mu-1)}$. The target system can be treated as a matrix, where each row corresponds to one equation, and each column corresponds to one term containing the same monomial. Apparently, the solution space is invariant under elementary row operations. Thus, we may eliminate some $\mathbf{C}^{(i)}_{\beta_1,\ldots,\beta_d}$'s for all $0 \leq i \leq \mu-1$. In fact, we can always eliminate $m - \mu$ such coefficients. In Section 4.1, we will discuss the probability of accessing each differential, and apparently we should eliminate the most frequently used $m-\mu$ $\mathbf{C}^{(i)}_{\beta_1,\ldots,\beta_d}$'s for all $0 \leq i \leq \mu - 1$ to achieve the greatest saving.

Chapter 4

Implementations

4.1 On NVIDIA GPUs, with CUDA

4.1.1 Overview

Our implementations allow parallel use of multiple GPU devices. Thus, at the beginning, the input system should be partially evaluated into intermediary systems, whose number equals to the number of devices. Then the same number of processes would be invoked such that each of them solves one of the intermediary systems by launching a kernel on one device.

Each of these processes would solve the first 32 equations with GGCE (enumeration phase), which would take place on the device. The solutions found by GGCE would be checked against the remaining equations using naïve evaluation (check phase) on CPU. We pick the number 32 to match the register width on GPUs. In this way, each 32-bit differential can be store in a register, and accumulating can be done by performing bitwise XOR on them.

Before the enumeration phase starts, some preperation must be done first. Since a GPU kernel usually requires enough threads to hide instruction latency, the first 32 equations of each intermediary system has to be partially evaluated into a large number of small systems, each to be solve by one GPU thread with GGCE. Then, the non-common parts ($\mathbf{C}_{\beta_1,\ldots,\beta_k}$'s where $k < d$) of the small systems would be sent to

global memory, while the common parts ($\mathbf{C}_{\beta_1,\ldots,\beta_d}$'s) are stored in constant memory.

After that, the enumeration phase can take the coefficients as input and run the instances of GGCE concurrently. The details and important issues in this phase will be introduced in the following subsections. By the end of this phase, the solution found by each thread would be stored in global memory, so they can be moved back to CPU for further processing.

The check phase is straightforward compared to the previous phase. Since only few (compared to 2^n) solutions are expected to entering this phase, there are no fancy techniques involved. The only notable issue in this phase is that it actually handles some "mending" work, which will also be discussed in Section 4.1.5.

4.1.2 Register Usage

Because of the scarcity of fast memory on GPU, register usage is usually a critical issue for CUDA programmers. The problem, unfortunately, seems inevitable since the number of differentials grows rapidly as d increases. Actually, the problem can be even worse since NVIDIA's `nvcc` compiler tends to allocate more registers than necessary. In fact, in our implementation for quadratic systems, everything fits in the registers after initialization. On the contrary, this is not the case in implementations for cubics and quartics.

In our implementation for quartics, each thread needs to maintain $\delta_{i,j,k}$ for $0 \leq i < j < k < K$, where K is the number of variables in the small systems. For $K = 10$, $\delta_{i,j,k}$'s take 120 registers if we just store all of them in registers, making the number of active warps in each MP no more than 4, not to mention others differentials. One may argue that this problem can be solved by restricting the number K. However, this implies that an extremely deep partial evaluation has to be carried out, which can be both time and memory consuming.

Our strategy for register usage follows the principle of caching—storing the most frequently used things in registers. To be precise, each differential δ_{*k} is accessed with probability $2^{-(k+1)}$ in each attempt. In other words, there exists a strong

bias in the probabilities of accessing each differential. So we can pick a suitable number γ and store all the δ_{*k} with $k \leq \gamma$ in registers and other differentials, global memory. The number of variables and actual number of registers allocated in our GPU implementations are shown in Table 4.1.

Table 4.1: Number of Registers Allocated in GPU Implementations

$d = 3$ ($\gamma = 9$)			$d = 4$ ($\gamma = 7$)		
diffs	other	actual	diffs	other	actual
46	11	64	64	15	80

4.1.3 Unrolling

While accumulating takes only few XORs (of differentials), indexing causes considerable overhead in each attempt. However, we find that the ubiquitous trick—unrolling—can be quite helpful to alleviate the overhead.

Let us take a look at Fig. 4.1, which exemplifies our unrolling scheme for a GGCE solving quartics with unroll factor 8. The indices listed are all in the same unrolled block. It is shown that some entries, such as $b_2(*\cdots*011)$, are constant. Other entries, although not fixed, can be determined once all $b_i(*\cdots*000)$'s are known. Thus, the indexing is no longer needed in every attempt. Instead, it only needs to be invoked in attempts with index being an integral multiple of 8. This example also illustrates the cases for quadratics and cubics.

The reason for this is not really complex. Consider any unrolled block with 2^u indices, where the first index is i. Any of other indices in this unrolled block can be defined as $i' = i + k$, where $k < 2^u$. Thus, the indices of the least significant HammingWeight(k) nonzero bits in i' must be constant and can be determined before runtime. If we need any more indices $b_j(i')$ for i' (which means HammingWeight(k) $< d$), it can always be computed by $b_j(i') = b_{j-h}(i)$, where $h =$ HammingWeight(k) and $j > h$.

Figure 4.1: An Example Unrolled Block with 8 Indices

index	b_4	b_3	b_2	b_1
$*\cdots*000$	β_4	β_3	β_2	β_1
$*\cdots*001$	β_3	β_2	β_1	0
$*\cdots*010$	β_3	β_2	β_1	1
$*\cdots*011$	β_2	β_1	1	0
$*\cdots*100$	β_3	β_2	β_1	2
$*\cdots*101$	β_2	β_1	2	0
$*\cdots*110$	β_2	β_1	2	1
$*\cdots*111$	β_1	2	1	0

4.1.4 Testing with Conditional Move

In Fig. 2.2(b), the testing simply adds the candidate vector into a set if its image is zero. However, this is infeasible in GPU implementations, for device memory is limited. Even if we assume the memory is large enough to store all candidate vectors, this can induce other problems such as synchronization between threads. Actually, the testing in our GPU implementation is delicately designed to fit the hardware platform.

In the process of implementation, we discovered an undocumented feature of CUDA for G2xx series GPUs: `nvcc` reliably generates conditional (predicated) move instructions, which are dispatched with exceptional adeptness. According to our experiment results, we believed that conditional moves can be dispatched by SFUs (Special Function Units), so the they can be executed simultaneously with other instructions (such as XORs) handled by SPs (Streaming Processors).

In order to exploit the feature, the testing in GPU implementations is somewhat different with the pseudocode. Each thread maintains two registers, *count* and *sol*, to keep track of the solution count and the last solution found during runtime. Note that *count* does not actually record solution count accurately. Instead, it contains only three states: 0, 1, and 2+ (greater or equal to 2 solutions). While the two registers can be easily initialized, they should be correctly maintained after each unrolled block, which can be achieved by maintaining the same but local data for each unrolled block.

For each unrolled block, we use a tiny queue Q with capacity of merely two

elements to maintain the local data. Whenever a solution is found, the corresponding test vector (actually a part of it) is enqueued. In this way, we can tell whether the solution count is 0, 1, or 2+ by checking whether there are 0, 1, or 2 elements in Q at the end of each unrolled block. Moreover, the last solution (if any) must lies in the back of Q.

In order to show the power of this technique, some actual CUDA codes are presented in Table 4.2(b). After applying `decuda` to our program, we found that the repetitive four-line code segments correspond to at least four instructions including two XORs and two conditional moves. However, according to our experiment result, the four instructions average less than three SP cycles, which means executions of XORs and conditional moves are much overlapped.

Figure 4.2: CUDA and Cubin Code Fragments of Degree-2 GPU Implementation

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

(a) `decuda` Result from Cubin

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

(b) CUDA Code for an Inner Loop Fragment

4.1.5 Re-enumeration

In the check phase, the solutions found in enumeration phase are examined. If a thread has found more than one solution, some mending work must be done on CPU or we may miss some actual solutions of the target system. A remedy for this is to repeat the work (GGCE) done by the thread again, which we call "re-enumeration." In fact, the re-enumeration can be aborted once the candidate solution meet the last solution returned by the thread. However, this is still a situation we would like to avoid as much as possible. Thus, our solution is to reduce the probability that any thread has found more than one solution by restricting the number of variables in

the small systems. The smaller the solution space is, the little the probability would be. In fact, when $n = 48$, we restrict the number of variables (in small systems) to 26 to make re-enumerations take negligible time.

4.2 On x86-64 CPUs, with SSE2 Intrinsics

4.2.1 Overview

At the beginning, the target system would be partially evaluated into several (this number usually equals to that of available cores) intermediary systems, each to be solved by one process. The processes would be properly assigned to CPU cores such that each core deals with (almost) equal amount of processes.

Each process would solve the first 16 equations of the intermediary system by GGCE. Candidate solutions found by GGCE would be checked for the next 16 equations using naïve evaluation with the early-abort strategy, which we call "filtering," and those passing the first 32 equations would be checked against the remaining equations using naïve evaluation (without early abort).

4.2.2 Batched Enumeration

In our GPU implementations, GGCE is implemented using a bit-slicing strategy, such that 32-bit differentials are stored in 32-bit registers. However, for CPU implementations, we would like to take advantage of the 128-bit XMM registers, while differentials are only 16-bit. Thus, we run eight instances of GGCE at the same time. Each differential is of the type `__int128`, where each 16-bit block of it correspond to one of the eight instances. In this way, accumulating can be done by manipulating (XORing) the 128-bit variables directly. Note that we need to partially evaluate the first 16 equations to generate at least eight sets of 16 equations.

After accumulating, the testing should be able to tell if there is any 16-bit all-zero block in δ. Actually this can be achieved by using a few lines of SSE2 intrinsics as presented in Fig. 4.3. The intrinsic `_mm_cmpeq_epi16` performs (16-bit) block-wise

comparison between `res` (stands for δ) and `zero`, an all-zero 128-bit variable. A 16-bit-block of `Mask` will then be set to `0xFFFF` if equivalence is found in the corresponding blocks, and `0x0000` otherwise. After that, `_mm_movemask_epi8` converts the 128-bit `Mask` into the 16-bit `mask` by extracting the most significant bits of the 8-bit blocks in `Mask`.

After executions of the two intrinsics, it is clear that `mask` would be nonzero if and only if there is any 16-bit all-zero block in `res`. When this happens, a routine `check` would be invoked to handle the following jobs. Note that this branch-away scheme is completely different with that in GPU implementations.

Figure 4.3: Code Fragments of Testing in CPU Implementations

```
Mask = _mm_cmpeq_epi16(res, zero);
mask = _mm_movemask_epi8(Mask);
if(mask) check(mask, idx, x^2);
```

4.2.3 Batched Filtering

Theoretically, we can evaluate $f^{(i)}(\mathbf{v})$ once the candidate vector $\mathbf{v}$ has passed $f^{(i-1)}$. However, this involves only single-bit operations (XORs and ANDs), which is not cost-effective in CPUs that support 128-bit wide operations. Thus, our solution to this is to maintain a buffer that can store up to 128 candidate vectors for each $f^{(i)}$. Once the buffer is full, we would rearrange 128 inputs of n bits such that they appear as n `__int128`'s, then evaluate one polynomial for 128 results in parallel using 128-bit wide ANDs and XORs.

Note that we use multivariate Horner's rule to achieve this batched filtering, for the actions it takes are the same for any vector in $\mathbb{F}_2^n$. Also note that the equations for filtering are also partially evaluated to reduce the cost for evaluation. However, substituting too many variables in partial evaluation can make it difficult to saturate most buffers, so this should be carefully tuned to achieve the best performance.

Chapter 5

Experiment Results

Table 5.1: Performance Results for $n = 48$

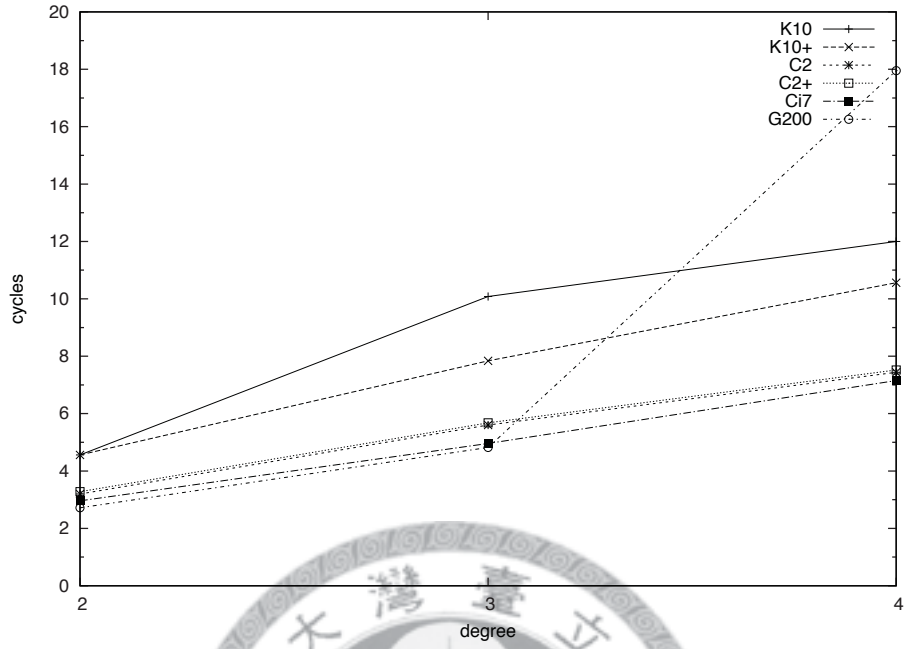
Minutes			Testing platform				#cores	#threads
$d = 2$	$d = 3$	$d = 4$	GHz	Arch.	Name	USD	available	launched
1217	2686	3191	2.2	K10	Phenom 9550	120	4	1
1157	1992	2685	2.3	K10+	Opteron2376	184	4	1
142	240	336	2.3	K10+	Opteron2376×2	368	8	8
780	1364	1819	2.4	C2	Xeon X3220	210	4	1
671	1176	1560	2.83	C2+	Core2 Q9550	225	4	1
179	294	390	2.83	C2+	Core2 Q9550	225	4	4
761	1279	1856	2.26	Ci7	Xeon E5520	385	4	1
139	213	327	2.26	Ci7	Xeon E5520×2	770	8	8
95	154	225	2.26	Ci7	Xeon E5520×2	770	8	16
41	73	271	1.3	G200	GTX 280	n/a	240	n/a
21	36	126	1.25	G200	GTX 295	500	480	n/a

Table 5.2: Efficiency Comparison: Cycles Per Candidate Tested on One Core

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

Architecture and Differences. In Table 5.1 and Table 5.2, we show our test results with a variety of machines and graphics cards. It can be concluded from the tables that cycles per attempt (candidate vector) is almost always a constant depending on the testing platform. In other words, we can easily estimate the running time given the architecture, frequency, number of cores, and n. Our implementations are scalable w.r.t. n, which has been explained by the discussion in Section 4.1.2.

Figure 5.1: Cycles Per Candidate Tested for Polynomials of Degree 2, 3, and 4



The marked cycle count difference between Intel and AMD cores is explained by Intel dispatching *three* XMM (SSE2) logical instructions to AMD's *two* per cycle and handling branch prediction and caching better.

Trends As Degree d Increases. Fig. 5.1 shows the variance in cycle count taken by some fix amount of candidate vectors (which is eight vectors per core for CPUs and one vector per SP for GPUs) as d increases. For most of the architectures, the cycle count increases almost linearly, which is consistent with theoretical complexity of GGCE. However, there are two exceptions. The burst in cycle count when $d = 4$ on G200 is apparently due to fast memory (register) pressure, while the anomaly on K10 is believed to be caused by insufficiency in cache size.

Gaussian Elimination. On GPUs, with $m - \mu = 32$ coefficients eliminated, we have a speed up of 21% on quadratic cases, 18% for cubics, and 4% for quadratics. On CPUs, with $m - \mu = 48$ coefficients eliminated, we have 16% on quadratic cases, 20% for cubics, and 9% for quadratics. Although there is still some room for improvement, we have shown that this technique can bring considerable improvement

in speed.

Bibliography

[1] B.-Y. Yang and J.-M. Chen All in the XL Family: Theory and Practice. Proc. 7th International Conference on Information Security and Cryptology, volume 3506, Lecture Notes in Computer Science, pages 67-86, 2004.

[2] Gregory V. Bard *Algebraic Cryptanalysis*. Springer-Verlag, New York, first edition, 2009.

[3] N. T. Courtois and Gregory V. Bard Algebraic Cryptanalysis of the Data Encryption Standard. Available at `http://eprint.iacr.org/2006/402/`.

[4] G. V. Bard, N. T. Courtois, and C. Jefferson. Efficient methods for conversion and solution of sparse systems of low-degree multivariate polynomials over GF(2) via SAT-solvers. `http://eprint.iacr.org/2007/024`.

[5] M. Bardet, J.-C. Faugère, and B. Salvy. On the complexity of Gröbner basis computation of semi-regular overdetermined algebraic equations. In *Proc. Int'l Conference on Polynomial System Solving*, pp. 71–74, 2004. INRIA report RR-5049.

[6] M. Bardet, J.-C. Faugère, B. Salvy, and B.-Y. Yang. Asymptotic expansion of the degree of regularity for semi-regular systems of equations. *Proc. MEGA 2005*, 2005.

[7] C. Berbain, H. Gilbert, and J. Patarin. QUAD: A practical stream cipher with provable security. Eurocrypt 2006, LNCS 4004, pp. 109–128.

[8] D. J. Bernstein, T.-R. Chen, C.-M. Cheng, T. Lange, and B.-Y. Yang. ECM on graphics cards. Eurocrypt 2009, LNCS 5479, pp. 483–501.

[9] Luk Bettale, Jean-Charles Faugère and Ludovic Perret. *Hybrid approach for solving multivariate systems over finite fields*, J. Math. Crypto. **3:3**(2009) pp. 177–197.

[10] C. Bouillaguet, J.-C. Faugère, P.-A. Fouque, and L. Perret. Differential-algebraic algorithms for the isomorphism of polynomials problem. `http://eprint.iacr.org/2009/583`

[11] C. Bouillaguet, P.-A. Fouque, A. Joux, and J. Treger. A family of weak keys in HFE (and the corresponding practical key-recovery). `http://eprint.iacr.org/2009/619`.

[12] B. Buchberger. *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal.* PhD thesis, Innsbruck, 1965.

[13] Johannes Buchmann, Daniel Cabarcas, Jintai Ding and Mohamed Saied Emam Mohamed. *Flexible Partial Enlargement to Accelerate Gröbner Basis Computation over* $\mathbb{F}_{\not 2}$, Africacrypt 2010, LNCS 6055, pp. 69–81.

[14] N. Courtois, G. V. Bard, and D. Wagner. Algebraic and slide attacks on Keeloq. FSE 2008, LNCS 5086, pp. 97–115.

[15] N. Courtois, L. Goubin, and J. Patarin. *SFLASH: Primitive specification (second revised version)*, 2002. `https://www.cosic.esat.kuleuven.be/nessie`

[16] N. T. Courtois, A. Klimov, J. Patarin, and A. Shamir. Efficient algorithms for solving overdefined systems of multivariate polynomial equations. Eurocrypt 2000, LNCS 1807, pp. 392–407. Extended ver.: `http://www.minrank.org/xlfull.pdf`.

[17] N. de Bruijn. *Asymptotic methods in analysis. 2nd edition.* Bibliotheca Mathematica. Vol. 4. Groningen: P. Noordhoff Ltd. XII, 200 p. , 1961.

[18] J.-C. Faugère. A new efficient algorithm for computing Gröbner bases (F_4). *J. of Pure and Applied Algebra*, **139**(1999), pp. 61–88.

[19] J.-C. Faugère. A new efficient algorithm for computing Gröbner bases without reduction to zero (F_5). ACM ISSAC 2002, pp. 75–83.

[20] J.-C. Faugère and A. Joux. Algebraic cryptanalysis of Hidden Field Equations (HFE) using Gröbner bases. CRYPTO 2003, LNCS 2729, pp. 44–60.

[21] A. Fog. *Instruction Tables*. Copenhagen University, College of Engineering, Feb 2010. Lists of Instruction Latencies, Throughputs and micro-operation breakdowns for Intel, AMD, and VIA CPUs, `http://www.agner.org/optimize/instruction_tables.pdf`.

[22] J. Patarin. Asymmetric cryptography with a hidden monomial. Crypto 1996, LNCS 1109, pp. 45–60.

[23] J. Patarin. Hidden Field Equations (HFE) and Isomorphisms of Polynomials (IP): two new families of asymmetric algorithms. Eurocrypt 1996, LNCS 1070, pp. 33–48. Extended ver.: `http://www.minrank.org/hfe.pdf`.

[24] J. Patarin, N. Courtois, and L. Goubin. QUARTZ, 128-bit long digital signatures http://www.minrank.org/quartz/. CT-RSA 2001, LNCS 2020, pp. 282–297.

[25] J. Patarin, L. Goubin, and N. Courtois. Improved algorithms for Isomorphisms of Polynomials. Eurocrypt 1998, LNCS 1403, pp. 184–200. Extended ver.: `http://www.minrank.org/ip6long.ps`.

[26] H. Raddum. MRHS equation systems. SAC 2007, LNCS 4876, pp. 232–245.

[27] M. Sugita, M. Kawazoe, L. Perret, and H. Imai. Algebraic cryptanalysis of 58-round SHA-1. FSE 2007, LNCS 4593, pp. 349–365.

[28] B.-Y. Yang and J.-M. Chen. Theoretical analysis of XL over small fields. ACISP 2004, LNCS 3108, pp. 277–288.

[29] B.-Y. Yang, J.-M. Chen, and N. Courtois, *On Asymptotic Security Estimates in XL and Gröbner Bases-Related Algebraic Cryptanalysis*, ICICS 2004, LNCS 3269, pp. 401-413.