國立臺灣大學電機資訊學院電子工程學研究所

碩士論文

Graduate Institute of Electronics Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Master Thesis

解佈於三元體之多項式方程組之快速窮舉法

Fast Exhaustive Search for Polynomial Systems over $\mathbb{F}_3$

王偉政

Wei-Jeng Wang

指導教授：鄭振牟 博士

Advisor: Chen-Mou Cheng, Ph.D.

中華民國 105 年 7 月

July, 2016

摘要

在密碼學的領域裡，如何在有限體裡面解多變量方程組是一個很重要的問題。我們已經知道在$\mathbb{F}_2$裡且低維度的系統下，當變數與方程式的數量一樣多時，列舉演算法比現存的任何演算法都還要有效率，當然，我們只考慮實際可能的問題大小。然而我們卻不知道對於其他有限體是否也有一樣的結果。

因此，在此研究中我們對於在$\mathbb{F}_3$裡維度較低的系統提出了適合平行化的窮舉搜尋演算法，這個演算法不僅可以透過SSE2指令集使的在CPU達到平行化的效果，也可以經由CUDA實作在顯示晶片上。它的優化技巧以及與$\mathbb{F}_2$演算法的差異我們也會詳加分析。

我們的實作在使用了nVidia顯示卡時，解維度2、變數30且方程式30+的多變量系統問題只需要14分鐘；此外，解維度3的系統可以在36分鐘內完成。這些表現也都超越了所有現存的演算法，同時根據這些結果我們可以進一步的比較隨著有限體的大小增加，Gröbner基底與列舉演算法之間的關係。

關鍵字: 多變量方程式、代數分析、窮舉搜尋、平行化、顯示晶片。

Abstract

Solving multivariate polynomial systems over finite fields is an important problem in cryptography. For random $\mathbb{F}_2$ low-degree systems with equally many variables and equations, enumeration is more efficient than advanced solvers for all practical problem sizes. Whether there are others remained an open problem.

We here study and propose an exhaustive-search algorithm for low-degree systems over $\mathbb{F}_3$ which is suitable for parallelization. We implemented it on Graphic Processing Units (GPUs) and commodity CPUs. Its optimizations and differences from the $\mathbb{F}_2$ case are also analyzed.

We can solve 30+ quadratic equations in 30 variables on an NVIDIA GeForce GTX 980 Ti in 14 minutes; a cubic system takes 36 minutes. This well outperforms existing solvers. Using these results, we compare Gröbner Bases vs. enumeration for polynomial systems over small fields as the sizes go up.

Keywords: *multivariate polynomial, algebraic cryptanalysis, exhaustive search, parallelization, Graphic Processing Units (GPUs)*

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

If one can solve large systems of polynomial equations, one can break all cryptosystems. This general approach is often called algebraic cryptanalysis [9]. Unfortunately, solving such systems is not easy. Indeed, not only is this an NP-complete problem [14], the following problem is conjectured to be probabilistically hard [3]:

Problem $\mathcal{M}Q(q; n, m)$: Solve $p_1(\mathbf{x}) = p_2(\mathbf{x}) = \cdots = p_m(\mathbf{x}) = 0$, where each p_i is a quadratic in $\mathbf{x} = (x_1, \ldots, x_n)$. All coefficients and variables are in $\mathbb{F}_q$.

To be exact, the QUAD stream cipher [3] can be proved secure for certain parameters under the assumption that: "If we randomly generate the $n^2(n+1)$ coefficients in a set of $2n$ quadratic equations in n-bit variables to generate instances in $\mathcal{M}Q(2; n, 2n)$, the probability for any algorithm $\mathcal{A}$ to terminate within time $\mathrm{poly}(n)$ with a solution would be less than any given fixed $\epsilon > 0$ as $n \to \infty$." To date, no one has seriously challenged this statement. Multivariate Public-Key Cryptosystems (MPKCs) [10][17][18], where the public map is a multivariate quadratic map. also require $\mathcal{M}Q$ to be hard. However, MPKCs have built-in trapdoors, so many effective known attacks are structural attacks solving the instance of extended Isomorphism of Polynomials. Of course, in practice $\mathcal{M}Q$ complexity still needs to be evaluated for every

MPKC, as an upper bound of security.

Since Buchberger [8], Gröbner basis techniques have been the most prominent tool for solving systems of equations. For over a decade, the standard benchmark of cryptographic system-solving has been the Gröbner basis algorithm $\mathbf{F}_4$[11], more precisely the variant that is commercially available in the computer algebra MAGMA [16]. A more advanced (but *not* publicly available) algorithm[12] $\mathbf{F}_5$ was the first to break the first HFE challenge in 2002 [13]. The properties of algebraic solvers such as $\mathbf{F}_4$ and XL has been studied in detail [20][1][2]

In 2010, Bouillaguet *et al* showed that exhaustive search algorithms can be made extremely efficient, practically faster than existing techniques for solving generic systems over $\mathbb{F}_2$ using commodity computers and graphics cards and even reconfigurable computing [7]. Especially for random systems over $\mathbb{F}_2$, it seems as if enumeration represents the best solution for most cases of cryptographical interest [21]. An open source software library [5] is available for use with SAGE. The leaders board of the Fukuoka MQ Challenge series I and IV (dealing with $\mathbb{F}_2$ systems) are dominated by enumerative solutions.

Since 2010, there seemed to have been folklore among cryptographers that similar results might hold for enumerative solutions of systems over $\mathbb{F}_3$ and possibly even larger fields, just as it does over $\mathbb{F}_2$. However, there is no publication on record to that effect.

1.2 Our Contribution

We provide a comparative study of enumerative solutions vs. Gröbner basis methods in small fields other than $\mathbb{F}_2$. Of course, $\mathbb{F}_3$ is very ill-suited for computers as it takes two bits to represent a ternary digit ("trit") but this is a handicap both for Gröbner basis methods and for enumeration. We can restate the end of the above section as the following open question:

> Do enumerative methods hold a similar advantage over Gröbner basis techniques for other small fields and how well does that enumeration do in practice for $\mathbb{F}_3$ (and $\mathbb{F}_4$, $\mathbb{F}_5 \ldots$)?

Our answer is that Brute Force or Enumeration can achieve nearly as much for $\mathbb{F}_3$ as it did for $\mathbb{F}_2$, although the set-up phase and book-keeping issues are messier. For quadratic systems, each test vector only takes on average two parallelized additions in $\mathbb{F}_3$ (which is the same as $\mathbb{F}_2$). The enumerative approach is also extensible to higher degrees for $\mathbb{F}_3$ just as for $\mathbb{F}_2$, although when enumerating for a degree-d system, each test vector would take more than d adds in $\mathbb{F}_3$.

Our algorithm has been implemented with several optimizations on CPU and GPU using SSE2 intrinsics and the CUDA framework respectively. Although there is still room for improvement (e.g., no provision to use multiple GPUs simultaneously), it outruns all existing Gröbner solvers to which we have access.

Today, we can solve 30+ quadratic equations in 30 variables with one NVIDIA GeForce GTX 980 Ti graphics card in 14 minutes. A cubic system under the otherwise the same conditions takes 26 minutes. Using MAGMA-2.21-9 on a 4-GHz core of the AMD FX-8350, F$\mathbf{F}_4$ (i.e., guess an optimal number of variables before running the $\mathbf{F}_4$ solver, the *hybrid approach* [20][4]) would take 150 core-days to solve 30 quadratic equations in as many $\mathbb{F}_3$-variables. This is the best Gröbner basis solvers commercially available today.

By the way, although we only illustrate all our methods and the problem with $\mathbb{F}_3$, the formulas and the experiment results about $\mathbb{F}_5$ are shown in the paper as well. However we will not introduce this part too much since the main idea is similar to that in $\mathbb{F}_3$.

Chapter 2

Preliminaries

2.1 Notational Conventions

In this paper, we enumerate over the finite-dimensional vector space $(\mathbb{F}_3)^n$, and use the base-3 or ternary numerals. Analogous to a bit, a ternary digit is a **trit**.

We use $\mathbf{C}_{\beta_1,\beta_2,\ldots,\beta_k}$ to stand for the coefficient of the monomial $x_{\beta_1}x_{\beta_2}\cdots x_{\beta_k}$ of a polynomial f, and use $\mathbf{C}$ for the constant term. Because we know that any x_β^α where $\alpha \geqslant 1$ can be reduced to x_β^γ where $\gamma \in \{1, 2\}$, $\gamma = \alpha \bmod 2$, so the restrictions on the indices β_i can be formulated as follows for $1 \leqslant i \leqslant k-2$:

1. $0 \leqslant \beta_1 \leqslant \beta_2 \leqslant \cdots \leqslant \beta_k < n$, and
2. $\beta_{i+1} \neq \beta_{i+2}$ if $\beta_i = \beta_{i+1}$.

In addition, we use $\boxplus$ to denote trit-wise addition of vectors in $\mathbb{F}_3^n$, which means that each corresponding pairs of trits is added together $\pmod 3$ without carry. In a similar way, trit-wise subtraction is denoted by $\boxminus$. We also use $\ggg$ (resp. $\lll$) to denote ternary right-shift (resp. left-shift) operation. Since $3 = 0$, we also have $2 = -1$ and occasionally subtracting the same variable is achieved by adding the same variable twice.

$\mathbb{F}_5$ Version

Now consider the finite-dimensional vector space $(\mathbb{F}_5)^n$. It is a numeral system with five as the base. We use the base-5 or quinary numerals to perform a quinary system. For a variable, the highest degree is four in $\mathbb{F}_5$, so any x_β^α where $\alpha \geqslant 1$ can be reduced to x_β^γ where $\gamma \in \{1, 2, 3, 4\}$, $\gamma = \alpha \bmod 4$. The restrictions are also modified as follows $(1 \leqslant i \leqslant k - 4)$:

1. $0 \leqslant \beta_1 \leqslant \beta_2 \leqslant \cdots \leqslant \beta_k < n$, and
2. $\beta_{i+3} \neq \beta_{i+4}$ if $\beta_i = \beta_{i+3}$.

2.2 Representation used for Ternary Arithmetic

Each trit must be represented by 2 bits. In our implementation, we represent 0 as 00, 1 as 10 and 2 as 11. This representation has the advantage that it is easy to check whether a trit is equal to zero just by checking its most significant bit (MSB). Suppose we have elements $x, y, z \in \mathbb{F}_3$ with their 2-bit representations being bits (x_0, x_1), (y_0, y_1), (z_0, z_1), where the MSB is indexed as 1 and the LSB as 0. Formulas corresponding to basic operations in $\mathbb{F}_3$ are:

- $z = x + y \leftrightarrow z_1 = (x_1 \oplus y_1) \vee (x_0 \oplus y_1 \oplus y_0)$, $z_0 = (x_1 \oplus y_0) \wedge (x_0 \oplus y_1)$.
- $z = xy \leftrightarrow z_1 = (x_1 \wedge y_1)$, $z_0 = (x_1 \wedge y_0) \vee (x_0 \wedge y_1)$.

$\mathbb{F}_5$ Version

Of course, a quinary digit, which we call a **quint**, is represented by 3 bits. We want to make less and less operations be required and keep the advantage which is mentioned above, hence we represent 0 as 000, 1 as 100, 2 as 101, 3 as 111 and 4 as 110. Formulas about addition and multiplication in $\mathbb{F}_5$ are:

- $z = x + y \leftrightarrow \begin{cases} z_2 = & (x_2 \oplus y_2) \vee (x_0 \oplus y_0) \vee (x_2 \oplus x_1 \oplus y_1). \\ z_1 = & \Big(\overline{((x_1 \wedge y_0) \vee (x_0 \wedge y_1) \vee (x_2 \oplus y_2))} \\ & \wedge \big((x_1 \wedge y_1) \vee x_0 \vee y_0)\Big) \vee \big((x_2 \oplus y_2) \wedge (x_1 \oplus y_1)\big). \\ z_0 = & \Big((x_2 \oplus x_1 \oplus y_1) \wedge \overline{((x_0 \wedge y_0) \vee (x_2 \oplus y_2))}\Big) \\ & \vee \big((x_2 \oplus y_2) \wedge (x_0 \oplus y_0)\big). \end{cases}$

- $z = xy \leftrightarrow \begin{cases} z_2 = & (x_2 \wedge y_2). \\ z_1 = & (x_2 \wedge y_2) \wedge \big((x_0 \wedge y_0) \oplus (x_1 \oplus y_1)\big). \\ z_0 = & (x_2 \wedge y_2) \wedge (x_0 \oplus y_0). \end{cases}$

Besides, we list other useful formulas in Table 7.1 as well (see Appendix). For example, multiplying a constant number 4 is one of those. If an action is to subtract a variable, we can add the same variable four times due to $4 = -1$ in $\mathbb{F}_5$. Further, a variable multiplied by 4 needs only one bit operation, thus the total number of bit operations the action needs is equal to addition′s number plus one.

2.3 Ternary Gray Code

A k-trit ternary Gray code, sometimes called a $(3, k)$-Gray Code[15] is a Hamiltonian path in $\mathbb{F}_3^k$, or a sequence of all 3^k possible k-trit sequence such that two successive values differ in only one trit. Ternary gray codes are not unique, but the example given in Wikipedia[19] seems as much of a standard as any other.

Definition 1 (Standard k-trit Ternary Gray Code)**.** *Express all integers in* $[0; 3^k - 1]$ *as k-trit ternary numerals, then*

$$\mathit{TERNARYGRAYCODE}(x) := x \boxminus (x \gg 1).$$

The is analogous to that of the standard Gray Code and may be in fact proved to be a valid ternary Gray Code in a similar manner[15]. Table 2.1 shows part of a standard 4-trit ternary Gray code along with their corresponding indices in ternary, and we also show the quinary version in Table 7.2 (see Appendix). For example, if $x = 012$, then $(x \ggg 1) = 001$, and therefore TERNARYGRAYCODE(x) $=$ $012 \boxminus 001 = 011$, which can be also found in Table 2.1. The b_i columns in that table, is the analogue of the "the i-th rightmost non-zero bit position" of the binary case. We can capture their meaning in the following definition. Let x be written in ternary, as an index of a ternary Gray code.

Definition 2 (Position of the i-th difference vector)**.** *The notation $b_1(x)$ is defined the index of the least significant nonzero trit of x as a ternary number, and -1 if $x = 0$. For $i > 1$ we can then define recursively $b_i(x) := b_{i-1}(x - 3^{b_1(x)})$.*

We can see as a corollary that if the Hamming weight of x, defined as the sum *as an integer* of all trits in the ternary expansion of x is equal to h, then $b_j(x) = -1$ for $h < j$.

Warning: One should note that in Definition 2, *when one of the trits in x is two, the corresponding index occurs twice in the b sequence.* Therefore, while $b_i(x)$ is the analogue of "the position of the i-th rightmost non-zero bit" in the binary case,, for our (ternary) case Definition 2 is *not* "the position of the i-th rightmost non-zero trit". If we want to think of b_i that way, we must split each trit further into its two-bit form and consider b_i as the i-th rightmost bit in that expansion. In the same example used above, $b_1(5) = 0, b_2(5) = 0, b_3(5) = 1$ and $b_4(5) = -1$ because $5_{10} = (012)_3$.

Lemma 1. *Let e_i be the unit vector in the i-th direction (3^i as an integer), then*

$$\mathit{TERNARYGRAYCODE}(x+1) = \mathit{TERNARYGRAYCODE}(x) \boxplus e_{b_1(x+1)}.$$

Definition 3 (Partial Derivative)**.** *Let f be a scalar- or vector-valued polynomials* *over $(\mathbb{F}_3)^n$. Then we define: $\frac{\partial f}{\partial x_i}(v) = f(v \boxplus e_i) - f(v)$. Thus for any vector v, we have:*

$$f(v \boxplus e_i) = f(v) + \frac{\partial f}{\partial x_i}(v) \; . \tag{2.1}$$

For our convenience, TERNARYGRAYCODE(v) is denoted by g_v in the following pages. So Lemma 1 can be re-written as

$$f(g_v) = f(g_{v-1}) + \frac{\partial f}{\partial x_{b_1(v)}}(g_{v-1}).$$

We will build on this result in our paper to construct a better exhaustive search algorithm.

Table 2.1: 4-trit Ternary Gray Code with Index and Enumeration Actions

index	code	b_1	b_2	b_3	actions (quadratic)	actions (cubic)
000	000	-1	-1	-1		
001	001	0	-1	-1	$\delta += \delta_0$	$\delta += \delta_0$
002	002	0	0	-1	$\delta += (\delta_0 += \delta_{0,0})$	$\delta += (\delta_0 += \delta_{0,0})$
010	012	1	-1	-1	$\delta += \delta_1$	$\delta += \delta_1$
011	010	0	1	-1	$\delta += (\delta_0 += (\delta_{0,0} + \delta_{0,1}))$	$\delta += (\delta_0 += (\delta_{0,0} + \delta_{0,1}))$
012	011	0	0	1	$\delta += (\delta_0 += \delta_{0,0})$	$\delta += (\delta_0 += (\delta_{0,0} += \delta_{0,0,1}))$
020	021	1	1	-1	$\delta += (\delta_1 += (- \delta_{0,1} + \delta_{1,1}))$	$\delta += (\delta_1 += (\delta_{0,1} + (\delta_{0,1} += \delta_{0,0,1}) + \delta_{1,1}))$
021	022	0	1	1	$\delta += (\delta_0 += (\delta_{0,0} + \delta_{0,1}))$	$\delta += (\delta_0 += (\delta_{0,0} + (\delta_{0,1} += (\delta_{0,0,1} + \delta_{0,1,1}))))$
022	020	0	0	1	$\delta += (\delta_0 += \delta_{0,0})$	$\delta += (\delta_0 += (\delta_{0,0} += \delta_{0,0,1}))$
100	120	2	-1	-1	$\delta += \delta_2$	$\delta += \delta_2$
101	121	0	2	-1	$\delta += (\delta_0 += (\delta_{0,0} + \delta_{0,2}))$	$\delta += (\delta_0 += (\delta_{0,0} + \delta_{0,2}))$
102	122	0	0	2	$\delta += (\delta_0 += \delta_{0,0})$	$\delta += (\delta_0 += (\delta_{0,0} += \delta_{0,0,2}))$
110	102	1	2	-1	$\delta += (\delta_1 += (\delta_{0,1} + \delta_{1,1} + \delta_{1,2}))$	$\delta += (\delta_1 += (\delta_{0,1} + (\delta_{1,1} += \delta_{0,1,1}) + \delta_{1,2}))$
111	100	0	1	2	$\delta += (\delta_0 += (\delta_{0,0} + \delta_{0,1}))$	$\delta += (\delta_0 += (\delta_{0,0} + (\delta_{0,1} += (\delta_{0,0,1} + \delta_{0,1,1} + \delta_{0,1,2}))))$
112	101	0	0	1	$\delta += (\delta_0 += \delta_{0,0})$	$\delta += (\delta_0 += (\delta_{0,0} += \delta_{0,0,1}))$
120	111	1	1	2	$\delta += (\delta_1 += (- \delta_{0,1} + \delta_{1,1}))$	$\delta += (\delta_1 += (\delta_{0,1} + (\delta_{0,1} += \delta_{0,0,1}) + (\delta_{1,1} += (- \delta_{0,1,1} + \delta_{1,1,2}))))$
121	112	0	1	1	$\delta += (\delta_0 += (\delta_{0,0} + \delta_{0,1}))$	$\delta += (\delta_0 += (\delta_{0,0} + (\delta_{0,1} += (\delta_{0,0,1} + \delta_{0,1,1}))))$
122	110	0	0	1	$\delta += (\delta_0 += \delta_{0,0})$	$\delta += (\delta_0 += (\delta_{0,0} += \delta_{0,0,1}))$
200	210	2	2	-1	$\delta += (\delta_2 += (- \delta_{1,2} + \delta_{2,2}))$	$\delta += (\delta_2 += ((\delta_{1,2} += \delta_{0,1,2}) + (\delta_{1,2} += \delta_{1,1,2}) + \delta_{2,2}))$
201	211	0	2	2	$\delta += (\delta_0 += (\delta_{0,0} + \delta_{0,2}))$	$\delta += (\delta_0 += (\delta_{0,0} + (\delta_{0,2} += (- \delta_{0,1,2} + \delta_{0,2,2}))))$
202	212	0	0	2	$\delta += (\delta_0 += \delta_{0,0})$	$\delta += (\delta_0 += (\delta_{0,0} += \delta_{0,0,2}))$
210	222	1	2	2	$\delta += (\delta_1 += (\delta_{0,1} + \delta_{1,1} + \delta_{1,2}))$	$\delta += (\delta_1 += (\delta_{0,1} + (\delta_{1,1} += \delta_{0,1,1}) + (\delta_{1,2} += (- \delta_{0,1,2} + \delta_{1,1,2} + \delta_{1,2,2}))))$
211	220	0	1	2	$\delta += (\delta_0 += (\delta_{0,0} + \delta_{0,1}))$	$\delta += (\delta_0 += (\delta_{0,0} + (\delta_{0,1} += (\delta_{0,0,1} + \delta_{0,1,1} + \delta_{0,1,2}))))$
212	221	0	0	1	$\delta += (\delta_0 += \delta_{0,0})$	$\delta += (\delta_0 += (\delta_{0,0} += \delta_{0,0,1}))$
220	201	1	1	2	$\delta += (\delta_1 += (- \delta_{0,1} + \delta_{1,1}))$	$\delta += (\delta_1 += (\delta_{0,1} + (\delta_{0,1} += \delta_{0,0,1}) + (\delta_{1,1} += (- \delta_{0,1,1} + \delta_{1,1,2}))))$
221	202	0	1	1	$\delta += (\delta_0 += (\delta_{0,0} + \delta_{0,1}))$	$\delta += (\delta_0 += (\delta_{0,0} + (\delta_{0,1} += (\delta_{0,0,1} + \delta_{0,1,1}))))$
222	200	0	0	1	$\delta += (\delta_0 += \delta_{0,0})$	$\delta += (\delta_0 += (\delta_{0,0} += \delta_{0,0,1}))$

Chapter 3

Known Techniques for Enumerations

3.1 Naïve Evaluation

The simplest way to perform an enumeration algorithm is to evaluate the polynomial f over $(\mathbb{F}_3)^n$. For every integer $0 \leqslant i < 3^n$, we can form a vector of trits from its ternary expansion zero-padded to n trits, and term that v_i, the vector formed by integer i. So numeration means we check whether $f(v_i)$ is equal to zero or not for $i = 0, 1, \ldots, 3^n - 1$. The process of naïve evaluation is shown in Figure 3.1.

While we analyze the process of this algorithm, we can know that the advantage of this scheme is its storage used. It only needs to store the coefficients of the equations, and this only depends on the number of monomial terms. Now the problem is how to calculate it. We believe that the most complex thing is dealing with those terms which contain repeated variables. Luckily, we have known that a variable appears two times at most. Therefore, monomials are classified according to the number of pairs of repeated variables. We calculate those terms where every variable is different and then calculate the terms which contain only one pair of the same variable, and so on. In this way, the number of monomials can be obtained by the following formula (n is the number of variables and d represents the degree of a polynomial):

$$N_{Monomial}(n,d) = \sum_{i=0}^{d}\sum_{j=0}^{\lfloor \frac{i}{2} \rfloor}\binom{n}{j}\binom{n-j}{i-2j}\,. \tag{3.1}$$

In contrast, the significant disadvantage is that it needs a lot of time to evaluate $f(v_i)$. We can use the required number of bit operations to perform how long the algorithm needs. First, if there are m monomial terms, $(m-1)$ additions are required and every addition has six operations. Second, if there is a degree-d monomial where $d > 1$, $(d-1)$ multiplication are required and every multiplication has four operations. Thus the expected bit operations of each evaluation would be:

$$N_{Addition}(n,d) = \sum_{i=0}^{d}\sum_{j=0}^{\lfloor \frac{i}{2} \rfloor}\binom{n}{j}\binom{n-j}{i-2j} - 1 \tag{3.2}$$

$$N_{Multiplication}(n,d) = \sum_{i=1}^{d}\sum_{j=0}^{\lfloor \frac{i}{2} \rfloor}(i-1)\binom{n}{j}\binom{n-j}{i-2j} \tag{3.3}$$

$$\begin{aligned} N_{Bit\_Naïve}(n,d) &= 6N_{Addition}(n,d) + 4N_{Multiplication}(n,d) \\ &= \sum_{i=1}^{d}\sum_{j=0}^{\lfloor \frac{i}{2} \rfloor}(4i+2)\binom{n}{j}\binom{n-j}{i-2j}\,. \end{aligned} \tag{3.4}$$

In summary, when we know how long evaluating $f(v_i)$ needs, we can know that the whole enumeration will take $\left[3^n N_{Bit\_Naïve}(n,d)\right]$ bit operations as well. Further, if a polynomial system has m equations, the total number of operations is $\left[m3^n N_{Bit\_Naïve}(n,d)\right]$. In other words, it would require $O(m3^n dn^d)$ bit operations.

```
Sol ← ∅;
for $i = 0$ to $3^n - 1$ do
    $\delta \leftarrow f(\mathbf{v}_i)$;
    if $\delta = 0$ then
        $Sol \leftarrow Sol \cup \{\mathbf{v}_i\}$;
    end
end
return Sol;
```

Figure 3.1: Pseudocode of Naïve Evaluation

3.2 Basic Ternary Gray Code Enumeration

According to the lemma 1, we compute $f(g_i)$ by updating $f(g_{i-1})$ with their difference $\frac{\partial f}{\partial x_{b_1(v_i)}}(g_{i-1})$. Therefore, this indicates that searching the candidate vectors in the order of ternary Gray code requires less arithmetical operations than the requirement in naïve evaluation. The pseudocode is shown in Figure 3.2.

In the similar way, we can estimate its storage and required time of the algorithm. It is almost the same in the storage side except that there are two new variables. Since we always derive next result by updating the last one, the value of $f(v_i)$ is kept in δ, and β_1 is a necessary factor in computing differences.

However, this algorithm has better performance than naïve evaluation in the time side. The reason is that we calculate a degree-$(d-1)$ polynomial instead of degree-d. Because two successive ternary Gray code differ in only one trit, we can know that $f(g_i)$ and $f(g_{i-1})$ have different coefficients in monomial terms which contain the variable x_i, where $g_i = g_{i-1} \boxplus e_i$, and ignore the others; hence we can treat their difference as a new polynomial. We show an example to illustrate how to calculate it. For a 3 variables (x_0, x_1, x_2) and degree-2 polynomial, the standard representation would be:

$$\mathbf{C}_{0,0}x_0^2+\mathbf{C}_{1,1}x_1^2+\mathbf{C}_{2,2}x_2^2+\mathbf{C}_{0,1}x_0x_1+\mathbf{C}_{0,2}x_0x_2+\mathbf{C}_{1,2}x_1x_2+\mathbf{C}_0x_0+\mathbf{C}_1x_1+\mathbf{C}_2x_2+\mathbf{C}\ .$$

Suppose two ternary Gray code differ in x_1, thus we only concern about the terms $\mathbf{C}_{1,1}x_1^2$, $\mathbf{C}_{0,1}x_0x_1$, $\mathbf{C}_{1,2}x_1x_2$ and $\mathbf{C}_1x_1$ because all of the others are eliminated with their same coefficients. In order to explain the main idea easily, the order of difference is changed as below.

$$(\mathbf{C}_{0,1}x_0+\mathbf{C}_{1,2}x_2+\mathbf{C}_1)x_1+(\mathbf{C}_{1,1})x_1^2\ .$$

According to the exponential value, the monomials are divided into two groups.

Then we can treat the coefficient of x_1 as a 2 variables (x_0, x_2) and degree-1 polynomial and the coefficient of x_1^2 as a 2 variables (x_0, x_2) and degree-0 polynomial. Furthermore, updating δ needs one addition, so the expected bit operations of each evaluation would be:

$$N_{Bit\_BTGCE}(n,d) = N_{Bit\_Na\ddot{i}ve}(n-1,d-1) + N_{Bit\_Na\ddot{i}ve}(n-1,d-2) + 1\ . \quad (3.5)$$

We know the first term is usually much more than the second term, thus this implies that we can ignore those so that $N_{Bit\_BTGCE}(n,d) \approx N_{Bit\_Na\ddot{i}ve}(n-1,d-1)$. Consider a m equations polynomial system, the whole number of operations is $\left[m3^n N_{Bit\_Na\ddot{i}ve}(n-1,d-1)\right]$, and it can be represented as $O(m3^n dn^{d-1})$.

```
Sol ← ∅;
δ ← C;
for $i = 0$ to $3^n - 1$ do
    $\beta_1 \leftarrow b_1(i)$;
    if $\beta_1 \geqslant 0$ then
        $\delta \leftarrow \delta + \frac{\partial f}{\partial x_{\beta_1(v_i)}}(\mathbf{g}_{i-1})$;
    end
    if $\delta = 0$ then
        $Sol \leftarrow Sol \cup \{\mathbf{g}_i\}$;
    end
end
return $Sol$;
```

Figure 3.2: Pseudocode of Basic Ternary Gray Code Enumeration

3.3 Generalized Ternary Gray Code Enumeration

From the last section, we have known that while any two successive values differ in only one trit, the evaluations can be accelerated. In this section, we introduce a new algorithm, which we call generalized ternary Gray code enumeration (GTGCE). This method is the extension of section 3.2. It not only keeps the advantage as mentioned above but also makes use of the recursive technique.

First of all, let us consider a special situation in lemma 1. If $\frac{\partial f}{\partial x_i}(v)$ and $\frac{\partial f}{\partial x_i}(v \boxplus e_j)$ are known differences, the equation $\frac{\partial^2 f}{\partial x_i \partial x_j}(v) = \frac{\partial f}{\partial x_i}(v \boxplus e_j) - \frac{\partial f}{\partial x_i}(v)$ can be derived by the definition. Similarly, the ternary Gray code form can be represented as $\frac{\partial f}{\partial x_{b_1(v_k)}}(g_k) = \frac{\partial f}{\partial x_{b_1(v_k)}}(g_{k-1}) + \frac{\partial^2 f}{\partial x_{b_1(v_k)} \partial x_{b_2(v_k)}}(g_{k-1})$. In summary, we can extend the lemma to any higher degree.

Now we are going to illustrate the algorithm. The pseudocode of GTGCE is shown in Figure 3.3. At the beginning, some variables need to be initialized (line 1-5). δ, which is also used in Figure 3.2, stores $f(g_i)$ and $\delta_{\beta_1,\ldots,\beta_k}$ store all kinds of differences $\frac{\partial^k f}{\partial x_{\beta_1(v)} \cdots \partial x_{\beta_k(v)}}(g)$. Further, we need to notice that addition in the subscript of g is trit-wise operation. For example, given $\beta_1 = 2, \beta_2 = 4$, we can derive $(3^2 - 1) \boxplus (3^4 - 1) = 0022 \boxplus 2222 = 2211$, and then g_{2211} equals 2020. Therefore, the following equations can be derived.

$$\begin{aligned}\delta_{2,4} &= \frac{\partial^2 f}{\partial x_2 \partial x_4}(2020) \\ &= \frac{\partial f}{\partial x_2}(12020) - \frac{\partial f}{\partial x_2}(2020) \\ &= (f(12120) - f(12020)) - (f(2120) - f(2020)) \\ &= \mathbf{C}_{2,4} + \mathbf{C}_{2,2,4} + \mathbf{C}_{2,4,4} - \mathbf{C}_{1,2,4} - \mathbf{C}_{2,3,4}\ . \end{aligned} \tag{3.6}$$

Initialization of $\delta_{\beta_1,\ldots,\beta_k}$ is listed in Table 3.1 and Table 3.2. These differences will always stay up-to-date (correct) because their values will be updated every round in the *for* loop. After initialization, the process will enter exhaustive search stage. The stage can be divided into three steps roughly.

Table 3.1: Initialization of differences with degree = 2 in $\mathbb{F}_3$

first order		
difference	constraint	initialization
δ_0		$\mathbf{C}_0$ + $\mathbf{C}_{0,0}$
δ_i	$0 < i$	$\mathbf{C}_i$ - $\mathbf{C}_{i\text{-}1,i}$ + $\mathbf{C}_{i,i}$
second order		
$\delta_{i,i}$		- $\mathbf{C}_{i,i}$
$\delta_{i,j}$	$i < j$	$\mathbf{C}_{i,j}$

Table 3.2: Initialization of differences with degree $= 3$ in $\mathbb{F}_3$

first order		
difference	constraint	initialization
δ_0		$\mathbf{C}_0 + \mathbf{C}_{0,0}$
δ_i	$0 < i$	$\mathbf{C}_i - \mathbf{C}_{i-1,i} + \mathbf{C}_{i,i} + \mathbf{C}_{i-1,i-1,i} - \mathbf{C}_{i-1,i,i}$
second order		
difference	constraint	initialization
$\delta_{0,0}$		$- \mathbf{C}_{0,0}$
$\delta_{0,1}$		$\mathbf{C}_{0,1} - \mathbf{C}_{0,0,1} + \mathbf{C}_{0,1,1}$
$\delta_{0,j}$	$1 < j$	$\mathbf{C}_{0,j} + \mathbf{C}_{0,0,j} - \mathbf{C}_{0,j-1,j} + \mathbf{C}_{0,j,j}$
$\delta_{i,i}$	$0 < i$	$- \mathbf{C}_{i,i} - \mathbf{C}_{i-1,i,i}$
$\delta_{i,i+1}$	$0 < i$	$\mathbf{C}_{i,i+1} - \mathbf{C}_{i-1,i,i+1} - \mathbf{C}_{i,i,i+1} + \mathbf{C}_{i,i+1,i+1}$
$\delta_{i,i+t}$	$0 < i$ & $1 < t$	$\mathbf{C}_{i,i+t} - \mathbf{C}_{i-1,i,i+t} + \mathbf{C}_{i,i,i+t} + \mathbf{C}_{i,i+t,i+t} - \mathbf{C}_{i,i+t-1,i+t}$
third order		
difference	constraint	initialization
$\delta_{i,i,k}$	$i < k$	$- \mathbf{C}_{i,i,k}$
$\delta_{i,j,j}$	$i < j$	$- \mathbf{C}_{i,j,j}$
$\delta_{i,j,k}$	$i < j$ & $j < k$	$\mathbf{C}_{i,j,k}$

The first step (line 7-8) is finding corresponding indices of the differences (where the non-zero trits are, 2's counting twice) in the ternary index i. If the degree of a polynomial system is d, we only need to record d least significant nonzero bits at most.

The second step (line 9-11) is updating the variable differences and the result according to the indices determined in the first step. In Figure 3.2, the formula $\delta + = \frac{\partial f}{\partial x_{\beta_1(v_i)}}(g_{i-1})$ updates the result. However, the value of $\frac{\partial f}{\partial x_{\beta_1(v_i)}}(g_{i-1})$ must be updated with the second order difference prior to being used. In short, we add one higher-order difference into a lower-order one to get its new value, recursively. These actions are clarified by the following expression (with $\alpha \geqslant 3$):

$$\delta + = (\delta_{\beta_1} + = (\delta_{\beta_1,\beta_2} + = (\delta_{\beta_1,\beta_2,\beta_3} + = \cdots))) \ . \tag{3.7}$$

These recursive in-place prefix-sum operations do not halt until we meet a terminal condition, which means a difference that need not be updated. Moreover, we sometimes need to add more than one higher-order difference in an update. That is, having to add several $\delta_{\beta_1,\ldots,\beta_j}$ to $\delta_{\beta_1,\ldots,\beta_{j-1}}$ is possible. If this happens in a terminal step (meaning, for a degree-d system, we are using one or more order-d differences), we would always precompute the sum of all involved differences to get a new difference constant $\delta^*_{\beta_1,\ldots,\beta_j}$, and in such case only one addition is needed. However, when $d > 2$, sometimes this situation happens in the middle of the recursive process in Eq. 2, and we cannot precompute so easily because every $\delta_{\beta_1,\ldots,\beta_j}$ must be updated individually before a dependent lower-order difference can be updated. In short, the process of updating depend on indices we find in the first step. The relation between these is illustrated in Table 3.3 and Table 3.4, and we illustrate their correctness with Table 2.1.

Now we are going to introduce the terminal conditions of the recursion. Note that a difference $\delta_{\beta 1,\ldots,\beta d}$ will be initialized to $\frac{\partial^d f}{\partial x_{\beta_1(v)} \cdots \partial x_{\beta_d(v)}}(g)$, and its value is equal to or some multiple of the monomial coefficient $\mathbf{C}_{\beta 1,\ldots,\beta d}$. It depends on β_i. If every β_i is different, the multiplier is 1; for every pair of $\beta_i = \beta_{i+1}$ (which means that a trit is 2), the multiplier will be doubled; thus with ℓ equal pairs of indices, the multiple is equal to 2^ℓ. Since these differences are always constants, the multiplier values need no updates. So, there are two ways we stop the recursive updating:

1. $\alpha = d$, and we have reached the highest degree difference, which is constant.

2. $\delta_{\beta_1,\ldots,\beta_\alpha}$ appears for the first time and we use its known initial value.

The last step (line 12-14) is checking whether the new result is equal to zero or not. If the condition is satisfied, we add the corresponding ternary Gray code to the group which contains all legal solutions. Of course, an actual run starts with a script which enumerate through the indices and compute the corresponding b_i's and generate the actual C program with no unnecessary branches or table lookups. A main difference with $\mathbb{F}_2$ is the possibility of having to add several differences to update one lower-order difference, causing the number of additions per candidate to be greater than d on average for a degree-d system when $d > 2$.

```
1  $Sol \leftarrow \varnothing$;
2  $\delta \leftarrow \mathbf{C}$;
3  foreach coefficient $\boldsymbol{C}_{\beta_1,\ldots,\beta_k}$ of $f$ do
4      $\delta_{\beta_1,\ldots,\beta_k} \leftarrow \frac{\partial^k f}{\partial x_{\beta_1(3^{\beta_1}+\cdots+3^{\beta_k})} \cdots \partial x_{\beta_k(3^{\beta_1}+\cdots+3^{\beta_k})}}(\mathbf{g}_{(3^{\beta_1}-1)\boxplus\cdots\boxplus(3^{\beta_k}-1)})$;
       (see Table 3.1, Table 3.2)
5  end
6  for $i = 0$ to $3^n - 1$ do
7      $\alpha \leftarrow \min(\text{HammingWeight}(i), d)$;
8      $\beta_1, \ldots, \beta_\alpha \leftarrow b_{1,\ldots,\alpha}(i)$;
9      for $j = \alpha$ to $1$ do
10         $\delta_{\beta_1,\ldots,\beta_{j-1}} \leftarrow \delta_{\beta_1,\ldots,\beta_{j-1}} + \delta_{\beta_1,\ldots,\beta_j}$ (see Table 3.3, Table 3.4)
11     end
12     if $\delta = 0$ then
13         $Sol \leftarrow Sol \cup \{\mathbf{g}_i\}$;
14     end
15 end
16 return $Sol$;
```

Figure 3.3: Pseudocode of Generalized Ternary Gray Code Enumeration

Finally, we also analyze the performance of this method. In BTGCE, β_1 is used to determine the difference; however, we need to calculate the order-k differences in GTGCE where $1 \leqslant k \leqslant d$, thus there are d registers required for $(\beta_1, \cdots, \beta_d)$. Since the main idea of this algorithm is updating the previous result to get the next one, we need to store every initial value of differences as well, and it equals to the number of monomial terms.

On the other hand, there is an obvious improvement in its execution time. While we observe the updating step (line 10), we can find that no calculating is needed. Each initial value of differences has been computed in the initial step, hence only additions which are used to update variables remains in the *for* loop. Now the problem is how to compute the number. From Table 2.1, we find that there are three order-2 differences at most to update a order-1 difference in the quadratic system. Similarly, if we list more actions in the cubic system, we also find that there are four order-3 differences at most to update a variable. Therefore, while we consider the worst case, every order-k difference needs $(k + 2)$ order-$(k + 1)$

differences. In general, the number of differences is equal to the number of additions required. However, we have known more order-d differences can be precomputed as mentioned above so that the number of additions in order-d equals to in order-$(d-1)$. Thus the expected bit operations of each evaluation would be:

$$N_{Bit\_GTGCE}(d) = 6N_{Add\_GTGCE}(d) = \begin{cases} 6(1), & \text{if } d = 1 \\ 6(\frac{1}{2}\sum_{i=2}^{d} i\,! + \frac{d!}{2}), & \text{otherwise.} \end{cases} \tag{3.8}$$

In brief, the value of $(i+1)\,!$ is always much more than $i\,!$, hence the equation can be simplified to $N_{Bit\_GTGCE}(d) \approx 6d\,!$ except $d = 1$ and then the total number of operations with m equations is $\left[m3^n N_{Bit\_GTGCE}(d)\right]$. The time complexity of GTGCE would be $O(m3^n d\,!)$. We know that this is usually much less than the previous two methods in practical cases.

$\mathbb{F}_5$ **Version**

We do not show the algorithms about $\mathbb{F}_5$ because the scheme of algorithms are similar to those in $\mathbb{F}_3$ except some formulas. Therefore, the formulas of initialization and updating in $\mathbb{F}_5$ are listed in the Appendix.

Table 3.3: formulas of Updating in Quadratic System in $\mathbb{F}_3$

b_1	b_2	constraint	formula
0	0		$\delta_0 \mathrel{+}= \delta_{0,0}$
0	j	$0 < j$	$\delta_0 \mathrel{+}= (\delta_{0,0} + \delta_{0,j})$
i	i	$0 < i$	$\delta_i \mathrel{+}= (- \delta_{i-1,i} + \delta_{i,i})$
i	j	$0 < i$ & $i < j$	$\delta_i \mathrel{+}= (\delta_{i-1,i} + \delta_{i,i} + \delta_{i,j})$

Table 3.4: formulas of Updating in Cubic System in $\mathbb{F}_3$

b_1	b_2	b_3	constraint	formula-1[1]	formula-2	formula-3
0	0	k	$0 < k$			$\delta_{0,0} \mathrel{+}= \delta_{0,0,k}$
0	1	1				$\delta_{0,1} \mathrel{+}= (\delta_{0,0,1} + \delta_{0,1,1})$
0	j	j	$1 < j$			$\delta_{0,j} \mathrel{+}= (- \delta_{0,j-1,j} + \delta_{0,j,j})$
0	j	k	$1 < j$ & $j < k$			$\delta_{0,j} \mathrel{+}= (\delta_{0,j-1,j} + \delta_{0,j,j} + \delta_{0,j,k})$
i	i	k	$0 < i$ & $i < k$	$\delta_{i-1,i} \mathrel{+}= \delta_{i-2,i-1,i}$	$\delta_{i-1,i} \mathrel{+}= \delta_{i-1,i-1,i}$	$\delta_{i,i} \mathrel{+}= (- \delta_{i-1,i,i} + \delta_{i,i,k})$
i	$i+1$	$i+1$	$0 < i$	$\delta_{i-1,i} \mathrel{+}= \delta_{i-2,i-1,i}$	$\delta_{i,i} \mathrel{+}= \delta_{i-1,i,i}$	$\delta_{i,i+1} \mathrel{+}= (- \delta_{i-1,i,i+1} + \delta_{i,i,i+1} + \delta_{i,i+1,i+1})$
i	$i+1$	k	$0 < i$ & $i+1 < k$	$\delta_{i-1,i} \mathrel{+}= \delta_{i-2,i-1,i}$	$\delta_{i,i} \mathrel{+}= \delta_{i-1,i,i}$	$\delta_{i,i+1} \mathrel{+}= (- \delta_{i-1,i,i+1} + \delta_{i,i,i+1} + \delta_{i,i+1,i+1} + \delta_{i,i+1,k})$
i	$i+t$	$i+t$	$0 < i$ & $1 < t$	$\delta_{i-1,i} \mathrel{+}= \delta_{i-2,i-1,i}$	$\delta_{i,i} \mathrel{+}= \delta_{i-1,i,i}$	$\delta_{i,i+t} \mathrel{+}= (- \delta_{i,i+t-1,i+t} + \delta_{i,i+t,i+t})$
i	$i+t$	k	$0 < i$ & $1 < t$ & $i+t < k$	$\delta_{i-1,i} \mathrel{+}= \delta_{i-2,i-1,i}$	$\delta_{i,i} \mathrel{+}= \delta_{i-1,i,i}$	$\delta_{i,i+t} \mathrel{+}= (\delta_{i,i+t-1,i+t} + \delta_{i,i+t,i+t} + \delta_{i,i+t,k})$

Formula-1 exists when $1 < i$.

Chapter 4

Variants and Analysis

4.1 Partial Evaluation

Now we understand how various exhaustive search solvers work, it is easy to see that they are suitable for parallelization. So we will divide an input system into multiple subsystems to be solved simultaneously.

Partial evaluation is an obvious method for splitting the problem. The main idea is to substitute all possible values for s variables. Therefore, we can use s to control the number of subsystems. It generates 3^s subsystems each with $n - s$ variables.

We illustrate with an example as below. Consider a system with $d = 2, n = 4$, and choose $s = 2$. Hence there are 4 variables, which are x_0, x_1, x_2 and x_3 in the input system, and x_2 and x_3 will be substituted to find 9 subsystem individually with only variables x_0 and x_1. After reorganizing the coefficients, we can obtain the following expression:

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

It is easy to see that coefficients of the highest degree still retains their original values. The new constant is $\mathbf{C}_{2,2}x_2^2 + \mathbf{C}_{3,3}x_3^2 + \mathbf{C}_{2,3}x_2x_3 + \mathbf{C}_2x_2 + \mathbf{C}_3x_3 + \mathbf{C}$; the new coefficient of x_0 is $\mathbf{C}_{0,2}x_2 + \mathbf{C}_{0,3}x_3 + \mathbf{C}_0$ and the new coefficient of x_1 is $\mathbf{C}_{1,2}x_2 + \mathbf{C}_{1,3}x_3 + \mathbf{C}_1$. We may substitute x_2 and x_3 with their different possible values (0,1 or 2) for each subsystem so that 9 subsystems will be derived. Since there may be many substituted variables, we can use the same Generalized Ternary Gray Code enumeration technique.

4.2 Early-abort Strategy

In contrast with partial evaluation, an early-abort strategy focuses on equations. Each candidate vector is first checked against a fixed portion of the equations. Only if the candidate passes that test do we check whether it satisfies the remaining equations.

This method is like a filter, which removes impossible vectors early. We call the first part "enumeration phase", and call the second part "check phase". Suppose a system has n variables and m equations. There are total 3^n candidate vectors which need to be evaluated. Then we check only the first k equations in enumeration phase. On average only $\frac{1}{3^k}$ vectors will pass the filter so that the number of possible vectors decreases obviously.

Chapter 5

Implementations

5.1 On GPU

5.1.1 Overview

We start with an input system with n variables and m equations and we compute all differences which will be used in the enumeration. We then divide the system into numerous subsystems by guessing s variables, with each subsystem going to one thread on the GPU. The differences for 3^s subsystems may not be all the same due to the substitutions. Every thread will only test the first 32 equations with GTGCE and return at most one candidate vector which satisfies all equations during in enumeration phase. All possible solutions are then checked against the other equations. If more than one vector is found in a thread, we will need to perform GTGCE again on the CPU during the check phase because we can only return one, as in the details below.

We test 32 equations simultaneously in the enumeration phase since that is the width of a GPU register. The two bits in the 2-bit representation of each of the 32 trit-results are split into two registers to take full advantage of bit-slicing. Addition (and occasional subtraction) uses only bitwise AND, OR and XOR with no carries.

5.1.2 Unrolling

When we update the result of $f(g_i)$, accumulating is a necessary procedure. However, another thing which is almost as important as that is finding indices, and it occupies a lot of time in this stage. For this reason, unrolling is an intuitive method for decreasing the overhead.

We illustrate the method with Table 5.6. It is a scheme of unrolling with unroll factor 27. Suppose the system is cubic, so we only need to consider three b_k. Recall that $b_k(i)$ represents the index of the k-th least significant nonzero bit in ternary index i. These columns indicate some b_k are fixed even if we do not know the values of higher trits. For example, $b_1(* \cdots * 012)$ is a constant. Further, we can determine unknown items only if all b_k in the first index, which is $(* \cdots * 000)$, are evaluated. The reason is that every series of higher trits is equal in the same scheme, thus we use these b_k's repeatedly in the other indices. For example, $b_1(* \cdots * 001) = b_0(* \cdots * 000)$.

Let us formulate the description above. We consider any scheme of unrolling with unroll factor 3^u, and the first index is i. The other indices in the same scheme can be defined as $i' = i + k$, where $0 < k < 3^u$. Hence the indices of the HammingWeight(k) least significant nonzero bits in i' are constants. These values can be computed before enumeration phase. In contrast, there are still some $b_j(i')$ which can not be known beforehand (HammingWeight(k) $< d$). However, these indices can be determined by $b_j(i') = b_{j-h}(i)$, where $h =$ HammingWeight(k) and $j > h$.

Table 5.6: A Scheme of Unrolling with Unroll Factor 27

index	b_4	b_3	b_2	b_1	index	b_4	b_3	b_2	b_1	index	b_4	b_3	b_2	b_1
$* \cdots * 000$	β_4	β_3	β_2	β_1	$* \cdots * 100$	β_3	β_2	β_1	2	$* \cdots * 200$	β_2	β_1	2	2
$* \cdots * 001$	β_3	β_2	β_1	0	$* \cdots * 101$	β_2	β_1	2	0	$* \cdots * 201$	β_1	2	2	0
$* \cdots * 002$	β_2	β_1	0	0	$* \cdots * 102$	β_1	2	0	0	$* \cdots * 202$	2	2	0	0
$* \cdots * 010$	β_3	β_2	β_1	1	$* \cdots * 110$	β_2	β_1	2	1	$* \cdots * 210$	β_1	2	2	1
$* \cdots * 011$	β_2	β_1	1	0	$* \cdots * 111$	β_1	2	1	0	$* \cdots * 211$	2	2	1	0
$* \cdots * 012$	β_1	1	0	0	$* \cdots * 112$	2	1	0	0	$* \cdots * 212$	2	1	0	0
$* \cdots * 020$	β_2	β_1	1	1	$* \cdots * 120$	β_1	2	1	1	$* \cdots * 220$	2	2	1	1
$* \cdots * 021$	β_1	1	1	0	$* \cdots * 121$	2	1	1	0	$* \cdots * 221$	2	1	1	0
$* \cdots * 022$	1	1	0	0	$* \cdots * 122$	1	1	0	0	$* \cdots * 222$	1	1	0	0

5.1.3 Returning

Let us recap the pseudocode in Figure 3.3. When a candidate vector satisfies all equations in enumeration phase it is added into a set of candidate solutions. Nevertheless there are some problems even with this simple approach. One problem is the limitation of memory, we may need to keep too many candidates. Another problem is synchronization on GPU. Every thread performs individually and finds their respective solutions. Integrating all of the solutions into an appropriate form is a complicated mission.

We use a similar techniques as in [6] to avoid branching or collecting too many solutions. Each thread has two variables: *count* the total number of possible vectors and *sol* the last candidate vector found. Since the thread keeps only one possible result. We also only care about *count* in three states: 0, 1 or 2+ (2 solutions are too many!). Since *sol* by design takes less than 32 bits, we simply assign the most significant trit (MST) of *sol* to 1 if *count* = 2+, and only returns we check whether more than one vector is founded or not by MST in CPU. Note that *sol* is always less than 32-bit, so changing the highest trit does not affect the correctness of the solution.

By the way, each of the threads does not calculate the result of $f(0,\ldots,0)$ in enumeration phase. It will be checked individually in CPU. Therefore, that *sol* equals zero represents no legal solution rather than an all zero solution.

5.1.4 Re-enumeration

After executions of enumeration phase, the candidate vectors which pass the first 32 equations will be tested for the rest of input equations in check phase. According to the previous section, we know that the number of solutions can be determined by the value of MST. If it exists only one, we evaluate the equations over this vector; if there are more than one, we will execute GTGCE with the first 32 equations again on CPU, which we call re-enumeration. *sol* can be used in this stage as well.

Since enumeration is performed in ternary Gray code order and *sol* is the last legal solution, re-enumeration can terminate early if the order of vector we are processing is larger than the order of *sol*. In addition, if any possible solution is found, it will be checked for the other equations immediately. The reason is that the cost of branches on CPU is less than GPU.

However, re-enumeration will spend a lot of time so that we do not want to encounter this situation. To avoid re-enumeration, we control the number of variables in every thread. That is to say, we determine variable s in section 4.1 to make each of the threads finds only one solution at most.

5.2 On CPU

5.2.1 Overview

The input system has n variables and m equations and all differences are computed initially. Next, the system is divided into various subsystems. This is "partial evaluation" similar to what we execute on GPU. At this point every thread processes the first 32 equations on GPU. On a CPU, each subsystem will have only 16 equations tested at a time so that a register can hold and therefore process information from more systems at the same time.

Another difference between CPU and GPU is the checking phase after enumeration. A possible solution is sent to the check queue immediately. The reason is that a CPU is comparatively much more efficient at branching and threading. It is because branches are so costly on a GPU, that there is no checking phase until all work on GPU is finished.

5.2.2 Batched Enumeration

In enumeration phase on CPU, we take advantage of the 128-bit XMM registers, and we make each of the subsystems process the first 16 equations. That is, all differences are only 16-bit. In this way, we can execute 8 subsystems simultaneously. Since the type of value is __int128, accumulating can be done by the intrinsics _mm_and_si128, _mm_or_si128 and _mm_xor_si128. However, that a result is equal to zero does not represent a legal solution owing to 8 solutions in a register. We use some SSE2 intrinsics to solve this problem similar to [6].

The method is shown in Figure 5.1. The intrinsic _mm_cmpeq_epi16 compares the 8 unsigned 16-bit blocks in res_x1 and the 8 unsigned 16-bit blocks in zero, which is a 128-bit all-zero variable, for equality. res_x1 represents the higher bit in a trit. If two corresponding blocks are equal in two variables, the same position of the block in $Mask$ will be assigned to 0xFFFF, and 0x0000 otherwise. Next, the intrinsic _mm_movemask_epi8 creates a 16-bit $mask$ from the most significant bits of the 16 unsigned 8-bit blocks in $Mask$; hence that $mask$ is not equal to zero represents at least one solution exists. Because a 16-bit block is all-zero in res, it makes a bit in $mask$ equals 1. Any candidate vector found is sent to the check queue to have the rest of equations tested. Note that we check every bit in $mask$ as more than one bit may be set, which means that more than one subsystem is satisfied.

```
Mask_x1 = _mm_cmpeq_epi16(res_x1, zero);
mask = _mm_movemask_epi8(Mask_x1);
if(mask) check(mask, idx, x1, x0);
```

Figure 5.1: A Few Lines of SSE2 Intrinsics in CPU Implementations

Chapter 6

Empirical Results and Discussion

We tabulate all results in the appendix. We can see that we can solve a 30-variable, 30-equation system over $\mathbb{F}_3$ on a GeForce GTX980 Ti in 14 minutes. The same run on an AMD FX-8350 core (4 GHz) takes 32 hours. For 20 variables and 20 equations, the GPU takes 0.21 seconds, and the CPU takes 1.2 seconds.

We tested MAGMA-2.21 on the same CPU with guessing (F$\mathbf{F_4}$ or Hybrid Approach) for the same systems and tabulate the results in the appendix. Not all runs are complete, as in some cases we only ran sufficiently many subsystems to ensure that the run with the correct guess and a run with an incorrect guess takes comparable amounts of time. From these data we may extrapolate where the Gröbner bases method will catch up to enumeration on the same CPU. *Going by just the endpoints, we expect that F$\mathbf{F_4}$ to catch up to enumeration at 58 equations and variables (2^{92} complexity) and if we use the regression line, the crossover point would be 60 equations and variables (2^{95} complexity).*

There are a number of caveats to this comparison. While MAGMA has well-optimized linear algebra and uses semi-sparse operations where possible, it is not specifically tuned for $\mathbb{F}_3$, and we expect that a tuned solver would do better. In the other direction, Gröbner basis methods takes a huge penalty once the state becomes larger. Also, GPUs can be used effectively in enumeration, speeding it up by a factor of two orders of magnitude on a per-dollar basis. We can also expect that an FPGA implementation similar to [7] would speed enumeration up by another order of magnitude. Factoring in everything, it is likely that for a generic quadratic system with as many equations as variables over $\mathbb{F}_3$, enumeration will be better than algebraic solvers for all tractable problem sizes. If we repeat the same computation for $\mathbb{F}_4$, we can estimate the crossover point to be close to 40 equations and variables, or 2^{80} complexity level (close to the limit of researchers' resources). Finally, for $\mathbb{F}_5$ we estimate that the crossover point would be as low as 2^{60} complexity, well within reach of academics.

One *big* difference from the $\mathbb{F}_2$ case: We extrapolate that F**F₄** will catch up to enumeration on the same CPU at $n = 30$, $m = 60$. After taking into consideration special hardware, we estimate algebraic solvers to match enumeration for the $m/n = 2$ overdetermined generic case in $\mathbb{F}_3$ around the 2^{64} (40 trits) level.

Table 6.1: Enumeration Performance on GPU in $\mathbb{F}_3$(nVidia GeForce GTX 980 Ti)

quadratic system					cubic system				
equations	variables	guesses	unroll	time(sec.)	equations	variables	guesses	unroll	time(sec.)
20	20	9	5	0.21	20	20	8	4	0.31
22	22	9	5	0.34	22	22	9	4	0.71
24	24	11	5	1.27	24	24	9	4	3.19
26	26	11	5	8.75	26	26	10	4	26.34
28	28	12	5	86.72	28	28	11	4	237.98
30	30	13	5	788.83	30	30	11	4	2143.35

Table 6.2: Enumeration Performance on 1 CPU core in $\mathbb{F}_3$(AMD FX-8350 4GHz)

quadratic system					cubic system				
equations	variables	guesses	unroll	time(sec.)	equations	variables	guesses	unroll	time(sec.)
20	20	8	5	1.20	20	20	8	5	1.80
22	22	8	5	8.40	22	22	8	5	17.40
24	24	8	5	76.80	24	24	8	5	159.60
26	26	8	5	691.20	26	26	8	5	1426.80
28	28	8	5	6241.80	28	28	8	5	12804.60
30	30	8	5	56140.80	30	30	8	5	115560.60

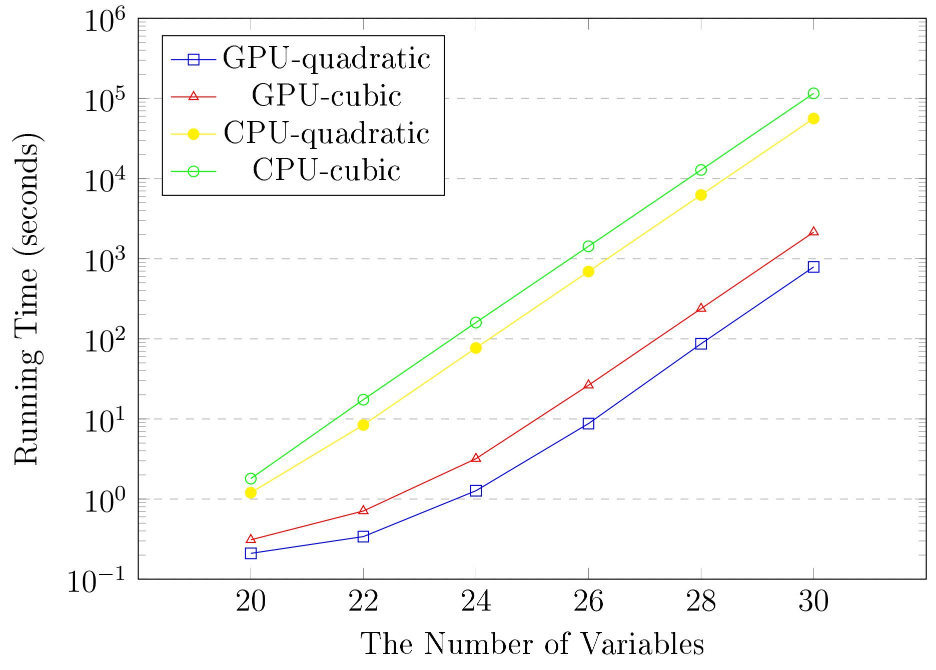

Figure 6.1: Comprehensive Results in $\mathbb{F}_3$

Table 6.3: Enumeration Performance on GPU in $\mathbb{F}_5$(nVidia GeForce GTX 980 Ti)

quadratic system					cubic system				
equations	variables	guesses	unroll	time(sec.)	equations	variables	guesses	unroll	time(sec.)
14	14	6	3	0.37	14	14	6	2	0.62
15	15	7	3	0.72	15	15	6	2	1.57
16	16	7	3	2.19	16	16	7	2	5.08
17	17	8	3	8.84	17	17	7	2	22.53
18	18	8	3	47.28	18	18	7	2	119.85
19	19	9	3	246.52	19	19	8	2	600.06
20	20	9	3	1249.83	20	20	8	2	3014.25

Table 6.4: Enumeration Performance on 1 CPU core in $\mathbb{F}_5$(AMD FX-8350 4GHz)

quadratic system					cubic system				
equations	variables	guesses	unroll	time(sec.)	equations	variables	guesses	unroll	time(sec.)
14	14	5	3	5.42	14	14	5	3	17.96
15	15	5	3	27.05	15	15	5	3	89.40
16	16	5	3	135.21	16	16	5	3	447.51
17	17	5	3	678.35	17	17	5	3	2225.90
18	18	5	3	3375.85	18	18	5	3	11217.34
19	19	5	3	17174.18	19	19	5	3	56001.30
20	20	5	3	87865.88	20	20	5	3	278696.14

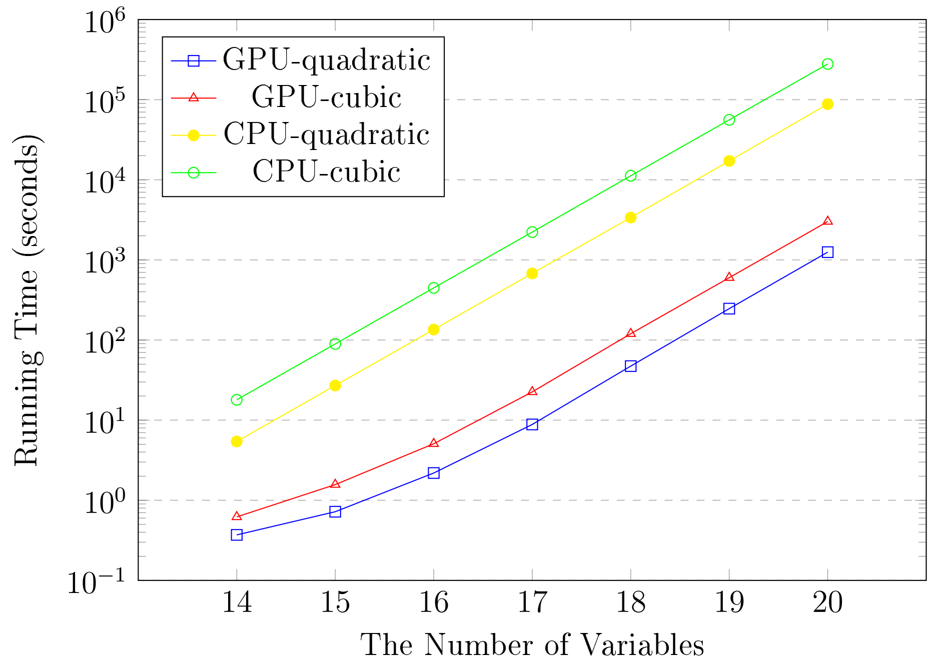

Figure 6.2: Comprehensive Results in $\mathbb{F}_5$

Table 6.5: Performance Results of Quadratic System in MAGMA

Testing platform: AMD FX(tm)-8350 Eight-Core @ 4 GHz											
$\mathbb{F}_3$				$\mathbb{F}_4$				$\mathbb{F}_5$			
equ.	var.	gue.	time(sec.)	equ.	var.	gue.	time(sec.)	equ.	var.	gue.	time(sec.)
18	18	8	190.27	16	16	5	111.62	14	14	5	59.38
19	19	8	452.71	17	17	5	316.42	15	15	5	153.13
20	20	9	1180.98	18	18	6	937.98	16	16	5	684.38
21	21	10	2893.40	19	19	7	2768.90	17	17	5	2059.38
22	22	11	6908.73	20	20	8	7143.42	18	18	5	7281.25
23	23	12	20726.20	21	21	8	22937.60	19	19	6	27500.00
24	24	12	52612.66	22	22	8	64225.28	20	20	7	88984.38
25	25	12	121699.99	23	23	8	174325.76	21	21	7	290546.88
26	26	12	291761.11	24	24	9	495452.16	22	22	8	1066015.63
27	27	13	829047.96								
28	28	14	2008846.98								
29	29	14	5213436.21								
30	30	15	13186645.53								
40	20	0	83.60	40	20	0	24.96	40	20	0	89.48
42	21	0	200.72	42	21	0	54.09	42	21	0	230.16
44	22	0	471.91	44	22	0	119.51	44	22	0	517.67
46	23	0	971.35	46	23	0	272.08	46	23	0	1141.19
48	24	0	2857.02	48	24	0	2028.01	48	24	0	3174.25
50	25	0	7277.18	50	25	0	4873.02	50	25	0	8332.86

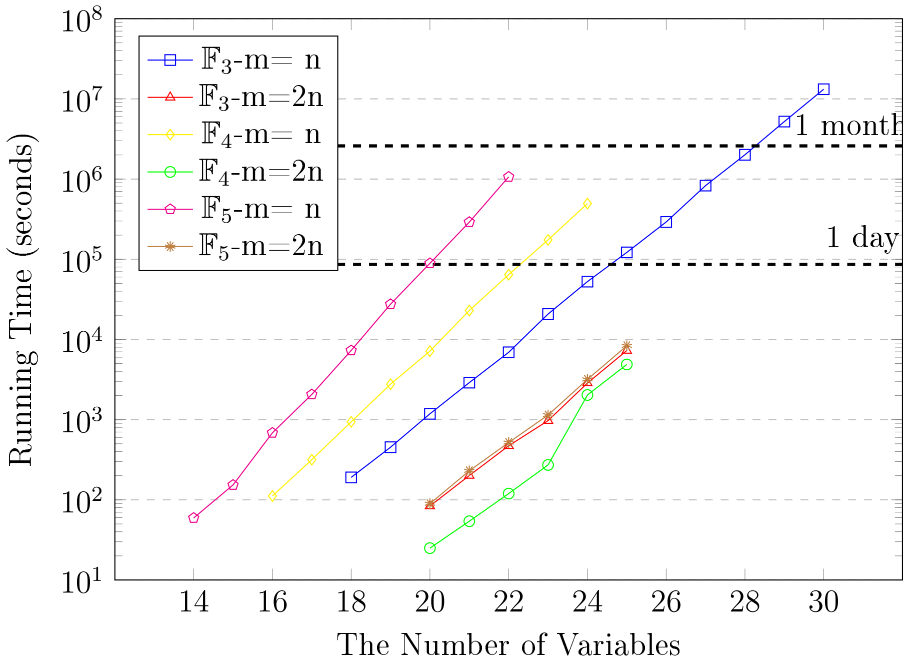

Figure 6.3: Comprehensive Results in MAGMA

Bibliography

[1] M. Bardet, J.-C. Faugère, and B. Salvy. On the complexity of Gröbner basis computation of semi-regular overdetermined algebraic equations. In *Proceedings of the International Conference on Polynomial System Solving*, pages 71–74, 2004. Previously INRIA report RR-5049.

[2] M. Bardet, J.-C. Faugère, B. Salvy, and B.-Y. Yang. Asymptotic expansion of the degree of regularity for semi-regular systems of equations. In P. Gianni, editor, *MEGA 2005 Sardinia (Italy)*, 2005.

[3] Côme Berbain, Henri Gilbert, and Jacques Patarin. QUAD: A practical stream cipher with provable security. In Serge Vaudenay, editor, *Advances in Cryptology — EUROCRYPT 2006*, volume 4004 of *Lecture Notes in Computer Science*, pages 109–128. Springer, 2006.

[4] L. Bettale, J.-C. Faugère, and L. Perret. Hybrid approach for solving multivariate systems over finite fields. *Journal of Mathematical Cryptology*, 3(3):177–197, 2010.

[5] Charles Bouillaguet. libFES: Fast exhaustive search for polynomial systems over $\mathbf{F_2}$. `http://www.lifl.fr/~bouillag/fes/`.

[6] Charles Bouillaguet, Hsieh-Chung Chen, Chen-Mou Cheng, Tung Chou, Ruben Niederhagen, Adi Shamir, and Bo-Yin Yang. Fast exhaustive search for polynomial systems in $\mathbb{F}_2$. In Stefan Mangard and François-Xavier Standaert, editors,

Cryptographic Hardware and Embedded Systems – CHES 2010, volume 6225 of *Lecture Notes in Computer Science*, pages 203–218. Springer, 2010. Extended Version: `http://www.lifl.fr/~bouillag/pub.html`.

[7] Charles Bouillaguet, Chen-Mou Cheng, Tung Chou, Ruben Niederhagen, and Bo-Yin Yang. Fast exhaustive search for quadratic systems in $\mathbf{F_2}$ on FPGAs. In Tanja Lange, Kristin E. Lauter, and Petr Lisonek, editors, *Selected Areas in Cryptography - SAC 2013 - 20th International Conference, Burnaby, BC, Canada, August 14-16, 2013, Revised Selected Papers*, volume 8282 of *Lecture Notes in Computer Science*, pages 205–222. Springer, 2013.

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

[9] Nicolas Courtois, Gregory V. Bard, and David Wagner. Algebraic and slide attacks on KeeLoq. In Kaisa Nyberg, editor, *Fast Software Encryption — FSE 2008*, volume 5086 of *Lecture Notes in Computer Science*, pages 97–115. Springer, 2008.

[10] Nicolas Courtois, Louis Goubin, and Jacques Patarin. *SFLASH, A Fast Asymmetric Signature Scheme for Low-Cost Smartcards: Primitive Specification*, 2002. Second Revised Version, `https://www.cosic.esat.kuleuven.be/nessie/tweaks.html`.

[11] Jean-Charles Faugère. A new efficient algorithm for computing Gröbner bases (F_4). *Journal of Pure and Applied Algebra*, 139(1–3):61–88, June 1999.

[12] Jean-Charles Faugère. A new efficient algorithm for computing Gröbner bases without reduction to zero (F_5). In *International Symposium on Symbolic and Algebraic Computation — ISSAC 2002*, pages 75–83. ACM Press, July 2002.

[13] Jean-Charles Faugère and Antoine Joux. Algebraic cryptanalysis of Hidden Field Equations (HFE) using Gröbner bases. In *Advances in Cryptology — CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 44–60. Dan Boneh, ed., Springer, 2003.

[14] Michael R. Garey and David S. Johnson. *Computers and Intractability — A Guide to the Theory of NP-Completeness.* W.H. Freeman and Company, 1979. ISBN 0-7167-1044-7 or 0-7167-1045-5.

[15] Dah-Jyh Guan. Generalized gray code with applications. *Proc. Natl. Sci. Council R.O.China (A)*, 22:841–848, 1998.

[16] MAGMA project, Computational Algebra Group, University of Sydney. The MAGMA computational algebra system for algebra, number theory and geometry. http://magma. maths.usyd.edu.au/magma/.

[17] Jacques Patarin. Hidden field equations (HFE) and isomorphisms of polynomials (IP): Two new families of asymmetric algorithms. In Ueli Maurer, editor, *Advances in Cryptology — EUROCRYPT 1996*, volume 1070 of *Lecture Notes in Computer Science*, pages 33–48. Springer, 1996.

[18] Jacques Patarin, Nicolas Courtois, and Louis Goubin. QUARTZ, 128-bit long digital signatures. In David Naccache, editor, *Topics in Cryptology — CT-RSA 2001*, volume 2020 of *Lecture Notes in Computer Science*, pages 282–297. Springer, 2001.

[19] Wikipedia. n-nary gray code. Version of 04:13, 13 February 2016.

[20] Bo-Yin Yang, Jiun-Ming Chen, and Nicolas Courtois. On asymptotic security estimates in XL and Gröbner bases-related algebraic cryptanalysis. In Javier Lopez, Sihan Qing, and Eiji Okamoto, editors, *Information and Communications Security — ICICS 2004*, volume 3269 of *Lecture Notes in Computer Science*, pages 401–413. Springer, October 2004.

[21] Jenny Yuan-Chun Yeh, Chen-Mou Cheng, and Bo-Yin Yang. Operating degrees for XL vs. F4/F5 for generic MQ with number of equations linear in that of variables. In Marc Fischlin and Stefan Katzenbeisser, editors, *Number Theory and Cryptography - Papers in Honor of Johannes Buchmann on the Occasion of His 60th Birthday*, volume 8260 of *Lecture Notes in Computer Science*, pages 19–33. Springer, 2013.

Chapter 7

Appendix

Table 7.1: Formulas in $\mathbb{F}_5$

	$z = 2x$	$z = 3x$	$z = 4x$	$z = x^2$	$z = x^3$
z_2	x_2	x_2	x_2	x_2	x_2
z_1	$x_1 \oplus x_0$	$x_2 \oplus x_1 \oplus x_0$	$x_2 \oplus x_1$	x_0	$x_1 \oplus x_0$
z_0	$x_2 \oplus x_0$	$x_2 \oplus x_0$	x_0	0	x_0

Table 7.2: 2-quint Quinary Gray Code with Index and Enumeration Actions

index	code	b_1	b_2	b_3	actions (quadratic)	actions (cubic)
00	00	-1	-1	-1		
01	01	0	-1	-1	$\delta += \delta_0$	$\delta += \delta_0$
02	02	0	0	-1	$\delta += (\delta_0 += \delta_{0,0})$	$\delta += (\delta_0 += \delta_{0,0})$
03	03	0	0	0	$\delta += (\delta_0 += \delta_{0,0})$	$\delta += (\delta_0 += (\delta_{0,0} += \delta_{0,0,0}))$
04	04	0	0	0	$\delta += (\delta_0 += \delta_{0,0})$	$\delta += (\delta_0 += (\delta_{0,0} += \delta_{0,0,0}))$
10	14	1	-1	-1	$\delta += \delta_1$	$\delta += \delta_1$
11	10	0	1	-1	$\delta += (\delta_0 += (\delta_{0,0} + \delta_{0,1}))$	$\delta += (\delta_0 += ((\delta_{0,0} += \delta_{0,0,0}) + \delta_{0,1}))$
12	11	0	0	1	$\delta += (\delta_0 += \delta_{0,0})$	$\delta += (\delta_0 += (\delta_{0,0} += (\delta_{0,0,0} + \delta_{0,0,1})))$
13	12	0	0	0	$\delta += (\delta_0 += \delta_{0,0})$	$\delta += (\delta_0 += (\delta_{0,0} += \delta_{0,0,0}))$
14	13	0	0	0	$\delta += (\delta_0 += \delta_{0,0})$	$\delta += (\delta_0 += (\delta_{0,0} += \delta_{0,0,0}))$
20	23	1	1	-1	$\delta += (\delta_1 += (- \delta_{0,1} + \delta_{1,1}))$	$\delta += (\delta_1 += (- (\delta_{0,1} += - \delta_{0,0,1}) + \delta_{1,1}))$
21	24	0	1	1	$\delta += (\delta_0 += (\delta_{0,0} + \delta_{0,1}))$	$\delta += (\delta_0 += ((\delta_{0,0} += \delta_{0,0,0}) + (\delta_{0,1} += \delta_{0,1,1})))$
22	20	0	0	1	$\delta += (\delta_0 += \delta_{0,0})$	$\delta += (\delta_0 += (\delta_{0,0} += (\delta_{0,0,0} + \delta_{0,0,1})))$
23	21	0	0	0	$\delta += (\delta_0 += \delta_{0,0})$	$\delta += (\delta_0 += (\delta_{0,0} += \delta_{0,0,0}))$
24	22	0	0	0	$\delta += (\delta_0 += \delta_{0,0})$	$\delta += (\delta_0 += (\delta_{0,0} += \delta_{0,0,0}))$
30	32	1	1	1	$\delta += (\delta_1 += (- \delta_{0,1} + \delta_{1,1}))$	$\delta += (\delta_1 += (- (\delta_{0,1} += - \delta_{0,0,1}) + (\delta_{1,1} += (- \delta_{0,1,1} + \delta_{1,1,1}))))$
31	33	0	1	1	$\delta += (\delta_0 += (\delta_{0,0} + \delta_{0,1}))$	$\delta += (\delta_0 += ((\delta_{0,0} += \delta_{0,0,0}) + (\delta_{0,1} += \delta_{0,1,1})))$
32	34	0	0	1	$\delta += (\delta_0 += \delta_{0,0})$	$\delta += (\delta_0 += (\delta_{0,0} += (\delta_{0,0,0} + \delta_{0,0,1})))$
33	30	0	0	0	$\delta += (\delta_0 += \delta_{0,0})$	$\delta += (\delta_0 += (\delta_{0,0} += \delta_{0,0,0}))$
34	31	0	0	0	$\delta += (\delta_0 += \delta_{0,0})$	$\delta += (\delta_0 += (\delta_{0,0} += \delta_{0,0,0}))$
40	41	1	1	1	$\delta += (\delta_1 += (- \delta_{0,1} + \delta_{1,1}))$	$\delta += (\delta_1 += (- (\delta_{0,1} += - \delta_{0,0,1}) + (\delta_{1,1} += (- \delta_{0,1,1} + \delta_{1,1,1}))))$
41	42	0	1	1	$\delta += (\delta_0 += (\delta_{0,0} + \delta_{0,1}))$	$\delta += (\delta_0 += ((\delta_{0,0} += \delta_{0,0,0}) + (\delta_{0,1} += \delta_{0,1,1})))$
42	43	0	0	1	$\delta += (\delta_0 += \delta_{0,0})$	$\delta += (\delta_0 += (\delta_{0,0} += (\delta_{0,0,0} + \delta_{0,0,1})))$
43	44	0	0	0	$\delta += (\delta_0 += \delta_{0,0})$	$\delta += (\delta_0 += (\delta_{0,0} += \delta_{0,0,0}))$
44	40	0	0	0	$\delta += (\delta_0 += \delta_{0,0})$	$\delta += (\delta_0 += (\delta_{0,0} += \delta_{0,0,0}))$

Table 7.3: Initialization of differences with degree = 2 in $\mathbb{F}_5$

first order		
difference	constraint	initialization
δ_0		$\mathbf{C}_0 + \mathbf{C}_{0,0}$
δ_i	$0 < i$	$\mathbf{C}_i - \mathbf{C}_{i-1,i} + \mathbf{C}_{i,i}$
second order		
$\delta_{i,i}$		$2\mathbf{C}_{i,i}$
$\delta_{i,j}$	$i < j$	$\mathbf{C}_{i,j}$

Table 7.4: Initialization of differences with degree = 3 in $\mathbb{F}_5$

first order		
difference	constraint	initialization
δ_0		$\mathbf{C}_0 + \mathbf{C}_{0,0} + \mathbf{C}_{0,0,0}$
δ_i	$0 < i$	$\mathbf{C}_i - \mathbf{C}_{i-1,i} + \mathbf{C}_{i,i} + \mathbf{C}_{i-1,i-1,i} - \mathbf{C}_{i-1,i,i} + \mathbf{C}_{i,i,i}$
second order		
difference	constraint	initialization
$\delta_{0,0}$		$2\mathbf{C}_{0,0} + \mathbf{C}_{0,0,0}$
$\delta_{0,1}$		$\mathbf{C}_{0,1} - \mathbf{C}_{0,0,1} + \mathbf{C}_{0,1,1}$
$\delta_{0,j}$	$1 < j$	$\mathbf{C}_{0,j} + \mathbf{C}_{0,0,j} - \mathbf{C}_{0,j-1,j} + \mathbf{C}_{0,j,j}$
$\delta_{i,i}$	$0 < i$	$2\mathbf{C}_{i,i} + \mathbf{C}_{i-1,i,i} + \mathbf{C}_{i,i,i}$
$\delta_{i,i+1}$	$0 < i$	$\mathbf{C}_{i,i+1} - \mathbf{C}_{i-1,i,i+1} - \mathbf{C}_{i,i,i+1} + \mathbf{C}_{i,i+1,i+1}$
$\delta_{i,i+t}$	$0 < i$ & $1 < t$	$\mathbf{C}_{i,i+t} - \mathbf{C}_{i-1,i,i+t} + \mathbf{C}_{i,i,i+t} + \mathbf{C}_{i,i+t,i+t} - \mathbf{C}_{i,i+t-1,i+t}$
third order		
difference	constraint	initialization
$\delta_{i,i,i}$		$\mathbf{C}_{i,i,i}$
$\delta_{i,i,k}$	$i < k$	$2\mathbf{C}_{i,i,k}$
$\delta_{i,j,j}$	$i < j$	$2\mathbf{C}_{i,j,j}$
$\delta_{i,j,k}$	$i < j$ & $j < k$	$\mathbf{C}_{i,j,k}$

Table 7.5: formulas of Updating in Quadratic System in $\mathbb{F}_5$

b_1	b_2	constraint	formula
0	0		$\delta_0 += \delta_{0,0}$
0	j	$0 < j$	$\delta_0 += (\delta_{0,0} + \delta_{0,j})$
i	i	$0 < i$	$\delta_i += (- \delta_{i-1,i} + \delta_{i,i})$
i	j	$0 < i$ & $i < j$	$\delta_i += (3\delta_{i-1,i} + \delta_{i,i} + \delta_{i,j})$

Table 7.6: formulas of Updating in Cubic System in $\mathbb{F}_5$

b_1	b_2	b_3	constraint	formula-1	formula-2	formula-3	formula-4
0	0	0					$\delta_{0,0} += \delta_{0,0,0}$
0	0	k	$0 < k$				$\delta_{0,0} += (\delta_{0,0,0} + \delta_{0,0,k})$
0	1	1				$\delta_{0,0} += \delta_{0,0,0}$	$\delta_{0,1} += \delta_{0,1,1}$
0	j	j	$1 < j$			$\delta_{0,0} += \delta_{0,0,0}$	$\delta_{0,j} += (- \delta_{0,j-1,j} + \delta_{0,j,j})$
0	1	k	$1 < k$			$\delta_{0,0} += \delta_{0,0,0}$	$\delta_{0,1} += (\delta_{0,1,1} + \delta_{0,1,k})$
0	j	k	$1 < j$ & $j < k$			$\delta_{0,0} += \delta_{0,0,0}$	$\delta_{0,j} += (3\delta_{0,j-1,j} + \delta_{0,j,j} + \delta_{0,j,k})$
1	1	1				$\delta_{0,1} += (- \delta_{0,0,1})$	$\delta_{1,1} += (- \delta_{0,1,1} + \delta_{1,1,1})$
i	i	i	$1 < i$			$\delta_{i-1,i} += (\delta_{i-2,i-1,i} - \delta_{i-1,i-1,i})$	$\delta_{i,i} += (- \delta_{i-1,i,i} + \delta_{i,i,i})$
1	1	k	$1 < k$			$\delta_{0,1} += (- \delta_{0,0,1})$	$\delta_{1,1} += (- \delta_{0,1,1} + \delta_{1,1,1} + \delta_{1,1,k})$
i	i	k	$1 < i$ & $i < k$			$\delta_{i-1,i} += (\delta_{i-2,i-1,i} - \delta_{i-1,i-1,i})$	$\delta_{i,i} += (- \delta_{i-1,i,i} + \delta_{i,i,i} + \delta_{i,i,k})$
1	2	2		$\delta_{0,1} += (- \delta_{0,0,1})$	$\delta_{0,1} += (- \delta_{0,0,1})$	$\delta_{1,1} += (3\delta_{0,1,1} + \delta_{1,1,1})$	$\delta_{1,2} += (- \delta_{0,1,2} + \delta_{1,2,2})$
i	i+1	i+1	$1 < i$	$\delta_{i-1,i} += (- \delta_{i-1,i-1,i})$	$\delta_{i-1,i} += (\delta_{i-2,i-1,i} - \delta_{i-1,i-1,i})$	$\delta_{i,i} += (3\delta_{i-1,i,i} + \delta_{i,i,i})$	$\delta_{i,i+1} += (- \delta_{i-1,i,i+1} + \delta_{i,i+1,i+1})$
1	2	k	$2 < k$	$\delta_{0,1} += (- \delta_{0,0,1})$	$\delta_{0,1} += (- \delta_{0,0,1})$	$\delta_{1,1} += (3\delta_{0,1,1} + \delta_{1,1,1})$	$\delta_{1,2} += (- \delta_{0,1,2} + \delta_{1,2,2} + \delta_{1,2,k})$
i	i+1	k	$1 < i$ & i+1 $< k$	$\delta_{i-1,i} += (- \delta_{i-1,i-1,i})$	$\delta_{i-1,i} += (\delta_{i-2,i-1,i} - \delta_{i-1,i-1,i})$	$\delta_{i,i} += (3\delta_{i-1,i,i} + \delta_{i,i,i})$	$\delta_{i,i+1} += (- \delta_{i-1,i,i+1} + \delta_{i,i+1,i+1} + \delta_{i,i+1,k})$
1	1+t	1+t	$1 < t$	$\delta_{0,1} += (- \delta_{0,0,1})$	$\delta_{0,1} += (- \delta_{0,0,1})$	$\delta_{1,1} += (3\delta_{0,1,1} + \delta_{1,1,1})$	$\delta_{1,1+t} += (- \delta_{1,t,1+t} + \delta_{1,1+t,1+t})$
i	i+t	i+t	$1 < i$ & $1 < t$	$\delta_{i-1,i} += (- \delta_{i-1,i-1,i})$	$\delta_{i-1,i} += (\delta_{i-2,i-1,i} - \delta_{i-1,i-1,i})$	$\delta_{i,i} += (3\delta_{i-1,i,i} + \delta_{i,i,i})$	$\delta_{i,i+t} += (- \delta_{i,i+t-1,i+t} + \delta_{i,i+t,i+t})$
1	1+t	k	$1 < t$ & 1+$t < k$	$\delta_{0,1} += (- \delta_{0,0,1})$	$\delta_{0,1} += (- \delta_{0,0,1})$	$\delta_{1,1} += (3\delta_{0,1,1} + \delta_{1,1,1})$	$\delta_{1,1+t} += (3\delta_{1,t,1+t} + \delta_{1,1+t,1+t} + \delta_{1,1+t,k})$
i	i+t	k	$1 < i$ & $1 < t$ & i+$t < k$	$\delta_{i-1,i} += (- \delta_{i-1,i-1,i})$	$\delta_{i-1,i} += (\delta_{i-2,i-1,i} - \delta_{i-1,i-1,i})$	$\delta_{i,i} += (3\delta_{i-1,i,i} + \delta_{i,i,i})$	$\delta_{i,i+t} += (3\delta_{i,i+t-1,i+t} + \delta_{i,i+t,i+t} + \delta_{i,i+t,k})$